Université de Mons
Faculté des Sciences
Département d'Informatique
Système d'information

# Etude de l'algorithme de Schnyder pour la représentation des graphes planaires.

Directeur : Mr Jef WIJSEN

Mémoire réalisé par
Guillaume PROOT

en vue de l'obtention du grade de
Master en Sciences Informatiques



Année académique 2022-2023

# Remerciements

Je tiens tout d'abord à remercier les différents professeurs du département d'informatique pour leurs enseignements et de m'avoir permis de réaliser ce travail.
Je remercie tout particulièrement mon directeur de mémoire, J.Wijsen, pour son accompagnement et son aide au cours de ce mémoire.
Je souhaite aussi remercier personnellement Virgil, Lisa, Maxime et Evan pour m'avoir soutenu tout le long de ces années et de ce projet.
Enfin, je remercie ma famille pour m'avoir aidé pour la relecture.

# Abstract

Les graphes planaires sont une forme très courante et utilisée de graphe dans le milieu de l'informatique. Dans ce mémoire, nous nous intéresserons à la visualisation de graphe planaire 3-connecté en implémentant un algorithme permettant cette représentation.

Pour ce faire, nous commencerons par un rappel de quelques notions essentielles sur les graphes. Il sera nécessaires pour la bonne compréhension de ce mémoire. Nous aborderons ensuite l'algorithme permettant la visualisation de ce type de graphe bien particulier.
Cet algorithme se base sur le théorème de *Walter Schnyder*[4] écrit en 1990 prouvant que pour chaque graphe planaire de n sommets où $n \geq 3$, il existe une représentation planaire sur une grille de $(n-2) \times (n-2)$.
Nous commencerons par nous attarder sur les diverses préconditions nécessaires au bon fonctionnement de cet algorithme. Nous aborderons ensuite la première phase du programme qui labélise les différents angles de chaque face de notre graphe. Par la suite, nous étudierons l'affectation d'une direction des arêtes internes et leurs numérotations. Enfin, nous nous pencherons sur le calcul des coordonnées des sommets permettant la visualisation.

# Table des matières

# Chapitre 1

# Introduction

La théorie des graphes est un domaine extrêmement vaste. L'objectif de cette discipline est d'étudier les différentes relations entre les données. Un graphe est un couple $(V, E)$ où $V$ est un ensemble de sommets et $E$ est un ensemble d'arêtes (chaque arête étant une paire contenant deux sommets). Ces graphes sont utilisés dans de nombreux domaines de l'informatique tels que le réseau, la modélisation de modèles, l'intelligence artificielle, l'analyse de risques et bien d'autres encore.

Dans ce mémoire, nous nous focaliserons sur un modèle bien précis de graphe qui se trouve être les graphes planaires et leurs représentations. Un graphe planaire est un type de graphe particulier où les arêtes qui décrivent les relations entre les données ne peuvent pas se croiser ailleurs qu'au niveau des sommets. La conception de tel graphe devient alors plus compliquée à réaliser.

Il est connu que chaque graphe planaire possède un "*straight line drawing*" c'est-à-dire une représentation où les arêtes sont des droites ne se coupant pas.

De plus, en 1990, Walter Schnyder a montré que chaque graphe planaire possède un "*straight line drawing*" sur une grille de taille $(n-2) \times (n-2)$, avec $n$ le nombre de sommets.
Plus particulièrement dans ce travail, nous étudierons la façon de représenter des graphes planaires 3-connectés sur une grille de $(n-2) \times (n-2)$ (où n est le nombre de nœuds du graphe) en un temps linéaire.

Voici un exemple de représentation planaire obtenue à l'aide de l'algorithme de Schnyder que nous aborderons lors de ce mémoire : voir figure 1.1.

Ce mémoire se découpera de la manière suivante : tout d'abord dans le chapitre 2,

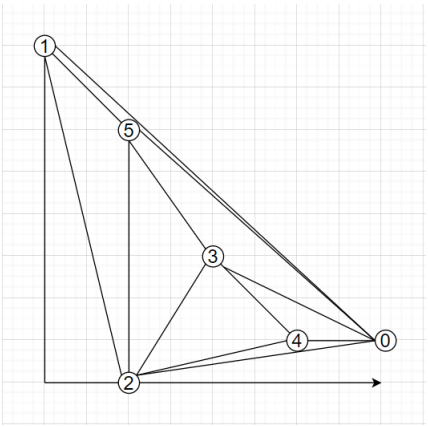



FIGURE 1.1 – Représentation planaire de graphe

nous rappellerons plusieurs notions sur les graphes utiles à la bonne compréhension de la problématique et de son implémentation. Ensuite, dans le chapitre 3, nous étudierons différentes approches utilisées pour résoudre le problème. Nous enchainerons par la suite avec les chapitres 4 et 5 où nous implémenterons l'algorithme de Schnyder afin de permettre la visualisation du graphe. Dans le chapitre 4, nous préparerons le graphe en vue de le rendre adéquat pour notre algorithme principal. Et dans le chapitre 5, nous nous pencherons sur l'algorithme à proprement parler permettant la réalisation de cette visualisation. Enfin, nous finirons par une conclusion de ce travail dans le chapitre 6.

L'implémentation de l'algorithme de Schnyder en Python se trouve en libre-accès sur le github suivant : `https://github.com/GuillaumeProot/Memoire.git`

# Chapitre 2

# Préliminaires

*Ce chapitre fait référence aux différentes notions de base de ce projet de Master. Les concepts, théorèmes et définitions sont essentiellement inspirés du livre " Planar Graph Drawing "[2], des chapitres sur les graphes et des arbres du cours de " Structure de données 1 "[12] donné par Hadrien Mélot et Véronique Bruyère ainsi que du cours de " Graphe et Optimisation combinatoire "[13] donné par Daniel Tuyttens.*

Avant d'étudier la manière dont on peut dessiner les graphes planaires 3-connecté dans un repaire dans une grille de $(n-2) \times (n-2)$, nous introduisons quelques concepts fondamentaux nécessaires à la compréhension de ce travail.

## 2.1 Graphe

**Définition 2.1.1.** *(**Graphe**). Un graphe $G = (V,E)$ est un couple où $V$ est un ensemble non vide de sommets et $E$ est un ensemble d'arêtes sous forme de paire non ordonnée $\{u, v\}$ où $u, v \in V$ avec $u \neq v$.*

Dans la suite du travail, nous supposerons que nous travaillons avec des graphes non orientés c'est-à-dire que si un sommet $u$ est connecté à un sommet $v$ par une arête $\{u, v\}$, il est également possible de se déplacer de $v$ à $u$ en utilisant la même arête.

La figure 2.1 nous montre un graphe non dirigé dont où l'ensemble $V$ est $\{1, 2, 3, 4, 5\}$ et l'ensemble $E$ est $\{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{1, 5\}, \{2, 3\}, \{2, 5\}, \{4, 5\}\}$.

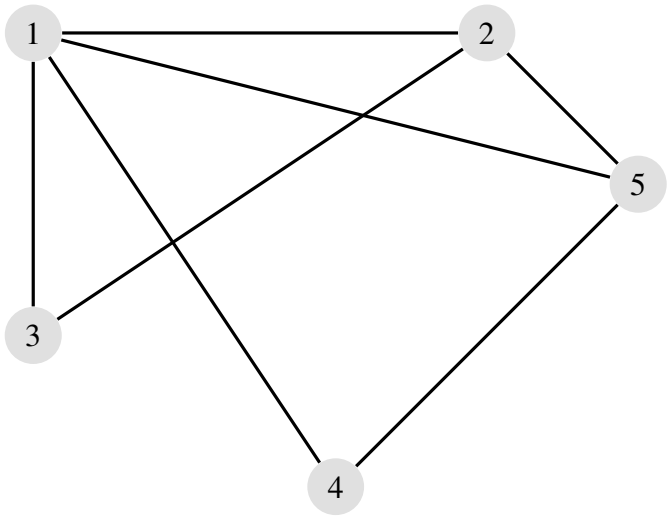



FIGURE 2.1 – Graphe à 5 sommets et 7 arêtes

**Définition 2.1.2.** *(**Graphe planaire**). Un graphe $G$ est planaire s'il existe un prolongement du graphe dans le plan euclidien tel que les sommets de $G$ correspondent à des points distincts du plan et chaque arête correspond à un chemin simple (ne passant pas deux fois par le même point) reliant ses deux sommets, et tel que deux arêtes quelconques ne se croisent qu'en leurs sommets communs.*

La figure 2.2 représente le même graphe que la figure 2.1, mais cette fois-ci nous pouvons constater qu'aucune arête ne se croise ailleurs qu'au niveau des sommets.

**Définition 2.1.3.** *(**Graphe connexe**). Un graphe est dit "connexe" si pour tout couple de sommets, il existe un chemin (une suite d'arêtes consécutives) qui les relie. Autrement dit, il est possible de se rendre de n'importe quel sommet à n'importe quel autre sommet du graphe en suivant les arêtes.*

*Plus formellement :*
*Soit G = (V, E) un graphe non orienté, avec V l'ensemble des sommets et E l'ensemble des arêtes. Un chemin entre $u$ et $v$ est une séquence $< u_0, u_1, u_2, ..., u_n >$ de sommets tels que :*

1. *$u_0 = u$, et $u_n = v$ ; et*
2. *Pour tout $i \in \{0, 1, ..., n-1, \}$ $E$ contient l'arête $\{u_i, u_{i+1}\}$*

*Un graphe est connexe s'il existe un chemin entre chaque paire de nœuds.*

Comme observé sur la figure 2.3, nous pouvons voir deux graphes. Pour la figure 2.3a on peut constater que ce graphe est bien connexe. En effet, pour chaque

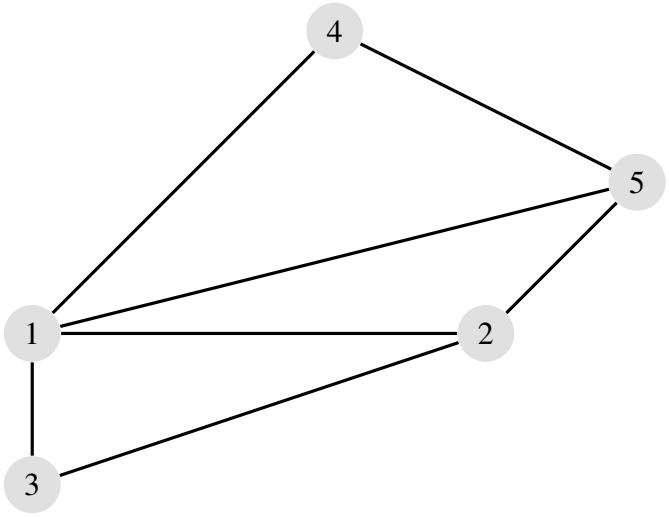
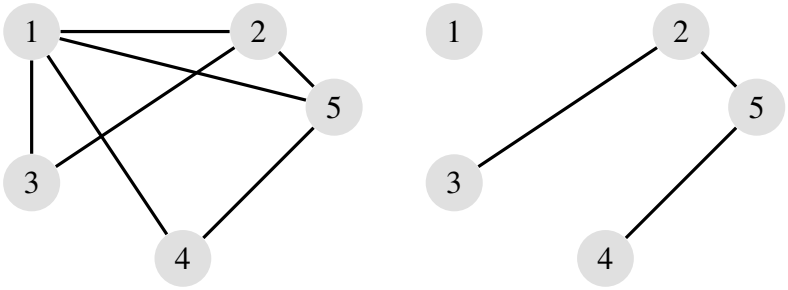



FIGURE 2.2 – Graphe planaire à 5 sommets et 7 arêtes




(a) Graphe connexe (b) Graphe non connexe

FIGURE 2.3 – Exemple de connexité d'un graphe

sommet du graphe, au moins un chemin est possible pour atteindre chaque nœud du graphe. À contrario, la figure 2.3b n'est pas connexe, car il n'existe aucun chemin permettant de relier le sommet $v1$ aux autres sommets du graphe.

**Définition 2.1.4.** *(**Graphe 3-connecté**). Un graphe tri-connecté (ou 3-connecté) est un graphe connexe qui reste connexe même si l'on supprime un ou deux sommets quelconques, avec leurs arêtes associées.*

*Plus formellement, un graphe $G$ est dit 3-connecté si, pour toutes paires $\{u, v\}$ de sommets distincts, si $s, t \in V \setminus \{u, v\}$, il existe toujours un chemin entre $s$ et $t$ dans $G$ qui ne passe ni par $u$, ni par $v$. Autrement dit, le graphe reste connexe lorsque n'importe quelles deux de ses sommets sont supprimées.*
*Un graphe $G$ doit répondre à ces 3 critères pour être dit 3-connecté :*

1. ***Connexité** : Le graphe $G$ est connexe, c'est-à-dire qu'il existe un chemin*

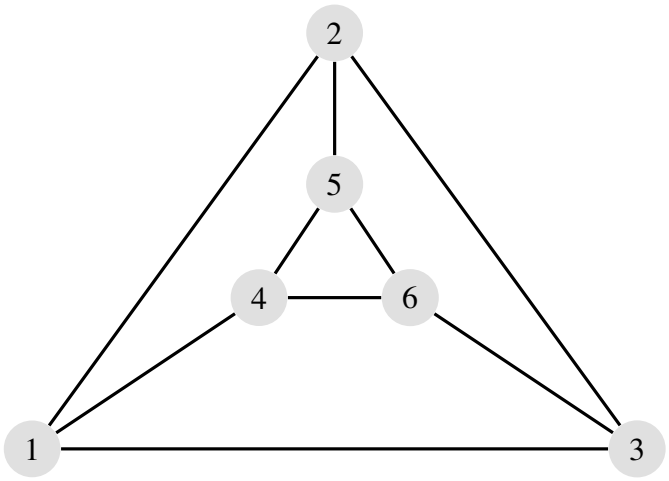



FIGURE 2.4 – Graphe 3-connecté

*entre n'importe quelle paire de sommets dans $G$.*

2. ***3-Connexe** : Le graphe $G$ est 3-connecté, ce qui signifie qu'il est connexe après la suppression de tout ensemble de deux sommets (et leurs arêtes incidentes). Formellement, si $G'$ est le graphe résultant de $G$ après la suppression de deux sommets quelconques, alors $G'$ reste connexe.*
3. ***Structure indivisible** : Le graphe $G$ ne peut pas être séparé en deux parties non vides en supprimant un sommet quelconque (ou une seule arête) du graphe. En d'autres termes, il n'existe pas de sommets ou d'arêtes dont la suppression divise le graphe en deux parties disjointes* [1].

*Un graphe tri-connecté est donc un graphe fortement connexe qui maintient sa connexité même après la suppression de n'importe quel ensemble de deux sommets.*

**Remarque 2.1.1.** *Cette notion se généralise à celle de k-connectivité : un graphe est dit k-connecté si, après la suppression de n'importe quel ensemble de $(k-1)$ sommets, le graphe reste connexe. Un graphe 3-connecté est donc aussi 2-connecté et 1-connecté.*

Comme observé sur la figure 2.4, le graphe est bien connexe et plus particulièrement, il le reste malgré la suppression d'ensemble de 2 sommets. De plus, le graphe est bien planaire et ne peut pas être divisé. Cela montre bien que notre exemple est bien un graphe 3-connecté.

1. Des sommets divisant le graphe en deux parties sont appelés points d'articulation

**Définition 2.1.5.** *(**Embedding planaire**) : Un embedding planaire, dans le contexte des graphes, est une façon de dessiner un graphe sur un plan euclidien de manière à ce que ses arêtes ne se croisent pas.*

*Plus formellement, un embedding planaire est une représentation d'un graphe dans le plan euclidien, où chaque sommet est représenté par un point distinct et chaque arête est représentée par une courbe simple (un chemin qui ne se croise pas lui-même) reliant ses deux sommets extrémités.*

*Un graphe qui admet un embedding planaire est appelé un graphe planaire. Il est important de noter que la planarité est une propriété du graphe lui-même, indépendamment de la façon dont il est dessiné. En d'autres termes, si un graphe est planaire, alors il existe au moins une façon de le dessiner sans croisements d'arêtes, même si le dessin initial peut comporter des croisements.*

Soit le graphe $G$ représenté sur la figure 2.5, on peut constater que l'embedding de ce graphe n'est pas planaire, de nombreuses arêtes se croisent tels que (4,7) et (1,2). Un embedding planaire de ce graphe est possible comme le montre la figure 2.6. Cet embedding conserve bien les propriétés du graphe initial tel que sa connexité et montre donc bien que ce graphe est planaire.

**Définition 2.1.6.** *(**Sous-graphe induit**) Un sous-graphe induit par un ensemble de partitions fait référence à un sous-graphe obtenu en choisissant un certain sous-ensemble de sommets du graphe original et en incluant toutes les arêtes qui connectent ces sommets dans le graphe original.*

*Plus formellement, soient $G = (V, E)$ un graphe non orienté et $S \subseteq V$ un sous-ensemble de sommets. Le sous-graphe induit par $S$ est le graphe $G_S = (S, E_S)$, où $E_S$ est l'ensemble des arêtes $\{u, v\} \in E$ telles que $u$ et $v$ sont tous deux dans $S$.*

On peut observer à la figure 2.7 un exemple de sous-graphe induit de la figure 2.4.

**Définition 2.1.7.** *(**Face d'un graphe**). Une face d'un graphe planaire est une région du plan délimitée par les arêtes du graphe. Autrement dit, c'est l'espace entre les arêtes tel que l'on peut le dessiner sans lever le crayon et sans traverser une arête. Chaque graphe planaire a au moins une face, appelée face externe, qui est la région du plan non contenue dans aucune autre face.*

*Formellement, on peut définir une face d'un graphe planaire comme suit :*
*Soit G = (V, E) un graphe planaire, où V est l'ensemble des sommets et E est l'ensemble des arêtes. Une face f de G est un sous-ensemble du plan défini par un*

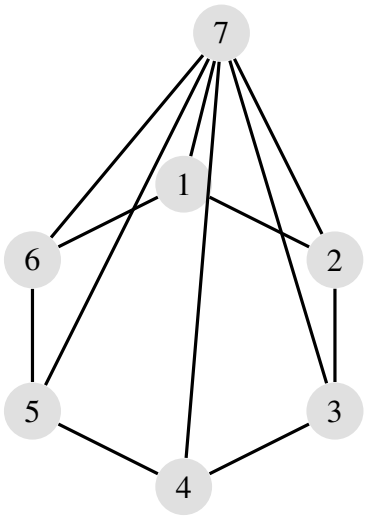



FIGURE 2.5 – Graphe $G$

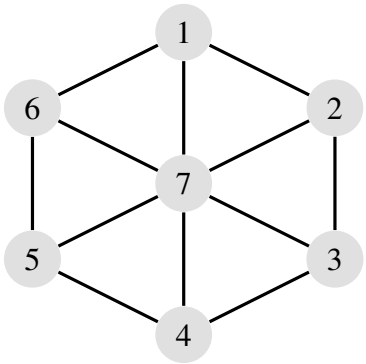



FIGURE 2.6 – Embedding planaire de $G$

*cycle fermé du graphe.*

Reprenons la figure 2.6, on peut observer une face externe $[(1,2),(2,3),(3,4),(4,5),(5,6),(6,1)]$ ainsi que 6 faces internes tel que par exemple la face $[(1,2),(2,7),(7,1)]$.

**Définition 2.1.8.** ***(Graphe triangulé)*** *Un graphe triangulé est un graphe où chaque cycle plus grand qu'un triangle inclut une arête entre deux sommets du cycle qui ne sont pas déjà voisins dans ce cycle. Cette propriété fait en sorte qu'il n'y ait pas de "trous" ou d'espaces vides à l'intérieur des cycles plus grands que trois sommets, car chaque tel cycle a au moins une arête diagonale.*

*Plus formellement, un graphe $G = (V, E)$ est dit triangulé si et seulement si, pour chaque cycle de longueur supérieure à 3 dans le graphe, il existe une arête reliant*

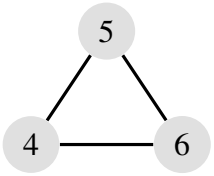



FIGURE 2.7 – Sous-graphe induit du graphe 2.4

*deux sommets non consécutifs de ce cycle. En d'autres termes, chaque cycle de longueur 4 ou plus doit avoir une arête supplémentaire reliant deux sommets non adjacents dans ce cycle.*

**Définition 2.1.9.** *(**Arête réductible**) Une arête d'un graphe est dite réductible si elle peut être contractée sans changer certaines propriétés du graphe. La contraction d'une arête consiste à fusionner les deux sommets auxquels elle est incidente en un seul sommet et à supprimer l'arête elle-même.*

*Plus formellement, la contraction d'une arête $e = \{u, v\}$ dans un graphe non orienté $G = (V, E)$ consiste à :*

1. *Supprimer les sommets $u$ et $v$.*
2. *Ajouter un nouveau sommet $w$.*
3. *Ajouter une arête $\{w, x\}$ pour chaque arête $\{u, x\}$ ou $\{v, x\}$ dans $G$, à l'exception de l'arête $e$, en s'assurant de ne pas créer de boucles.*

*Le graphe résultant est noté $G \setminus e$.*

**Définition 2.1.10.** *(**Angles**) Les angles d'un graphe planaire triangulé $G$ sont les angles de toutes les faces internes de $G$.*

**Définition 2.1.11.** *(**i-path**) Le i-path $P_i(v)$ commençant en $v$ est le chemin dans l'arbre $T_i$ [2] de $v$ à la racine $r_i$ de $T_i$. Par conséquent, $P_1(v), P_2(v), P_3(v)$ divisent le graphe en trois régions $R_1(v), R_2(v), R_3(v)$ où $R_i(v)$ représente la région opposée à $r_i$ de $T_i$. On note aussi $R_i(v)$ l'ensemble des sommets de la région incluant tous les sommets de $P_{i-1}(v)$ et $P_{i+1}(v)$. Cette définition est illustrée à la figure 2.8.*

## 2.2 Arbre

**Définition 2.2.1.** *(**Arbres**). Un arbre est une structure de données hiérarchique* [3] *et acyclique, constituée de nœuds reliés entre eux par des arêtes. Les arbres,*

2. L'arbre $T_i$ est l'arbre de racine $r_i$ tel que l'arbre est composé des arêtes de label $i$

3. c'est-à-dire qu'il est organisé en niveaux ou couches, où chaque niveau est composé de nœuds ayant une relation parent-enfant.

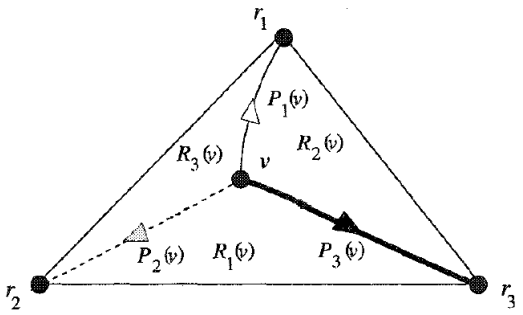



FIGURE 2.8 – I-path de $P_i(v)$[2]

*qui sont des graphes connexes et sans cycles, présentent les caractéristiques suivantes :*

- ***Nœuds** : Chaque nœud de l'arbre représente une entité ou une donnée. Chaque nœud a un seul nœud parent (à l'exception d'un nœud particulier appelé nœud racine) et peut avoir zéro, un ou plusieurs nœuds enfants. Ces nœuds peuvent être considérés comme la racine de leur propre sous-arbre.*
- ***Arêtes** : Les arêtes, aussi appelées liens, branches ou arcs, relient les nœuds entre eux. Chaque arête relie un nœud enfant à son nœud parent. Un arbre contenant $n$ nœuds a toujours $n - 1$ arêtes.*
- ***Nœud racine** : L'arbre a un nœud racine unique, qui est le seul nœud n'ayant pas de nœud parent. Il est le point d'entrée de l'arbre.*
- ***Nœuds enfants** : Chaque nœud de l'arbre peut avoir zéro ou plusieurs nœuds enfants, c'est-à-dire les nœuds qui sont reliés à lui par des arêtes sortantes.*
- ***Chemin unique** : Entre chaque paire de nœuds de l'arbre, il existe un unique chemin, ce qui signifie qu'il n'existe aucun cycle et qu'il n'y a pas de redondances*[4] *dans l'arbre. De plus, cela garantit la connectivité de l'arbre, c'est-à-dire qu'il existe un chemin entre toutes les paires de sommets.*

*Les arbres sont utilisés dans de nombreux domaines de l'informatique pour organiser et structurer des données de manière hiérarchique. Ils sont largement utilisés dans les algorithmes de recherche, les structures de données, les bases de données, les compilateurs, les systèmes de fichiers, les représentations de la syntaxe dans les langages de programmation, etc. Grâce à leur capacité à accéder, insérer, supprimer et manipuler des données de manière efficace, les arbres constituent une*

4. c'est-à-dire qu'un nœud ne peut apparaitre deux fois dans l'arbre.

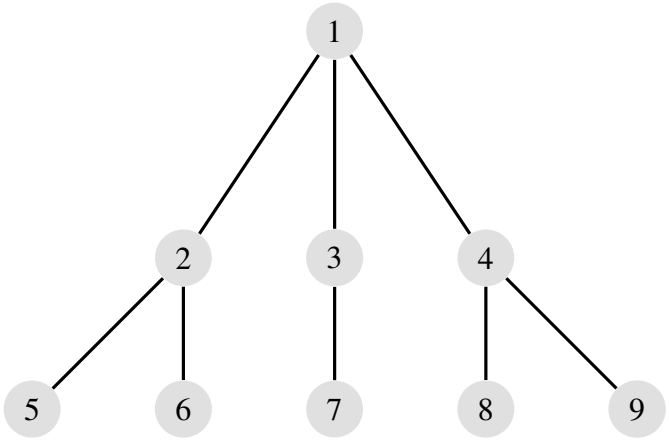



FIGURE 2.9 – Arbre de 9 nœuds et 8 arêtes

*structure de données fondamentale en informatique.*

Dans la figure 2.9 on peut retrouver un arbre de 9 nœuds (dont "1" est la racine de l'arbre) et 8 arêtes (donc bien n-1 arêtes ou n est le nombre de nœuds). On peut aussi constater que le chemin entre chaque nœud est unique et qu'il n'y a aucun cycle.

**Définition 2.2.2.** *(**Spanning tree**) Un spanning tree d'un graphe est un sous-graphe qui est un arbre et qui contient tous les sommets du graphe. Autrement dit, un arbre couvrant est un arbre formé à partir du graphe original (appelé le graphe "parent") en conservant tous les sommets, mais en supprimant certaines arêtes de manière à éliminer tous les cycles, tout en gardant le graphe connexe.*

*Plus formellement, soit $G = (V, E)$ un graphe connexe, où $V$ est l'ensemble des sommets et $E$ est l'ensemble des arêtes. Un arbre couvrant $T = (V', E')$ de $G$ est un sous-graphe de $G$ tel que :*

- *$T$ est un arbre, c'est-à-dire qu'il est acyclique et connecté.*
- *$V' = V$ et $E' \subseteq E$*

*Un graphe peut avoir plusieurs arbres couvrants différents. Par exemple, si le graphe est lui-même un arbre, alors l'arbre lui-même est son unique arbre couvrant. Si le graphe est un cycle simple, alors chaque arête du graphe peut être supprimée pour former un arbre couvrant différent.*

# Chapitre 3

# Etat de l'art

*Dans ce chapitre, nous aborderons diverses solutions déjà existantes pour résoudre la problématique de la représentation des graphes planaires en détaillant les avantages et inconvénients qu'apporte chacune de ces méthodes. Nous nous pencherons par la suite sur les outils utilisés dans ce travail pour résoudre ce problème.*

## 3.1 Anciens travaux

Penchons-nous tout d'abord sur divers travaux déjà existants permettant de résoudre de près ou de loin la problématique de la représentation de graphe planaire 3-connecté dans un repaire de taille $(n-2) \times (n-2)$.
Trois types d'algorithmes sont ressortis après recherches :

1. Les algorithmes par décomposition canonique
2. Les algorithmes par recherches de composantes triconnectées
3. L'algorithme de Schnyder

D'autres travaux ont aussi été étudiés tels que celui de Fraysseix, Path et Pollack[15] qui représentait l'*embedding* planaire sur une grille de $(n-2) \times (2n-4)$, mais étant trop anciens et servant de base aux 3 grands types d'algorithmes retenus, ne seront pas abordés dans le document.

## Les algorithmes par décomposition canonique

Ces algorithmes sont utilisés pour trouver un dessin de graphes 3-connecté planaires sur une grille de taille $(n-2) \times (n-2)$. Ces algorithmes sont basés sur la décomposition canonique de graphes 3-connecté planaire qui est la généralisation de l'ordonnancement canonique d'un graphe.

### Définition

**Définition 3.1.1.** ***(Décomposition canonique)*** *La décomposition canonique d'un graphe est un procédé divisant le graphe en un ensemble de sous-graphes satisfaisant diverses conditions. Ce procédé a un intérêt certain dans la représentation de graphe permettant ainsi de dessiner le graphe sous-graphe par sous-graphe.[1]*

*Une décomposition canonique d'un graphe planaire 3-connecté prend le graphe et le divise en sous-groupes de sommets, formant des sous-graphes. Le premier groupe de sommets contient tous les sommets de la face interne liée à la première arête, et le dernier groupe contient un sommet externe unique. Chaque sous-graphe doit être 3-connecté, ce qui signifie qu'il existe au moins trois chemins distincts entre chaque paire de sommets. De plus, les sommets dans chaque groupe suivant doivent soit former une chaîne externe et avoir au moins un voisin dans le sous-graphe suivant (voir figure 3.2), soit être un sommet unique avec au moins deux voisins dans le sous-graphe précédent et un voisin dans le suivant. Cette décomposition aide à organiser le graphe en parties qui satisfont ces conditions spécifiques, facilitant ainsi l'analyse et la manipulation du graphe.*

La figure 3.1 illustre la décomposition canonique $\Pi = (U_1, U_2, ..., U_8)$ d'un graphe planaire 3-connecté avec $n = 15$ sommets.

### Fonctionnement

La première étape de ces algorithmes est de trouver la décomposition canonique du graphe comme expliqué précédemment.

**Théorème 3.1.1.** ***(Décomposition canonique d'un graphe 3-connecté planaire)*** *Chaque graphe 3-connecté planaire de $n \geq 4$ sommet à une décomposition canonique $\Pi$, et $\Pi$ peut être trouvé en temps linéaire.*

Ce théorème non démontré dans ce travail [1] montre que l'obtention de la décomposition canonique d'un graphe respecte bien la condition de complexité

1. Pour plus de renseignement veuillez vous référer au livre Planar Graph Drawing[1]

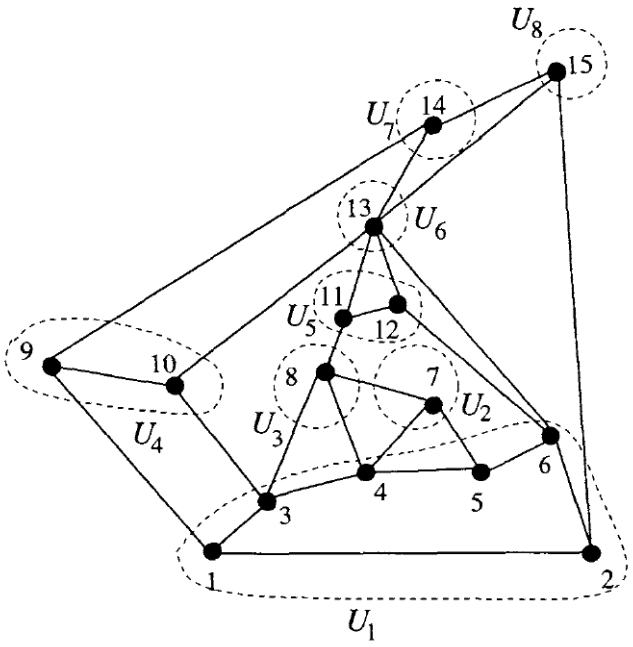



FIGURE 3.1 – Décomposition canonique d'un graphe planaire 3-connecté[1]

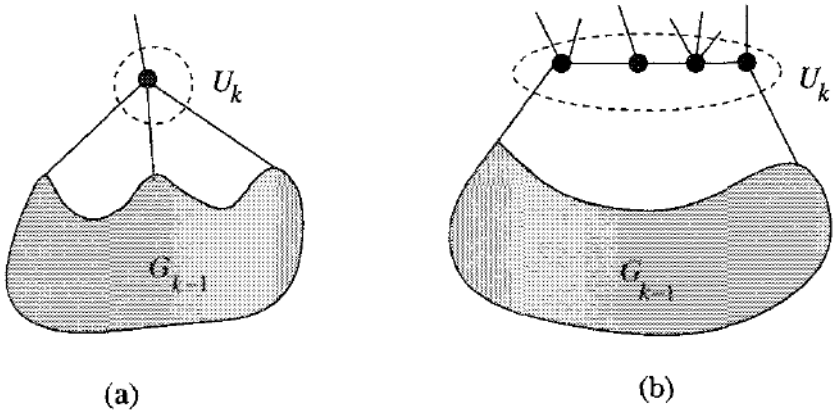



FIGURE 3.2 – $G_k$ dont plusieurs arêtes joignent $U_k$ et $\overline{G_k}$[1]

imposée par la problématique étudiée.

Une fois cette décomposition obtenue, en suivant la méthode de représentation de graphe planaire par décomposition canonique, on va pouvoir créer un embedding planaire de ce graphe. Pour ce faire, l'algorithme va ajouter les sommet de l'ensemble $\Pi$ au dessin, un par un dans l'ordre $U_1, U_2, ..., U_l$ ajustant le dessin à chaque étape.

**Avantages**

- Cette méthode de représentation de graphe planaire répond parfaitement aux critères de temps imposés par la problématique.
- La méthode répond aux critères de représentation d'embedding connexe sur une grille de $(n-2) \times (n-2)$.

**Inconvénients**

- Le gros inconvénient de cette méthode est le manque de documentation trouvée pour l'expliquer correctement.
- La méthode de placement de sommets est très complexe à cause des corrections engendrées au fur et à mesure du placement des sommets.

## Les algorithmes par recherches de composantes triconnectées

**Définition**

**Définition 3.1.2.** *(**Composantes triconnectées**[5]) Une composante d'un graphe est un sous-graphe connecté maximal. En d'autres termes, une composante est une partie du graphe où il existe un chemin entre chaque paire de nœuds, et aucun autre nœud extérieur à cette partie ne peut être ajouté sans violer la propriété de connectivité.*

*Une composante triconnectée d'un graphe est une composante d'un type particulier tel que cette composante est un sous-graphe qui est lui-même 3-connecté. Pour rappel, un graphe est dit 3-connecté (ou k-connecté avec k=3) si celui-ci n'est pas déconnecté lorsqu'on retire un ou deux sommets avec leurs arêtes adjacentes. Il faut qu'au moins trois sommets soient retirés pour que le graphe devienne non connecté.*

**Théorème 3.1.2.** ***(Composantes triconnectées)*** *Les composantes triconnectées d'un graphe $G$ sont uniques.*

### Fonctionnement

La méthode par recherche de composantes triconnectées se divise en 3 grandes étapes.

Lors de la première étape de l'algorithme de cette méthode, une recherche en profondeur sera effectuée pour trouver les composantes triconnectées du graphe. Cet partie de l'algorithme s'effectue en **O(n)** et se base sur l'algorithme de Hopcroft et Tarjan[16].

La deuxième étape de l'algorithme est la triangulation du graphe planaire. En effet, la majorité des algorithmes de représentation de graphes se basent sur des graphes triangulés, cette étape est donc nécessaire au bon fonctionnement de l'algorithme. Cet algorithme est tout comme l'étape 1 en **O(n)**.

Dans la dernière étape, on va trouver une variante pondérée par les sommets à l'aide de la méthode *shift-method* permettant de produire un dessin rectiligne de graphe triangulés sur une grille avec une limite d'aire quadratique par rapport aux nombres de sommets du graphe. Tout comme les autres étapes, l'algorithme peut se faire en **O(n)**.

### Avantages

- L'avantage principal de ce type d'algorithme est l'implémentation de celui-ci. Les deux premières étapes étant assez triviales, seul la *shift-method* est plus complexe, mais celle-ci est bien documentée.
- Assez bien de documentation et de pseudo-code sont disponibles.
- Chaque étape peut s'implémenter en **O(n)** et donc il y a bien un respect de la complexité.

### Inconvénients

- La méthode va représenter le graphe sous un format 3D et une étape supplémentaire pour transformer la 3D vers de la 2.5D [2] sera nécessaire pour le représenter sur une grille.

2. 2D avec effet de profondeur pour simuler la troisième dimension.

- L'implémentation de cet algorithme n'assure pas la représentation sur un repère de taille $(n-2) \times (n-2)$.

## L'algorithme de Schnyder

L'algorithme de Schnyder va montrer que chaque graphe planaire avec $n \geq 3$ sommets admet un *embedding* planaire sur une grille de $(n-2) \times (n-2)$ et que cet *embedding* est calculable en temps O(n).

### Définition

**Théorème 3.1.3.** ***(Théorème de Schnyder (1990))*** *Chaque graphe planaire avec $n \geq 3$ sommets admet un embedding planaire sur une grille de $(n-2) \times (n-2)$.*

Ce théorème améliore les résultats trouvés précédemment par Schnyder qui représentait l'*embedding* sur une grille de $(2n-5) \times (2n-5)$ écrit en 1989[17] ainsi que celui de Fraysseix, Path et Pollack[15] qui représentait l'embedding planaire sur une grille de $(n-2) \times (2n-4)$ écrit en 1988.

### Fonctionnement

L'entrée de l'algorithme de Schnyder est supposée être un graphe planaire, sans boucle [3] ni adjacences multiples [4], avec la topologie d'une disposition planaire déjà spécifiée en termes d'ordonnancement dans le sens des aiguilles d'une montre des arêtes à chaque sommet, comme le montrent les résultats de Hopcroft-Tarjan ou d'autres algorithmes d'incorporation planaire en temps linéaire. En outre, chaque face du plan, y compris la face extérieure, est supposée être un triangle.

Il est donc nécessaire de s'assurer que le graphe est triangulé. Pour ce faire, on va détecter les faces du graphes et travailler face par face afin de les trianguler au besoin comme sur la figure 3.3.

Une fois la triangulation effectuée, on va désigner l'une des faces du graphe comme externe, et le reste comme interne. Une arête ou un sommet est interne s'il n'appartient pas à la face externe. La méthode de Schnyder repose sur deux concepts étroitement liés dans un graphe planaire triangulé : une labelisation normale et une réalisation. (Tel que représenté à la figure 3.4).

3. Arête d'un sommet vers lui-même
4. Plusieurs arêtes reliant deux mêmes sommets

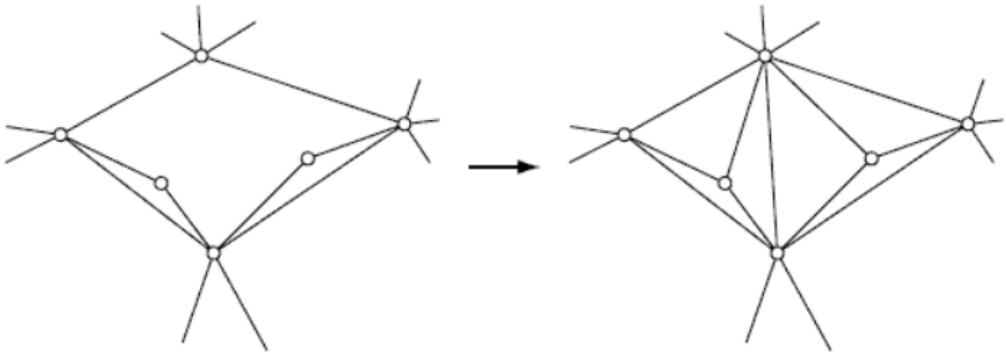


FIGURE 3.3 – Résultat de la triangulation de graphe[10]

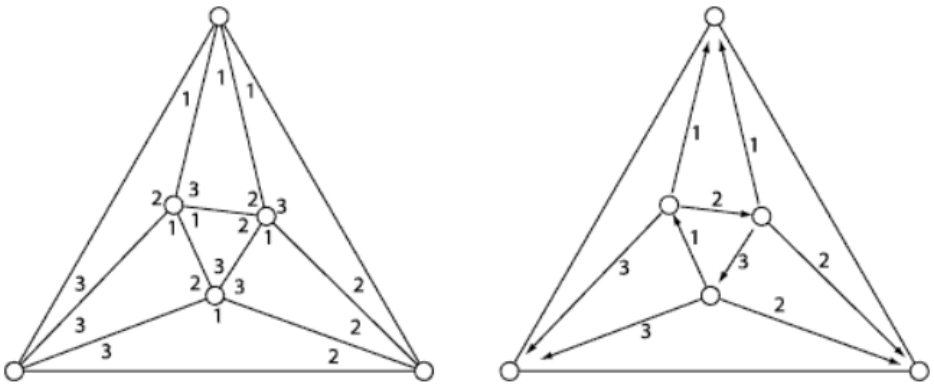


FIGURE 3.4 – Labélisation et réalisation d'un graphe[10]

Une labélisation normale est une attribution des nombres $1-2-3$ aux coins de chaque triangle intérieur, de sorte que chaque triangle ait les trois labels dans l'ordre des aiguilles d'une montre, et que les labels autour de chaque sommet interne forment des blocs contigus de $1$, de $2$ et de $3$ dans l'ordre des aiguilles d'une montre.

Une réalisation est une attribution d'une direction à chaque arête interne, et un nombre $1-2-3$ à chaque arête interne, de telle sorte que chaque sommet interne ait exactement une arête sortante avec chacun des trois nombres, ces arêtes apparaissent dans l'ordre des aiguilles d'une montre, et entre deux arêtes sortantes étiquetées $i$ et $j$, toutes les arêtes entrantes ont pour label $(6-i-j)$.

Dans la figure 3.4, nous voyons une labelisation normal et une réalisation du même graphe.

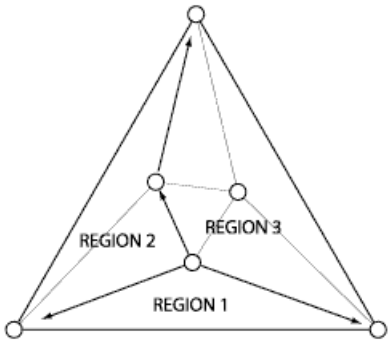



FIGURE 3.5 – Région du graphe[10]

Enfin, l'embedding du graphe est prêt à être défini dans la grille. Étant donné que le réalisateur du graphe possède un spanning tree pour chaque label, il existe un chemin d'arêtes label-i entre chaque sommet $v$ et le sommet externe $i$. Ces trois chemins divisent le graphe en trois régions comme sur la figure 3.5. Nous utilisons les nombres de triangles dans chaque région comme coordonnées tridimensionnelles de $v$ tel que la figure 3.6. Bien que les coordonnées soient tridimensionnelles, nous pouvons les considérer comme des coordonnées barycentriques dans une grille triangulaire bidimensionnelle : la position $(x, y, z)$ est $x$ lignes au-dessus du bord inférieur du triangle, $y$ lignes à droite du bord gauche et $z$ lignes à gauche du bord droit. Nous pouvons également former une disposition sur une grille carrée en supprimant la troisième coordonnée.

Schnyder poursuit en trouvant une disposition légèrement plus compacte, en comptant les sommets dans chaque région (y compris le chemin bordant la région dans le sens des aiguilles d'une montre, mais pas dans le sens inverse). Le résultat est un encastrement avec des coordonnées entières positives dans le plan $x + y + z = n - 2$, ou en deux dimensions dans une grille $(n - 2) \times (n - 2)$, en temps linéaire.

**Avantages**

- Beaucoup de documentation est présente sur cet algorithme.
- Cet algorithme correspond aux critères de complexité en **O(n)**.
- Cet algorithme garantit un embedding sur une grille de $(n-2) \times (n-2)$.

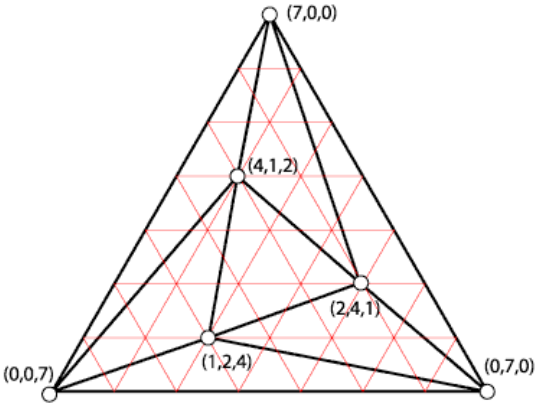



FIGURE 3.6 – Embedding du graphe[10]

**Inconvénients**

- La méthode de labélisation normale est assez complexe.
- L'algorithme ne promet pas un embedding planaire connexe directement, il est nécessaire de passer par les propriétés des embeddings afin d'assurer que celui-ci correspond bien aux critères imposés.

## 3.2 Outils

### Langage

Deux langages ont été envisagés lors de la conception de cet algorithme : **JAVA** et **PYTHON**. Dans cette section, nous allons étudier les avantages et inconvénients de chacun de ces deux langages et comprendre pourquoi le langage **PYTHON** a finalement été choisi pour la conception de ce programme.

**JAVA :**

+ Bon langage pour une interface graphique.
+ Langage orienté objet utile pour la réalisation de graphe.
+ Beaucoup de documentation.
- Peu de libraires open source existantes pour le programme.
- Langage lent rendant difficile l'exécution en **O(n)**.
- Multiplication de classe.

**PYTHON :**

+ Librairie existante pour le travail sur les graphes.
+ Beaucoup de documentation.
+ Langage rapide parfait pour des algorithmes en **O(n)**.
- Interprétation graphique plus complexe.
- Moins orienté objet et plus procédural.

## Package utilisé

### NetworkX :

Une fois le langage sélectionné, le package Python **NetworkX**[3] a été envisagé, répondant à tous les critères nécessaires à la réalisation de l'algorithme. En effet, NetworkX est un package Python pour la création, la manipulation et l'étude de la structure, de la dynamique et des fonctions des réseaux complexes. Plus particulièrement, le package permet le travail sur les graphes grâce à de nombreuses fonctions déjà existantes et la représentation de ces mêmes graphes, ce qui répond parfaitement aux critères nécessaires à la réalisation de ce mémoire. De plus, NetworkX a une large documentation permettant l'utilisation de ces fonctions.

## Environnement de travail

### Jupyter Notebook

Le choix de l'environnement de programmation s'est porté sur Jupyter Notebook[6] celui-ci ayant de nombreux avantages. Il est 100% open-source et est interactif. Il a aussi l'avantage de ne pas devoir installer les différents modules sur notre machine et permet une visualisation plus simple des graphes par rapport à Python classique. C'est pourquoi cet IDE a été choisi pour l'implémentation du programme de ce mémoire.

# Chapitre 4

# Préparation du graphe

*Dans ce chapitre, nous étudierons la façon dont le graphe doit être créé. Nous aborderons ensuite les algorithmes permettant de vérifier les caractéristiques du graphe et de le modifier au besoin pour qu'il soit accepté par l'algorithme de Schnyder.*

**Remarque 4.0.1.** *A partir de ce chapitre tous les exemples seront des exemples créés spécifiquement pour ce travail afin d'aider à la compréhension de l'article[4].*

## 4.1 Parser

*Dans cette section, nous aborderons la façon dont le fichier texte en entrée doit être agencé et la manière dont le graphe est créé.*

### Format du fichier texte

Le fichier texte en entrée représentant le graphe doit être configuré de la manière suivante :

- La première ligne est composée de deux entiers $v$ et $e$ séparé par un espace respectivement le nombre de sommets et le nombre d'arêtes du graphe.
- Les $e$ lignes suivantes sont composées de deux entiers $v_1$ et $v_2$ séparé par un espace respectivement le sommet initial et le sommet final de l'arête.

De plus les arêtes doivent déjà être triées dans le sens des aiguilles d'une montre.

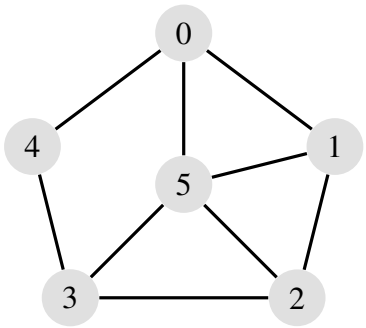



FIGURE 4.1 – Représentation de l'exemple du Parser

## Exemple de fichier texte

Prenons un graphe à 6 sommets et 9 arêtes, le fichier texte sera écrit de la manière suivante :

*6 9*
*0 1*
*1 2*
*2 3*
*3 4*
*4 0*
*0 5*
*1 5*
*2 5*
*3 5*

On peut constater que les arêtes sont bien triées dans le sens des aiguilles d'une montre. Une représentation possible de cet exemple se trouve à la figure 4.1.

## Idée générale

Le parser lit un fichier texte qui représente un graphe. La première ligne du fichier contient deux nombres : le premier entier représente le nombre de sommets et le deuxième entier est le nombre d'arêtes dans le graphe. Les lignes suivantes représentent chaque arête du graphe par deux entiers séparés par un espace. La fonction lit chaque arête et l'ajoute au graphe. Enfin, elle renvoie le graphe complet.

## Pseudo-code

**Algorithm 1** parsing

**Require:** $filename$ un fichier texte
**Ensure:** $graph$ un graphe du module NetworkX
  Initialisation de $graph$
  Ouverture du fichier
  Lire la première ligne du fichier et diviser la ligne en éléments
  Convertir le deuxième élément en un entier et l'assigner à $arete$
  **for** $i$ in range $arete$ **do**
    Lire la ligne suivante du fichier et la diviser en éléments
    Convertir les éléments en entiers pour former une arête
    Ajouter l'arête à $graph$
  **end for**
  **return** $graph$

## 4.2 Triangulation

*Dans cette section, nous étudierons la façon de trianguler un graphe c'est-à-dire le fait de vérifier et au besoin transformer chaque face du graphe en triangle.*

### Idée générale

L'idée générale est assez simple. Tout d'abord, nous allons vérifier le nombre de sommets de notre graphe. L'objectif étant de trianguler notre graphe si celui-ci est composé uniquement de 3 sommets, il suffit de vérifier si celui-ci comporte bien 3 arêtes et si pas en ajouter une nouvelle entre les sommets non reliés afin de transformer le graphe en triangle.

Dans le cas de plus de 3 sommets, nous allons lister chaque face du graphe et travailler face par face pour les trianguler au besoin.

**Remarque 4.2.1.** *Par définition d'une face d'un graphe, trianguler face par face permet de conserver la caractéristique de planarité du graphe. En effet, aucune arête ajoutée à une face ne croisera une arête d'une autre face, la triangulation se faisant à l'intérieur de la face.*

Deux cas de figure peuvent être possible :
- La face a **3 côtés** : Dans ce cas de figure, la face est déjà un triangle, il n'est donc pas nécessaire de la trianguler et on peut donc passer à la face suivante.

- La face a **4 côtés ou plus** : Dans ce cas de figure, nous allons ajouter des arêtes entre 2 sommets non directement liés de tel sorte que :
  Soit $x, y, z \in$ face on a $(x, y)$ et $(y, z)$ dans la face on ajoute l'arête $(x, z)$ afin de créer une nouvelle face : $[(x, y), (y, z), (x, z)]$. On renouvelle cette technique jusqu'à avoir parcouru entièrement la face. Une fois cela fait, une nouvelle face interne est apparue. On relance donc l'algorithme sur cette face pour vérifier si celle-ci est triangulée ou non.

## Pseudo-code

Pour cette phase de notre programme, trois algorithmes sont nécessaires afin de trianguler notre graphe.

Dans un premier temps, nous avons l'algorithme 2 $add\_edge\_to\_graph()$ permettant d'ajouter une arête au graphe ainsi qu'à la liste d'arêtes ajoutées lors de la triangulation.

**Algorithm 2** $add\_edge\_to\_graph$

**Require:** $graph$ un graphe du module NetworkX
**Require:** $edge$ une arête sous la forme $(x, y)$ où $x, y \in graph$
**Require:** $added\_edges$ la liste d'arêtes ajoutées
**Ensure:** $added\_edges$ la liste d'arêtes ajoutées avec la nouvelle arête en plus
  **if** $edge$ n'est pas dans $graph$ **then**
    Ajout de $edge$ à $graph$
    Ajout de $edge$ à $added\_edge$
  **end if**
  **return** $added\_edge$

Ensuite, nous disposons de l'algorithme 3 $recur\_face()$ permettant de trianguler une face jusqu'à ce que la face soit composée que de faces triangulées élémentaires.

Enfin, nous avons l'algorithme principal de cette étape du projet qui est l'algorithme 4 $triangulate\_graph()$ qui permet de trianguler le graphe et de retourner la liste d'arêtes ajoutées.

**Algorithm 3** $recur\_face$

**Require:** $f$ une face du graphe
**Ensure:** $new\_edge$ la liste d'arêtes ajoutées à $f$
**if** taille de f est égale à 3 **then return** une liste vide
**else**
Initialisation de $new\_face$ comme une liste vide
Définition de $size$ à la taille de $f$
Définition de $count$ à 0
Initialisation de $new\_edges$ comme une liste vide
**while** $count < size - 1$ **do**
Définition de $new\_edge$ à un tuple (f[count][0], f[count+1][1])
**if** $new\_edge$ est dans le graphe **then**
Ajout de $f[count]$ à $new\_face$
**if** $count$ est égal à $size$-2 **then**
Arrêt de la boucle
**end if**
Augmentation de $count$ de 1
continuer
**end if**
Ajout d'une arête entre $new\_edge$[0] et $new\_edge$[1] dans le graphe $G$
Ajout de $new\_edge$ à $new\_edges$
Ajout de $new\_edge$ à $new\_face$
Augmentation de $count$ de 2
**end while**
**if** $count$ n'est pas égal à $size$ **then**
Ajout de $f$[-1] à $new\_face$
**end if**
Définition de $new\_recur$ à l'appel de la fonction $recur\_faces$ avec $new\_face$ **return** la concaténation de $new\_edges$ et $new\_recur$
**end if**

**Algorithm 4** $triangulate\_graph$

**Require:** $graph$ un graphe du module NetworkX
**Require:** $neighbor\_dict$ un dictionnaire sommet, voisin
**Ensure:** $added\_edges$ la liste d'arêtes ajoutées lors de la triangulation
**if** le nombre de nœuds du graphe est inférieur à 3 ou le graphe n'est pas connecté **then**
Lancement d'une exception
**end if**
Initialisation de $added\_edges$ à une liste vide
**if** le nombre de nœuds du graphe est égal à 3 et le nombre d'arêtes du graphe est égal à 2 **then**
Définition de $edge$ à (1, 2) si le nombre d'arêtes entre 0 et 1 est supérieur à 0 et le nombre d'arêtes entre 0 et 2 est supérieur à 0, sinon
Définition de $edge$ à (0, 2) si le nombre d'arêtes entre 1 et 0 est supérieur à 0 et le nombre d'arêtes entre 1 et 2 est supérieur à 0, sinon
Définition de $edge$ à (0, 1)
**end if**
définition de $faces$ à l'appel de la fonction $find\_faces$ avec $graph$ et $neighbor\_dict$
**for** chaque $face$ dans $faces$ **do**
Définition $new\_edges$ à l'appel de la fonction $recur\_faces$ avec $face$
**if** $new\_edges$ est égal à une liste vide **then**
continuer
**else**
**for** $i$ dans $new\_edges$ **do**
Ajout de $i$ à $added\_edges$
**end for**
**end if**
**end for**
**return** $added\_edges$

## Fonctionnement

$add\_edge\_to\_graph()$ **:**

Le fonctionnement de $add\_edge\_to\_graph()$ est trivial. Il vérifie que l'arête prise en entrée appartient déjà ou non au graphe initial. Si c'est le cas alors, on retourne la liste non modifiée sinon on ajoute l'arête au graphe ainsi qu'à la liste des arêtes ajoutées et l'on retourne une liste modifiée.

$recur\_face()$ **:**

L'algorithme $recur\_face()$ est une fonction récursive qui vise à diviser une "face" d'un graphe en triangles si ce n'est pas déjà le cas.

L'algorithme fonctionne de la manière suivante :

1. Si la face $f$ est déjà un triangle (elle a donc trois sommets), la fonction retourne une liste vide, car aucun ajout d'arêtes n'est nécessaire pour trianguler la face.
2. Si la face $f$ a plus de trois sommets, l'algorithme crée une nouvelle face $new\_face$ et une liste pour stocker les nouvelles arêtes ajoutées $new\_edges$.
3. L'algorithme passe ensuite en revue les sommets de la face *f*. Pour chaque paire consécutive de sommets, il crée une nouvelle arête potentielle $new\_edge$.
4. Si cette nouvelle arête existe déjà dans le graphe *G*, l'algorithme ajoute le sommet actuel à $new\_face$ et passe au sommet suivant.
5. Si l'arête n'existe pas déjà, l'algorithme l'ajoute à la fois au graphe $G$ et à la liste $new\_edges$. Il ajoute également cette nouvelle arête à $new\_face$ et passe ensuite au sommet après le suivant.
6. Après avoir passé en revue tous les sommets, si tous les sommets n'ont pas été traités (ce qui peut arriver si la dernière arête ajoutée ne finit pas à la fin de la liste), l'algorithme ajoute le dernier sommet à $new\_face$.
7. L'algorithme fait ensuite un nouvel appel à lui-même avec la $new\_face$ comme argument, ce qui lui permet de diviser davantage les faces qui ne sont pas encore des triangles.
8. Finalement, il retourne la liste de toutes les nouvelles arêtes ajoutées.

En somme, cet algorithme fonctionne en ajoutant des arêtes à une face pour la diviser en triangles, en suivant une approche récursive.

$triangulate\_graph()$ **:**

Cet algorithme réalise la triangulation d'un graphe. Il procède de la manière suivante :

1. Tout d'abord, l'algorithme vérifie si le graphe remplit certains critères : il doit avoir au moins trois nœuds et être connexe. Si le graphe ne remplit pas ces critères, l'algorithme lève une exception.
2. Si le graphe a exactement trois nœuds et deux arêtes, alors il forme une ligne droite et l'algorithme ajoute une troisième arête pour former un triangle, puis termine.
3. Pour un graphe plus grand, l'algorithme commence par trouver toutes les faces du graphe à l'aide de la fonction $find\_faces()$.
4. Ensuite, pour chaque face du graphe, l'algorithme utilise la fonction $recur\_faces$ pour la diviser en triangles. Cette fonction renvoie une liste des nouvelles arêtes qui ont été ajoutées au graphe pour créer les triangles.
5. Si $recur\_faces$ ne renvoie pas de nouvelles arêtes (ce qui signifie que la face était déjà un triangle), l'algorithme passe simplement à la face suivante.
6. Si des arêtes ont été ajoutées, elles sont ajoutées à la liste $added\_edges$.
7. Au final, l'algorithme renvoie la liste $added\_edges$ qui contient toutes les arêtes qui ont été ajoutées au graphe pour le transformer en un graphe triangulé.

Cet algorithme suppose que le graphe d'entrée est planaire, c'est-à-dire qu'il peut être dessiné sur un plan sans que les arêtes ne se croisent. Il ne fonctionnera pas correctement si le graphe d'entrée n'est pas planaire.

## Preuve d'exactitude

$add\_edge\_to\_graph()$ **:**

Il est trivial de montrer que cette fonction est en **O(1)**. En effet, la fonction est seulement composée dans le pire des cas de deux ajouts d'éléments qui sont en **O(1)** donc la fonction est bien en **O(1)**.

$recur\_face()$ **:**

Pour prouver que l'algorithme se termine, nous devons montrer qu'il ne peut pas entrer dans une boucle infinie. L'algorithme termine si la face donnée est un triangle, ce qui est le cas de base de notre récursion. Si ce n'est pas un

triangle, l'algorithme ajoute de nouvelles arêtes, créant des faces plus petites, puis récursivement les traite. Étant donné qu'il n'y a qu'un nombre fini de sommets et que chaque appel récursif traite une face plus petite, nous avons une condition de terminaison garantie. En d'autres termes, à chaque itération, soit nous nous rapprochons du cas de base (un triangle), soit nous traitons une face qui ne peut plus être divisée. Ainsi, l'algorithme se termine.

$triangulate\_graph()$ **:**

La preuve d'exactitude de l'algorithme $triangulate\_graph()$ repose sur plusieurs affirmations que nous devons vérifier :

1. Le graphe d'entrée est planaire et connexe.
2. Chaque face du graphe est correctement identifiée par la fonction $find\_faces()$.
3. La fonction $recur\_faces$ est capable de transformer n'importe quelle face en une série de faces triangulaires de manière récursive.
4. Si une face est déjà un triangle, aucun changement n'est apporté.
5. L'algorithme se termine, et renvoie correctement toutes les arêtes ajoutées.

Pour la première affirmation, c'est une condition préalable de l'algorithme. Si le graphe n'est pas planaire ou n'est pas connexe, l'algorithme lève une exception et se termine.

Pour l'affirmation suivante, nous étudierons le comportement de $find\_face$ dans la section 4.3 "Détection de faces" et prouverons que notre algorithme détecte bien chaque face de notre graphe.

Pour la troisième affirmation, la fonction $recur\_faces$ ajoute des arêtes à chaque face non triangulaire jusqu'à ce qu'elle soit composée uniquement de triangles. Cela est garanti par la manière dont la fonction est écrite : elle ajoute une arête entre chaque paire de nœuds adjacents dans la face qui ne sont pas déjà connectés par une arête, puis répète le processus de manière récursive jusqu'à ce que tous les nœuds de la face soient connectés en triangles.

Pour la quatrième affirmation, c'est explicite dans le code de $recur\_faces$ : si la face est déjà un triangle, aucune action n'est effectuée.

Pour la dernière affirmation, l'algorithme se termine car il visite chaque face une fois et chaque appel à $recur\_faces$ réduit la taille de la face (en termes de nombre de nœuds non connectés par une arête). Par conséquent, il ne peut pas y avoir de boucle infinie. La liste $added\_edges$ est correctement mise à jour avec chaque nouvelle arête ajoutée, donc le résultat final est correct.

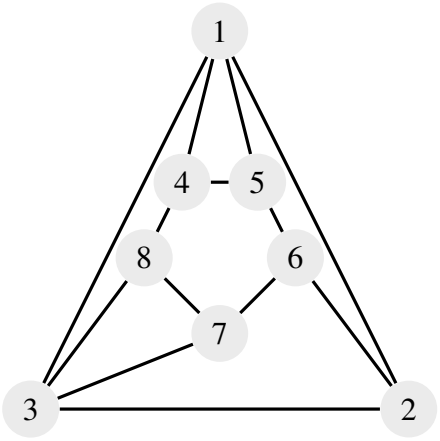



FIGURE 4.2 – Graphe de 8 sommets et 13 arêtes

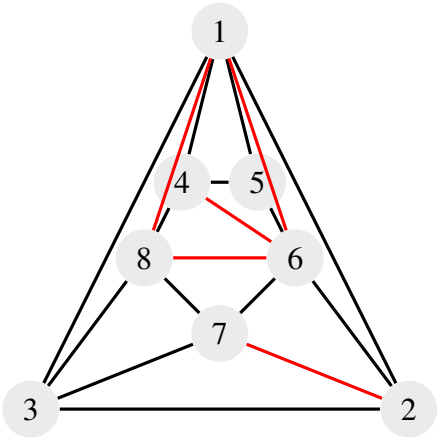



FIGURE 4.3 – Graphe de 8 sommets et 13 arêtes triangulé

## Exemple

### Exemple basique :

Prenons le graphe suivant représenté par la figure 4.2. Le résultat de la triangulation de ce graphe sera tel que la figure 4.3. On peut constater que ce graphe, composé de 8 sommets et 13 arêtes, possède 6 faces. Ce cas de figure est assez simple, en effet il ne suffit que d'un passage dans la récursion de l'algorithme par face pour trianguler celles-ci. On peut aussi constater que malgré la triangulation et l'ajout de nouvelles arêtes, le graphe reste planaire et est donc apte à être utilisé dans l'algorithme de Schnyder.

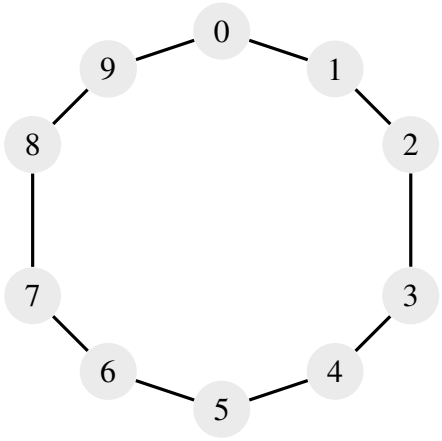



FIGURE 4.4 – Graphe de 10 sommets et 10 arêtes

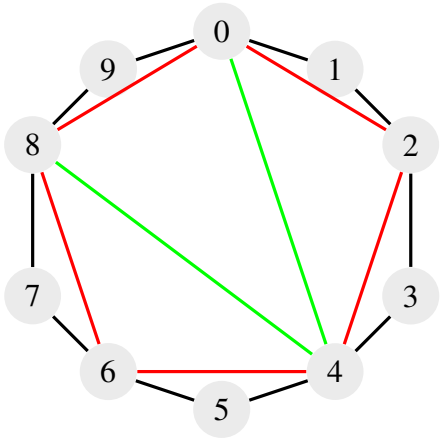



FIGURE 4.5 – Graphe de 10 sommets et 17 arêtes triangulé

**Exemple récursif :**

Prenons le graphe suivant représenté par la figure 4.4. Le résultat de la triangulation de ce graphe sera tel que la figure 4.5. On peut constater que ce graphe composé de 10 sommets et 10 arêtes, possède 1 face. Lors du premier passage dans l'algorithme, une nouvelle face (représentée par des arêtes rouges) est créée et celle-ci est composée de 5 sommets et 5 arêtes. Il faut donc passer une deuxième fois dans l'algorithme afin de rediviser cette nouvelle face en triangles élémentaires (représenté par des arêtes vertes). Une fois cela fait, on a bien un graphe triangulé composé maintenant de 8 faces (plus la face externe) et celui-ci conserve bien ces caractéristiques de planarité.

## 4.3 Détection de faces

*Dans cette section, nous aborderons la manière dont la détection de faces d'un graphe planaire est effectuée. Nous prouverons aussi que celle-ci se fait en* ***O(n)****.*

### Idée générale

La détection de faces a un rôle crucial dans la préparation du graphe pour l'algorithme de Schnyder. En effet, pour que le graphe soit correct, il faut que chaque face de notre graphe soit de forme triangulaire.
Cet algorithme va donc parcourir le graphe et renvoyer une liste dont le premier élément est la face externe du graphe et les éléments suivants les différentes faces du graphe.

Pour ce faire, l'algorithme examine un graphe et suit un processus en plusieurs étapes pour identifier ses faces. Il utilise une liste d'arcs dirigés, parcourant les arcs et examinant les connexions entre les nœuds pour déterminer si une face est trouvée. Si un arc est déjà dans le chemin actuel, une face est ajoutée à la liste. Sinon, l'arc est ajouté au chemin. Le processus continue jusqu'à ce que tous les arcs soient visités, et les faces trouvées sont cataloguées.

### Pseudo-code

*Voir l'algorithme 5* `find_face()`

### Fonctionnement

Tout d'abord, l'algorithme va répertorier l'intégralité des arcs dirigés du graphe et initialiser la première face avec le premier arc de la liste. Ensuite tant que cette liste n'est pas vide c'est-à-dire qu'il reste encore au moins un arc non visité. On prend le dernier élément du chemin actuel, on regarde le nœud adjacent suivant du dernier élément du chemin.

Si l'arc entre le dernier élément du chemin et le nouveau nœud existe déjà dans le chemin, une face est trouvée, on ajoute donc cette face à notre liste et l'on réinitialise le chemin avec le premier arc de notre liste d'arcs dirigés.

Si au contraire l'arc n'existe pas, on l'ajoute à notre chemin et on le supprime de la liste d'arcs dirigés.

**Algorithm 5** `find_face`

**Require:** $graph$ un graphe du module NetworkX
**Require:** $neighbor\_dict$ un dictionnaire sommet, voisin
**Ensure:** $face\_list$ la liste des faces du graphe
  `directed_edges` ← la liste des arcs diriges du graphes
  Initialisation de `face_list`
  Initialisation de `current_path`
  Ajout du premier edge dans `current_path`
  **while** `directed_edges`$\neq 0$ **do**
    Initialisation de $x, y$ comme le dernier élément de `current_path`
    Initialisation de $neighbors$ par `neighbor_dict[y]`
    Initialisation de $z$ comme le voisin suivant de $y$
    $edge$ devient $(y, z)$
    **if** $edge$ est dans `current_path` **then**
      Ajout de `current_path` dans `face_list`
      Ajout du premier edge dans `current_path`
    **else**
      Ajout de $edge$ dans `current_path`
      Remplacement de `edge_index` par l'index de $edge$ dans `directed_edges`
      Suppression de l'arc d'indice `edge_index` de `directed_edges`
    **end if**
  **end while**
  **if** `current_path` **then**
    Ajout de `current_path` dans `face_list`
  **end if**
  **return** `face_list`

Enfin si le chemin n'est pas vide, on insère le dernier chemin dans les faces du graphe.

## Preuves d'exactitude

On peut facilement prouver que chaque face de notre graphe sera trouvée. En effet, dans un graphe planaire, chaque arc fait partie de deux faces [1]. L'algorithme ici présent représente ces deux utilisations d'un même arc par une liste d'arcs dirigés.
Ensuite, nous pouvons prouver que le cycle trouvé est bien minimal et donc est bien une face du graphe. En effet, l'algorithme va suivre les différents nœuds du cycle dans l'ordre des aiguilles d'une montre et va donc tourner jusqu'à arriver au sommet initial.
Enfin, on peut ajouter le dernier chemin aux faces, car en suivant les propriétés des graphes planaires, tout arc aura bien été parcouru exactement deux fois et donc le chemin restant sera bien un cycle et donc une face.

Il est assez simple de montrer que cet algorithme est bien en **O(n)**. En effet, chaque instruction est en temps constant et seule la boucle while peut faire varier notre complexité. Lors de cette boucle, nous allons parcourir exactement 2 fois chaque arête de notre graphe. On est donc en **O(2m)** où $m$ est le nombre d'arêtes. Par la théorie des graphes connexes, on a que $m = \frac{n+f}{2}$ où $n$ est le nombre de sommets et $f$ le nombre de faces du graphe. On a donc une complexité en **O(n+xn)**, car une face est composée de $x$ sommets et donc une complexité finale en **O(n)**.

## Exemple

Prenons maintenant le graphe représenté à la figure 4.6 comme exemple pour illustrer notre algorithme. L'ensemble de ces arcs de ce graphe est [(0,1),(1,2) ,(2,3),(3,4),(4,0),(0,5),(1,5),(3,5),(1,0),(2,1),(3,2),(4,3),(0,4),(5,0),(5,1),(5,3)].

Lors du premier passage dans la boucle, la face extérieure va tout d'abord être détectée c'est-à-dire [(0,1),(1,2),(2,3),(3,4),(4,0)]. En effet, les prérequis de constructions de graphe demandant à ce que les arêtes soient notées dans l'ordre des aiguilles d'une montre, la face la plus externe sera détectée en première avant de détecter les différentes faces internes de notre graphe.

L'arête suivante dans la liste est (0,5) après que les différentes arêtes de la première face aient étés retirées. En suivant l'algorithme, la prochaine face [(0,5),(5,1),(1,0)]

1. Pour les arcs extérieurs, ils font partie à la fois d'une face interne et d'une face externe

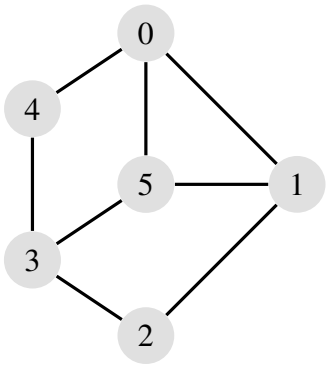



FIGURE 4.6 – Détection de face

est détectée.

Lors de l'étape suivante, la face [(1,5),(5,3),(3,2),(2,1)] est obtenue.

Enfin, la face [(3,5),(5,0),(0,4),(4,3)] est obtenue.

On peut constater que l'union de chaque ensemble d'arêtes représentant les différentes faces trouvées lors des passages dans la boucle correspond bien à notre ensemble d'arcs, nous avons donc bien détecté l'intégralité des faces de notre graphe.

# Chapitre 5

# Algorithme de Schnyder

*Dans ce chapitre, nous aborderons l'algorithme de Schnyder en lui-même. Nous commencerons par une explication de sur quoi celui-ci est basé, nous enchainerons ensuite sur l'implémentation étape par étape de l'algorithme. Nous verrons la façon d'effectuer une labélisation normale des faces de notre graphe, la création d'un réalisateur et enfin le calcul des coordonnées.*

## 5.1 Théorème

**Théorème 5.1.1.** ***(Embedding planaire)*** *Soit $\lambda_1, \lambda_2, \lambda_3$ trois droites non parallèles deux à deux dans le plan. Alors, chaque graphe planaire possède un embedding dans laquelle deux arêtes disjointes sont séparées par une ligne droite parallèle à $\lambda_1, \lambda_2, \lambda_3$.*

Un embedding planaire sur une grille $m$ par $n$ est un embedding dans lequel les sommets $v$ ont des coordonnées à valeurs entières dans l'intervalle $0 \leq v_1 \leq m$, $0 \leq v_2 \leq n$. De par cette définition, Schnyder a prouvé que :

**Théorème 5.1.2.** ***(Théorème de Schnyder (1990)[4])*** *Chaque graphe planaire avec $n \geq 3$ sommets admet un embedding planaire sur une grille de $(n-2) \times (n-2)$.*

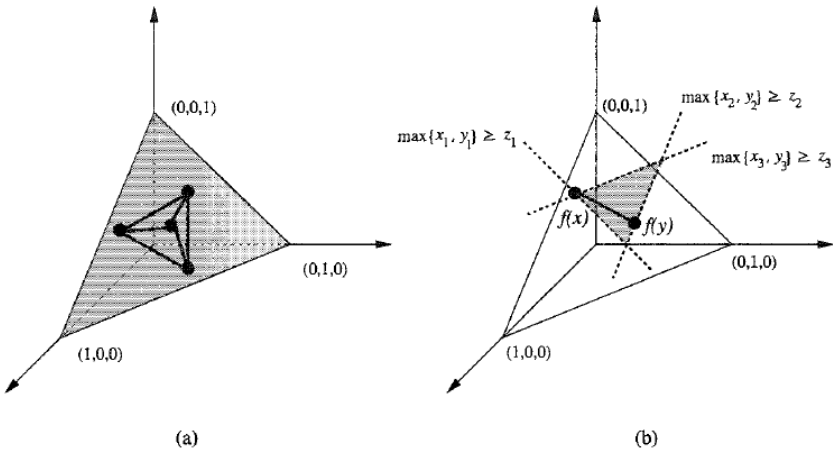



FIGURE 5.1 – Conditions de la représentation barycentrique[2]

## 5.2 Représentation barycentrique

Pour prouver la véracité de ce théorème, Schnyder utilise le concept de représentation barycentrique d'un graphe $G$.

**Définition 5.2.1.** *(**Représentation barycentrique**) La représentation barycentrique d'un graphe G est une fonction injective telle que : $f : v \in V(G) \to (v_1, v_2, v_3) \in \mathcal{R}^3$ satisfaisant les deux conditions suivantes :*

1. *$v_1, v_2, v_3 = 1$ pour tout $v$.*
2. *Pour chaque arête $(x, y)$ et chaque sommet $z \notin \{x, y\}$,il existe un index $k \in \{1, 2, 3\}$ tel que $x_k < z_k$ et $y_k < z_k$.*

Tel qu'illustré à la figure 5.1.

Cette définition implique la propriété suivante :

**Propriété 5.2.1.** *La représentation barycentrique $f$ d'un graphe $G$ est un dessin planaire linéaire dans le plan compris entre les 3 points (1,0,0), (0,1,0) et (0,0,1).*

*Démonstration.* (retraduite de [2]) Soit $(x, y)$ une arête de $G$ et $z$ un sommet tel que $z \notin \{x, y\}$. avec la condition (2) $x_k < z_k$ et $y_k < z_k$.
Par conséquent, le point $f(z)$ ne se trouve pas sur le segment de droite $f(x)f(y)$ entre les points $f(x)$ et $f(y)$ ; sinon, $f(z) = cf(x) + (1 - c)f(y)$ pour un certain nombre $c$, $0 \leq c \leq 1$, mais alors

$$z_k = cx_k + (1 - c)y_k < cz_k + (1 - c)z_k = z_k$$

est une contradiction.
Soit $(x, y)$ et $(u, v)$ deux arêtes quelconques de $G$ telles que les quatre extrémités $x, y, u$ et $v$ soient distinctes par paire. Par la condition 2, il existe des indices $i, j, k, l \in \{1, 2, 3\}$ tels que :

$$u_i, v_i < x_i \quad (a)$$

$$u_j, v_j < y_j \quad (b)$$

$$x_k, y_k < u_k \quad (c)$$

$$x_l, y_l < v_l \quad (d)$$

Alors $i \neq k, l$ par (a),(c) et (d) et $j \neq k, l$ par (b),(c) et (d). Donc $i = j$ et $k = l$. Autrement, les quatre indices $i, j, k, l$ seraient distincts les uns des autres bien que $i, j, k, l \in \{1, 2, 3\}$. On peut donc supposer que $i = j = 1$. Alors par (a) et (b), les deux segments de droite $f(x)f(y)$ et $f(u)f(v)$ sont séparés par une droite parallèle au segment de droite (0,1,0)(0,0,1). Les deux segments de droite $f(x)f(y)$ et $f(u)f(v)$ ne se coupent donc pas.

□

Cette propriété implique que seuls les graphes planaires peuvent avoir une représentation barycentrique.

En utilisant ce type de représentation, on peut obtenir un dessin linéaire de graphe planaire sur une grille de $(2n - 5) \times (2n - 5)$. Plus particulièrement en utilisant la représentation barycentrique faible, on peut réduire la taille de la grille à $(n - 2) \times (n - 2)$.

**Définition 5.2.2.** ***(Représentation barycentrique faible)*** *La représentation barycentrique d'un graphe $G$ est une fonction injective telle que : $f : v \in V(G) \rightarrow (v_1, v_2, v_3) \in \mathcal{R}^3$ satisfaisant les deux conditions suivantes :*

1. *$v_1, v_2, v_3 = 1$ pour tout $v$.*
2. *Pour chaque arête $(x, y)$ et chaque sommet $z \notin \{x, y\}$, il existe un index $k \in \{1, 2, 3\}$ tel que $(x_k, x_{k+1}) <_{lex}$ [1]$(z_k, z_{k+1})$ et $(y_k, y_{k+1}) <_{lex} (z_k, x_{k+1})$, où les indices sont calculés mod 3.*

1. on écrit $(a, b) <_{lex} (c, d)$ si soit $a < c$ ou $a = c$ et $b < d$

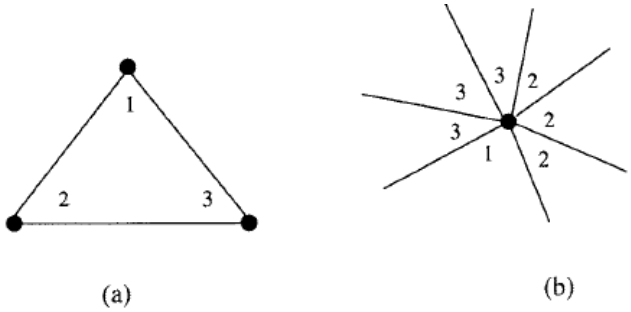



FIGURE 5.2 – Conditions de labélisation[2]

Une propriété similaire à la propriété 5.2.1 dit que :

**Propriété 5.2.2.** *Une représentation barycentrique faible $f$ d'un graphe $G$ est un dessin planaire et linéaire de $G$ dans le plan délimité par 3 points (1,0,0), (0,1,0) et (0,0,1).*

La preuve de cette propriété étant assez similaire à celle de la propriété 5.2.1 celle-ci ne sera pas faite dans ce mémoire [2].

## 5.3 Labélisation

*Dans cette section, nous aborderons la partie de l'algorithme permettant la labélisation du graphe triangulé. Nous prouverons aussi par la même occasion que cette partie de l'algorithme est bien en O(n).*

**Définition 5.3.1.** *(**Labélisation**) La labélisation de Schnyder de $G$ consiste à apposer un label 1,2,3 à tous les angles de $G$ en satisfaisant les conditions suivantes :*

*(a) Chaque face interne triangulée de G à un angle labelé 1, labelé 2 et labelé 3. Les trois sommets correspondants du triangle apparaissent dans l'ordre inverse des aiguilles d'une montre.*

*(b) les labels des angles de chaque sommet intérieur de G forment dans le sens inverse des aiguilles d'une montre un intervalle non vide de 1 suivi d'un intervalle non vide de 2 suivi d'un intervalle non vide de 3.*

Ces deux conditions sont illustrées à la figure 5.2.

2. Preuve présente dans le livre Planar Graph Drawing[2]

Une labélisation de graphe planaire $G$ est illustrée à la figure 3.4 du côté gauche. On peut constater que pour chaque sommet externe, ceux-ci ont des labels uniques 1,2,3 et qu'ils apparaissent dans le sens antihoraire. Ceci sera démontré par la suite.

Prouvons tout d'abord le théorème suivant :

**Théorème 5.3.1.** *Chaque graphe planaire triangulé possède une labélisation de Schnyder.*

*Démonstration.* Soit $G$ un graphe planaire triangulé et soit $a$ un sommet externe de $G$. On va prouver par induction sur le nombre $n$ de sommets de $G$ que $G$ a une labélisation de Schnyder dans laquelle tous les angles en $a$ ont le label 1.

Le cas où $n = 3$ est trivial et par conséquent, laissons $n \geq 4$ et supposons que notre affirmation est vraie pour tous les graphes planaires triangulés ayant moins de $n$ sommets.

**Théorème 5.3.2.** *Soit G un graphe triangulaire à $n \geq 4$ sommets. Si $a, b$ et $c$ désignent les sommets extérieurs de G, alors il existe un voisin $x \neq b, c$ de a tel que l'arête $\{a, x\}$ est réductible.*

*Ainsi, à chaque graphe triangulaire correspond une séquence de "contractions autorisées" transformant ce graphe en triangle. Et, inversement, chaque graphe triangulaire peut être obtenu à partir d'un triangle par une séquence d'expansions.*

En suivant le théorème 5.3.2 [3], ça implique qu'il existe un sommet intérieur $x$ adjacent à $a$ tel que l'arête $(a, x)$ est réductible et donc le graphe $G \setminus (a, x)$ obtenu à partir de $G$ en contractant l'arête $(a, x)$ a une labélisation de Schnyder dans laquelle tous les angles à $a$ ont le label 1. Cette labélisation peut être étendue à une labélisation de Schnyder de $G$ dans lequel tous les angles en $a$ ont le label 1. Cette preuve est illustrée par la figure 5.3 .

□

3. Ce théorème ne sera pas prouvé dans ce travail.

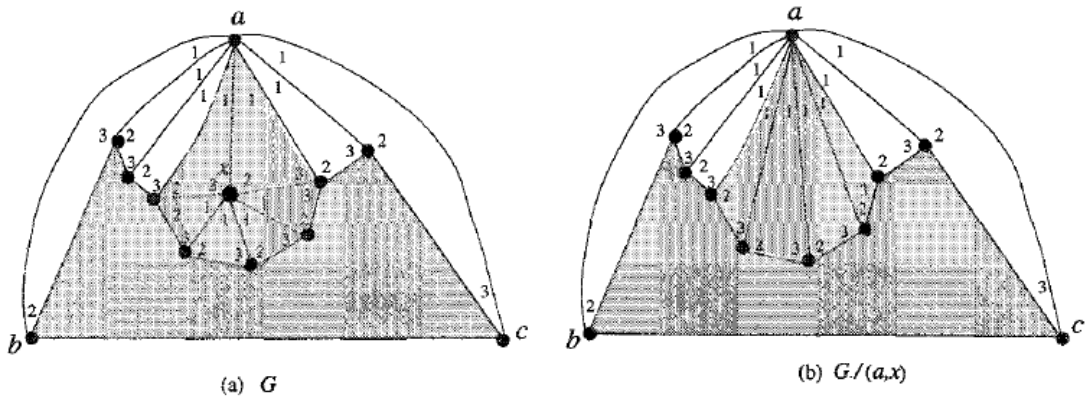



FIGURE 5.3 – Conditions de labélisation[2]

## Idée générale de l'algorithme

En se basant sur le théorème 5.3.1, il est possible de construire une labélisation de Schnyder en **O(n)**.

L'algorithme de Schnyder commence par trier les sommets de la face externe d'un graphe planaire triangulé. Il crée ensuite des structures de données pour suivre les relations entre les sommets.

Les sommets reliés avec le premier sommet trié sont identifiés et mis de côté pour être traités ultérieurement. Le graphe est ensuite réduit en retirant certains sommets et en ajoutant des arcs, en tenant compte des relations entre les sommets.

Ce processus continue jusqu'à ce qu'il ne reste que trois sommets dans le graphe. À ce stade, ces sommets sont étiquetés avec des numéros uniques.

Les sommets qui ont été retirés précédemment sont ensuite réintroduits dans le graphe. Ils sont traités et étiquetés en fonction de leurs relations avec les autres sommets et selon qu'ils remplissent ou non certaines conditions.

Finalement, l'algorithme renvoie un ensemble d'étiquettes pour les sommets du graphe et une liste des trois sommets initiaux.

## Pseudo-code

*Voir l'algorithme 6* `schnyder_labeling()`

**Algorithm 6** schnyder_labeling

**Require:** $G$ une graphe du module NetworkX
**Require:** $dico$ un dictionnaire sommet, voisin
**Require:** $external\_face$ la face externe du graphe
**Ensure:** labels un dico qui pour chaque sommet a pour donnée les labels des angles si rapportant
Initialisation de $dico\_adj$ à 0 pour chaque sommet
Initialisation de $dico\_v0$ au dictionnaire du premier sommet de $external\_face$
Assignation du nombre de voisins communs entre un sommet de $dico\_v0$ et de $dico\_v0$ à $dico\_adj$
Initialisation de deux liste $to\_strip$ et $strip$
Ajout de(s) sommet(s) ayant 2 sommets adjacents en commun et n'appartenant pas à v à $to\_strip$
**while** Le nombre de noeuds est superieur a 3 **do**
  **if** $to\_strip$ vide **then**
    Sortie de la boucle
  **end if**
  Extraction de $u$ le dernier élément de $to\_strip$ du graphe
  Ajout des sommets adjacents à $u$ mais pas à $v0$ à $dico\_v0$
  Ajout de $u$ à la liste $strip$
  Mise à jour de $dico$ sans u
  Suppression de $u$ de $dico\_v0$
  Ajout d'un arc pour les sommets adjacents à $u$ mais pas à $v0$
  Ajout à $dico\_v0$ les sommets qui étaient adjacents à $u$ mais pas à $v0$
  Mise à jour de $to\_strip$ avec le graphe sans $u$
**end while**
Tri des 3 derniers sommets
Création des labels pour les sommets racines
**while** $strip$ non nul **do**
  Extraction de $u$ dernier élément de $strip$
  Suppression des arêtes ajoutées
  **if** aucune arête n'a été ajouté **then**
    Ajout du label de $u$
    Mise à jour du label des sommets adjacents de $u$
  **end if**
  Ajout des arêtes entre $u$ et ses sommets adjacents
**end while**
**return** Liste des labels, face externe

## Fonctionnement

L'algorithme de labélisation de Schnyder fonctionne de la manière suivante :

1. On commence par trier les sommets de la face externe du graphe (représenté par $external\_face$) en ordre croissant, enregistré dans la variable $v$.
2. On initialise deux dictionnaires. Le premier, $dico\_adj$, est utilisé pour compter le nombre de sommets communs entre le premier sommet trié $v0$ et tous les autres sommets du graphe. Le deuxième, $dico\_v0$, contient la liste des sommets adjacents à $v0$.
3. On identifie ensuite les sommets ayant exactement deux sommets adjacents en commun avec $v0$ et qui ne sont pas dans $v$. Ces sommets sont ajoutés à la liste $to\_strip$.
4. Le graphe est réduit jusqu'à ce qu'il ne reste que trois sommets. Pour cela, un sommet $u$ est choisi à partir de $to\_strip$, et ses voisins sont divisés en deux groupes : ceux qui sont aussi voisins de $v0$ ($new\_dico$) et ceux qui ne le sont pas ($old\_dico$).
5. Le sommet u est alors retiré du graphe et des arcs sont ajoutés entre $v0$ et tous les sommets de $old\_dico$.
6. À ce stade, chaque triplet de sommets est étiqueté avec les numéros 0, 1 et 2.
7. L'algorithme s'occupe ensuite des sommets qui ont été enlevés auparavant. Chaque sommet $u$ est réintroduit dans le graphe, et ses voisins sont reclassés en deux groupes $new\_dico$ et $old\_dico$ comme précédemment.
8. Si $old\_dico$ est vide, cela signifie que tous les voisins de u sont également voisins de $v0$. Les étiquettes sont alors attribuées en conséquence.
9. Après avoir traité tous les sommets enlevés, l'algorithme renvoie le dictionnaire d'étiquettes et la liste des trois sommets initiaux.

## Preuve d'exactitude

Prouvons maintenant que cet algorithme fonctionne et renvoie bien la labélisation du graphe suivant la labélisation de Schnyder.

L'algorithme agit en deux temps. En premier lieu, il va simplifier au fur et à mesure le graphe, en éliminant les sommets, tout en conservant l'information nécessaire pour construire un étiquetage de Schnyder valide pour le graphe original. Ensuite, dans un second temps, l'algorithme va ajouter les sommets précédemment supprimés, un à un, tout en mettant à jour la labélisation du graphe.

Montrons tout d'abord que la suppression de sommets s'effectue correctement tout en gardant les informations nécessaires et renvoie bien un graphe composé de trois sommets formant un triangle élémentaire qui se trouve être la face externe du graphe.

Au début de chaque itération de la boucle while, le graphe $G$ est planaire et triangulaire et le sommet $v0$ est toujours l'un des sommets de la face externe. De plus, pour chaque sommet $u$ dans $to\_strip$, $u$ a exactement deux voisins dans $dico\_v0$.

Cette propriété est maintenue tout au long de l'algorithme, car chaque fois qu'un sommet $u$ est supprimé de $G$, tous les voisins de $u$ qui n'étaient pas dans $dico\_v0$ sont reliés à $v0$, ce qui maintient la planarité et la triangularité de $G$.

Montrons maintenant que cette boucle s'arrête bien. L'algorithme se termine, car à chaque itération de la boucle while, au moins un sommet est supprimé de $G$. Par conséquent, après un nombre fini d'itérations, $G$ est réduit à un graphe avec seulement trois sommets ($v0$, $v1$, $v2$).

Il faut maintenant montrer que le graphe est bien reconstruit avec tous les sommets extraits et que la labélisation s'effectue correctement.

Prouvons par induction que la labélisation s'effectue correctement.

À l'initialisation, on a $G$ possédant 3 sommets externes et 0 sommet internes. Il est donc trivial de labéliser le graphe en donnant comme label le numéro du sommet à l'angle s'y rapportant.

Supposons que $k$ sommets ont déjà été replacés sur le graphe où $0 < k < nbre\_vertex - 3$ et la labélisation effectuée correctement, montrons que replacer le $k + 1$ conserve la labélisation correcte du graphe. Lors de l'itération du sommet $k + 1$, on retire tout d'abord de la liste le label, des sommets adjacents au sommet $k + 1$. On met ensuite à jour les différents labels des faces ayant le sommet $k + 1$ compris dedans. Cela permet de modifier les labels uniquement des faces comprenant le sommet $k + 1$ sans modifier le reste du graphe et donc le graphe est bien labelé à la fin des itérations.

**Preuve de complexité de l'algorithme**

Montrons que l'algorithme est bien en **O(n)**.

Dans le cas de l'algorithme $schnyder\_labeling$, les principales opérations sont :

1. Parcourir chaque nœud du graphe $G$ : **O(n)** où $n$ est le nombre de nœuds du graphe.
2. Retirer un nœud du graphe et reconstruire le dictionnaire d'adjacence : **O(1)** pour chaque nœud, donc **O(n)** au total.
3. Les opérations de tri et d'ajout/suppression dans les listes ($strip$, $dico\_v0$, etc.) peuvent être considérées comme **O(n)** dans le pire des cas (où $n$ est le nombre de nœuds du graphe).
4. L'ajout et la suppression d'arcs dans le graphe sont en $O(1)$ pour chaque opération dans NetworkX, mais ces opérations étant effectuées dans une boucle, leur complexité totale peut être **O(n)**.

Ainsi, la complexité temporelle globale de l'algorithme $schnyder\_labeling$ est probablement de l'ordre d'**O(n)** dans le pire des cas, où $n$ est le nombre de nœuds du graphe.

## Exemple

Illustrons ce procédé complexe par un exemple.

Prenons le graphe suivant (voir figure 5.4a).

Ce graphe est composé de 6 sommets et 12 arêtes et les différentes faces sont $[[(0,1),(1,2),(2,0)],[(0,4),(4,3),(3,0)],[(0,5),(5,1),(1,0)],[(0,2),(2,4),(4,0)],$ $[(0,3),(3,5),(5,0)],[(1,5),(5,2),(2,1)],[(2,3),(3,4),(4,2)],[(2,5),(5,3),(3,2)]]$.

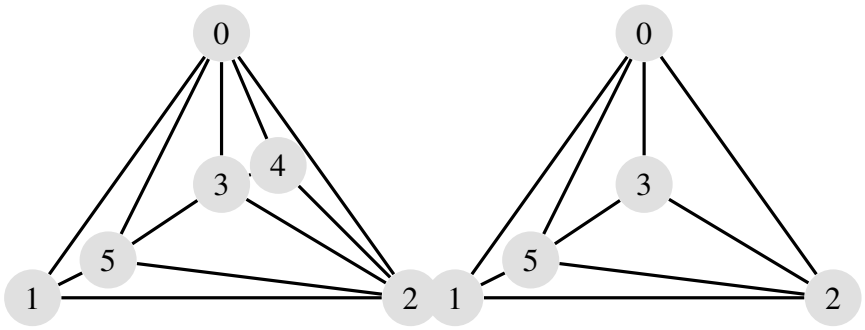
Soit $[(0,1),(1,2),(2,0)]$ la face externe de notre graphe.

Lors de la première partie de notre algorithme, nous allons supprimer un à un les sommets internes en les stockant dans la liste $strip$.

Le premier sommet à extraire est le sommet $v4$. En effet comme le montre la figure 5.4a. C'est le seul sommet ayant seulement deux sommets adjacents à $v0$. Une fois la suppression effectuée on obtient le graphe de la figure 5.4b et la liste $strip$ devient $[(4,[3,2,0],[])]$.

On extrait ensuite le sommet $v3$. Une fois la suppression effectuée on obtient le graphe de la figure 5.5a et la liste $strip$ devient $[(4,[3,2,0],[]),(3,[2,0,5],[])]$.

Enfin, on extrait le sommet $v5$. Une fois la suppression effectuée on obtient le graphe de la figure 5.5b et la liste $strip$ devient $[(4,[3,2,0],[]),(3,[2,0,5],[]),(5,[2,0,1],[])]$.

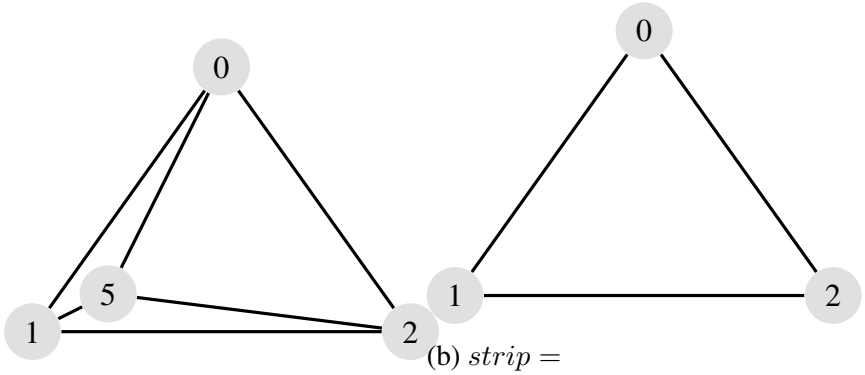



(a) Graphe triangulé de 6 nœuds et 12 arêtes

(b) $strip =$ $[(4, [3, 2, 0], [])]$

FIGURE 5.4 – Exemple labélisation partie 1

Le graphe est devenu élémentaire, on arrête la suppression de sommets.




(a) $strip =$ $[(4, [3, 2, 0], []), (3, [2, 0, 5], [])]$

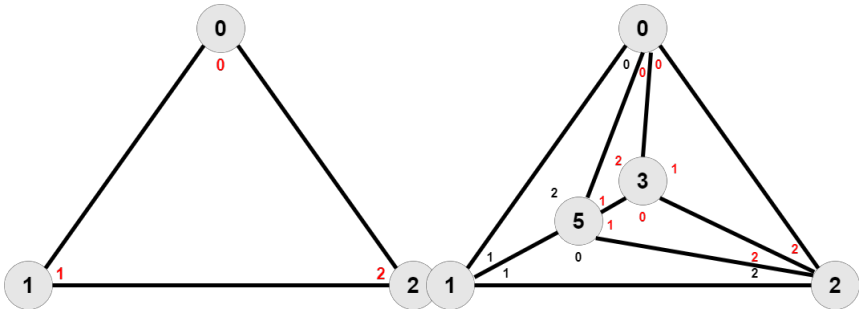
(b) $strip =$ $[(4, [3, 2, 0], []), (3, [2, 0, 5], []),$ $(5, [2, 0, 1], [])]$

FIGURE 5.5 – Exemple labélisation partie 2

À partir du graphe élémentaire, on initialise les labels des 3 angles tels que :$\{0 : \{(1, 2) : 0\}, 1 : \{(0, 2) : 1\}, 2 : \{(0, 1) : 2\}\}$, comme illustré à la figure 5.6a.

On peut commencer à dépiler $strip$, on ajoute donc le sommet $v5$ et l'on met à jour les faces qui possèdent le sommet $v5$. On obtient les labels :$\{0 : \{(1, 5) : 0, (2, 5) : 0\}, 1 : \{(0, 5) : 1, (2, 5) : 1\}, 2 : \{(0, 5) : 2, (1, 5) : 2\}, 5 : \{(0, 1) : 2, (0, 2) : 1, (1, 2) : 0\}\}$, comme représenté sur la figure 5.6b.

On réitère l'opération avec le sommet $v3$. On obtient les labels :$\{0 : \{(1, 5) : 0, (2, 3) : 0, (3, 5) : 0\}, 1 : \{(0, 5) : 1, (2, 5) : 1\}, 2 : \{(1, 5) : 2, (0, 3) : 2, (3, 5) :$



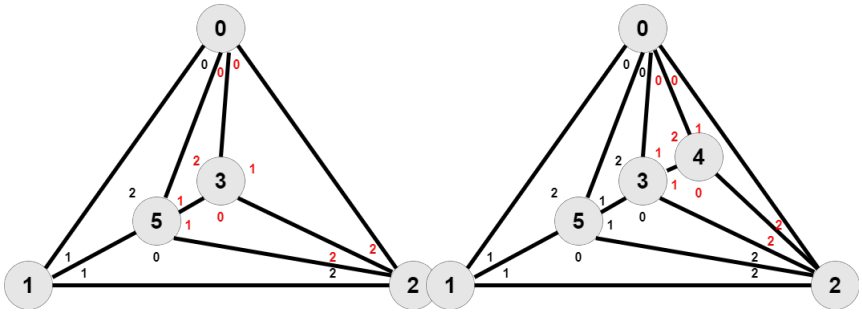

(a) Initialisation sommet (b) Ajout v5

FIGURE 5.6 – Exemple labélisation partie 3

$2\}, 5 : \{(0,1) : 2, (1,2) : 0, (0,3) : 1, (2,3) : 1\}, 3 : \{(0,2) : 1, (0,5) : 2, (2,5) : 0\}\}$ représenté sur la figure 5.7a.

Enfin, on ajoute le dernier sommet $v4$. On obtient les labels : $\{0 : \{(1,5) : 0, (3,5) : 0, (2,4) : 0, (3,4) : 0\}, 1 : \{(0,5) : 1, (2,5) : 1\}, 2 : \{(1,5) : 2, (3,5) : 2, (0,4) : 2, (3,4) : 2\}, 5 : \{(0,1) : 2, (1,2) : 0, (0,3) : 1, (2,3) : 1\}, 3 : \{(0,5) : 2, (2,5) : 0, (0,4) : 1, (2,4) : 1\}, 4 : \{(0,2) : 1, (0,3) : 2, (2,3) : 0\}\}$ représenté sur la figure 5.7b.

À la fin du programme, on a bien que le graphe est labélisé comme illustré à la




(a) Ajout v3 (b) Ajout v4

FIGURE 5.7 – Exemple labélisation partie 4

figure 5.7b en respectant les différentes règles énoncées précédemment et est prêt pour la création du réalisateur.

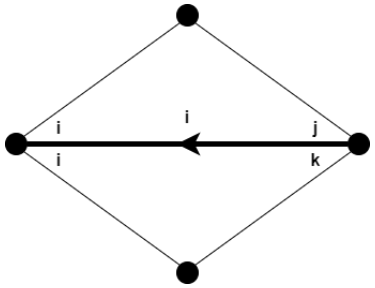



FIGURE 5.8 – Orientation des arêtes

## 5.4 Réalisateur

*L'étape suivante de notre algorithme est d'effectuer la réalisation de notre graphe, c'est-à-dire donner une orientation à chaque arête ainsi qu'un label.*

Chaque arête interne de $G$ appartient à deux faces triangulaires. Les conditions (a) et (b) de la définition de labélisation nous montrent que chaque arête interne a deux labels distincts $j$ et $k$ d'un côté et le troisième label $i$ répété deux fois de l'autre côté. On va appeler ce label distinct $i$ le label de l'arête et orienter cette arête de l'extrémité avec des labels distincts vers l'extrémité possédant le label $i$. Comme illustré à la figure 5.8.

**Définition 5.4.1.** *(**Réalisation**) Une réalisation d'un graphe triangulaire $G$ est une partition des arêtes intérieures de $G$ en trois ensembles $T_1$, , $T_2$, , $T_3$ d'arêtes dirigées de telle sorte que pour chaque sommet intérieur $v$, celui respecte les conditions suivantes :*

- *$v$ a une arête sortante dans chaque $T_1$, , $T_2$, , $T_3$.*
- *L'ordre antihoraire des arêtes passant par $v$ est : On sort de $T_1$, on entre en $T_3$, on sort de $T_2$, on entre en $T_1$, on sort de $T_3$, on entre en $T_2$. tel qu'illustré à la figure 5.9.*

**Théorème 5.4.1.** *Chaque graphe planaire triangulé à un réalisateur.*
Un exemple de réalisateur est illustré à la figure 5.12 et représente le réalisateur du graphe labellé à la figure 5.7b.

### Idée générale de l'algorithme

L'idée générale du fonctionnement de l'algorithme est assez simple.

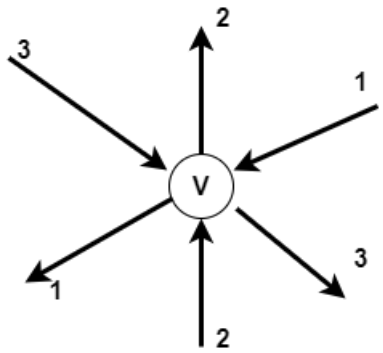



FIGURE 5.9 – Réalisateur de $v$

Tout d'abord, on va créer un dictionnaire pour représenter notre réalisateur ainsi qu'un dictionnaire par nœud contenant un triplet qui représentera notre arbre.

L'algorithme parcourt ensuite chaque sommet du graphe original. Pour chaque sommet, il crée une liste de listes pour contenir les sommets pour chaque label (0, 1, 2).

Pour chaque sommet, l'algorithme ajoute les sommets à la liste appropriée en fonction de leur label.

Pour chaque label, l'algorithme trie les sommets avec le même label. Il ajoute ensuite des arcs au graphe dirigé pour les sommets avec le même label, en veillant à ce que les arcs pointent dans la direction correcte.

## Pseudo-code

*Voir l'algorithme 7* `realizer()`

## Fonctionnement

Expliquons le fonctionnement de l'algorithme étape par étape :

1. Extraction des labels et de la face externe : L'algorithme commence par extraire les labels et la face externe de l'entrée $schnyder\_labeling$. Les labels sont le résultat de la labélisation de Schnyder, et la face externe est un triplet de sommets qui forment la face externe du graphe planaire.
2. Tri des sommets de la face externe : Les sommets de la face externe sont triés.

**Algorithm 7** realizer

**Require:** $G$ une graphe du module NetworkX
**Require:** $schnyder\_labeling$ Un tuple contenant les labels et la face externe
**Ensure:** Un tableau contenant les coordonnées des sommets
  Initialisation de $labels$ et $external\_face$ à $schnyder\_labeling$
  Tri de la face externe
  Initialisation d'un graphe dirigé
  Création d'un dictionnaire pour chaque sommet, associant à chaque sommet un triplet de noeuds étiquetés du numéro de sommet
  **for** node dans G **do**
    Initialisation de $angles$ comme triplet de liste
    **for** $angle$, $label$ dans labels[node] **do**
      Ajout de $angle$ dans $angles[label]$
    **end for**
    **for** Chaque angle **do**
      Tri de $angle\_label$
      Ajout des arcs au graphe dirigé ayant le même angle
    **end for**
  **end for**
    **return** $calculate\_coords()$

3. Création d'un graphe dirigé : Un nouveau graphe dirigé est créé pour représenter les connexions entre les sommets du graphe original.
4. Création d'un dictionnaire pour chaque sommet : Un dictionnaire est créé pour chaque sommet du graphe. Chaque entrée du dictionnaire est un triplet de nœuds, chacun étant un TreeNode avec le numéro du sommet comme label.
5. Parcours des sommets du graphe : Chaque sommet du graphe est parcouru en suivant :

   *(a)* Initialisation de listes pour les angles : Pour chaque sommet, trois listes sont initialisées pour représenter les trois angles possibles formés avec les sommets adjacents.

   *(b)* Ajout des sommets aux listes en fonction des labels : Les sommets adjacents sont ajoutés aux listes correspondantes en fonction de leur label dans l'étiquetage de Schnyder.

   *(c)* Parcours des angles pour chaque sommet : Chaque liste d'angles est parcourue, et les sommets adjacents avec le même angle sont ajoutés comme arcs au graphe dirigé. Les sommets adjacents sont également ajoutés comme enfants du nœud correspondant dans l'arbre représenté par $tree\_nodes$.

- Calcul des coordonnées : Enfin, une fonction $calculate\_coords$ est appelée avec le graphe dirigé, le dictionnaire d'arbre de nœuds et la face externe triée pour calculer les coordonnées des sommets.

Cette fonction renvoie les coordonnées calculées, qui peuvent être utilisées pour dessiner le graphe avec les sommets placés en fonction de leur étiquetage de Schnyder.

## Preuves d'exactitude

La preuve de l'exactitude de cet algorithme, qui calcule un réalisateur à partir d'une labélisation de Schnyder, repose sur plusieurs propriétés de la labélisation de Schnyder elle-même.

Montrons d'abord que l'algorithme suit la labélisation de Schnyder : le fait que l'algorithme utilise les labels de la labélisation de Schnyder pour déterminer les relations entre les sommets (en particulier, la construction des arbres pour chaque couleur) garantit que l'orientation et la coloration des arêtes sont respectées.
Montrons ensuite que l'algorithme se termine : l'algorithme parcourt tous les sommets du graphe une fois, il termine donc en un temps fini.

### Preuve de complexité

Prouvons que notre algorithme est bien en **O(n)**

- Création d'une copie du graphe et création du dictionnaire des sommets : ces opérations ont une complexité d'**O(n)**, où n est le nombre de sommets dans le graphe.
- Parcourir chaque sommet dans le graphe (la boucle for principale) : ceci a également une complexité d'**O(n)**, puisque l'on visite chaque sommet une seule fois.
- À l'intérieur de cette boucle, nous avons deux opérations principales :
  1. Pour chaque sommet, le code crée trois listes d'angles ($angle\_label$). Le remplissage de ces listes se fait en parcourant toutes les arêtes adjacentes au sommet actuel. Donc, cette opération a une complexité d'**O(m)**, où m est le nombre total d'arêtes dans le graphe.
  2. Après cela, pour chaque angle, nous avons une autre boucle qui parcourt les éléments de $angle\_label$ (qui contient des arêtes). Cette boucle a une complexité **d'O(m)**, car dans le pire des cas, nous pourrions parcourir toutes les arêtes du graphe.
- Enfin, l'appel à $calculate\_coords()$ : La complexité de cette fonction dépend de son implémentation, mais supposons qu'elle soit également linéaire par rapport au nombre de sommets et d'arêtes, c'est-à-dire **O(n + m)**.

En combinant toutes ces opérations, la complexité de cet algorithme est **O(n + 4m) + O(n + m)** pour $calculate\_coords()$, ce qui est finalement O(n + m), car les termes constants et les termes linéaires inférieurs sont en général omis dans la notation de la complexité en temps. Ainsi, dans un graphe planaire où m est en O(n), la complexité est essentiellement en **O(n)**.

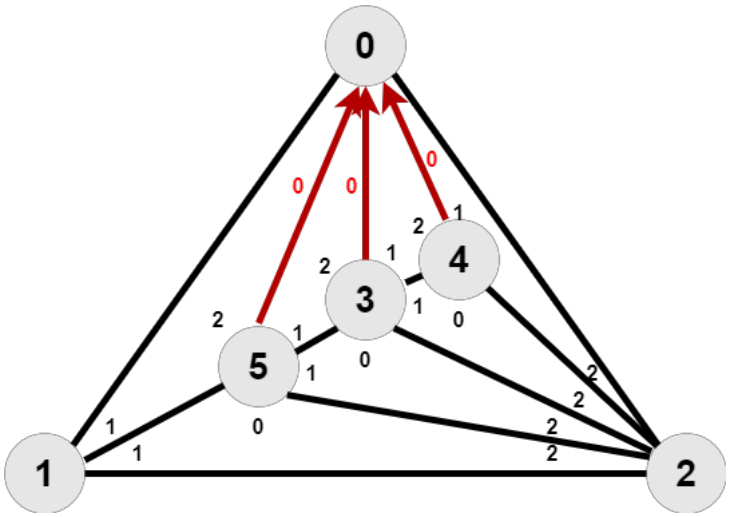
## Exemple

*Illustrons cette partie de l'algorithme par un exemple.*

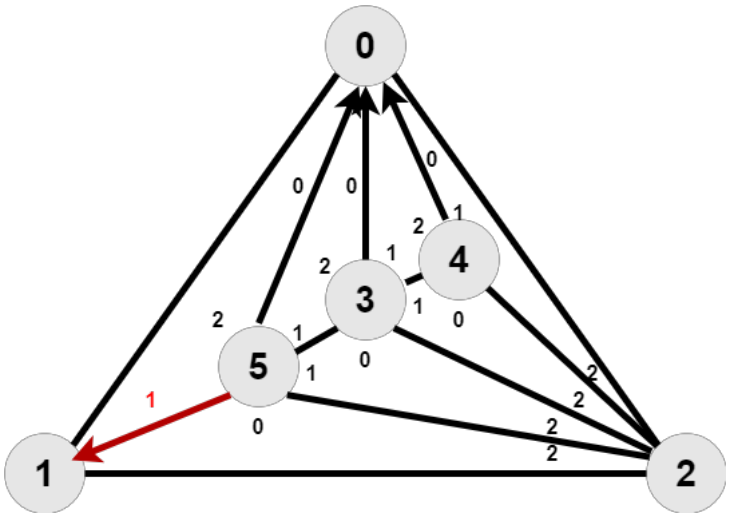
Reprenons la labélisation obtenue dans l'exemple de la labélisation de Schnyder représentée à la figure 5.7b.

On regarde tout d'abord le sommet $v0$, on observe que suivant la labélisation, on a la création d'un arc dirigé $(5, 0)$ car le label des angles est en commun, un arc dirigé $(3, 0)$ ainsi que $(4, 0)$ pour les mêmes raisons ce qui nous donne la figure 5.10a.

On s'attarde ensuite sur le sommet $v1$, pour celui-ci on crée un arc dirigé $(5, 1)$ étant le seul arc allant vers $v1$ qui a un label d'angle commun. Cela se représente sur la figure 5.10b.




(a) Réalisation v0

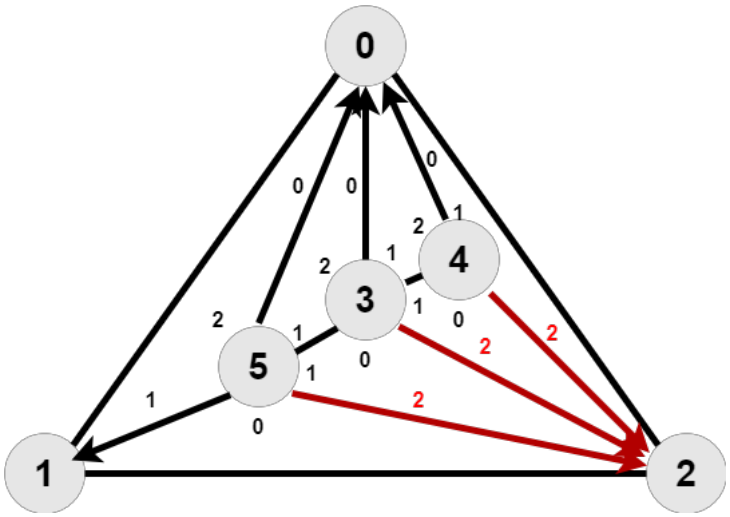



(b) Réalisation v1

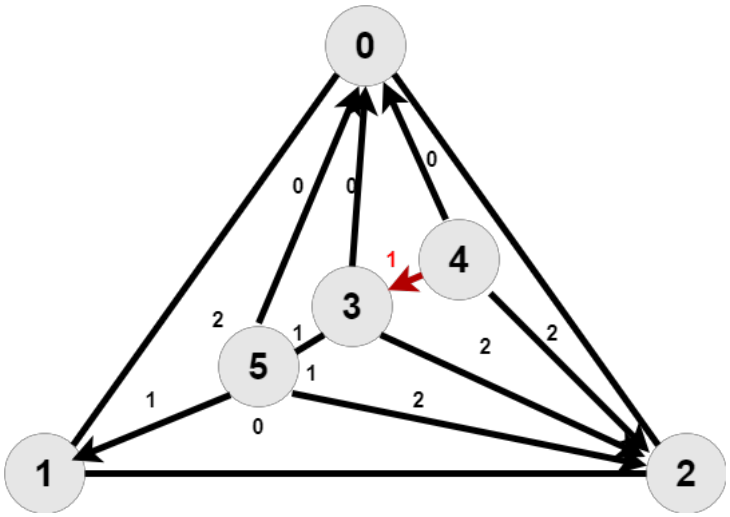
FIGURE 5.10 – Exemple réalisation partie 1

On passe au sommet $v2$, on observe la création d'un arc dirigé $(5, 2)$, d'un arc dirigé $(3, 2)$ ainsi que d'un arc dirigé $(4, 2)$ car ceux-ci ont des labels d'angles valant 2 en commun. Cette création d'arcs se représente sur la figure 5.11a.

Ensuite, on étudie le cas de $v3$, ici seul un arc dirigé $(4, 3)$ est créé. Seuls les 2 angles des faces ayant l'arête $(3, 4)$ en commun ont un label égal valant 1. On peut l'observer sur la figure 5.11b.




(a) Réalisation v2




(b) Réalisation v3

FIGURE 5.11 – Exemple réalisation partie 1

L'avant-dernier sommet à étudier est $v4$, celui-ci ne possède aucun nouvel arc entrant étant donné qu'il n'y a aucun label en commun au niveau des angles entourant $v4$.

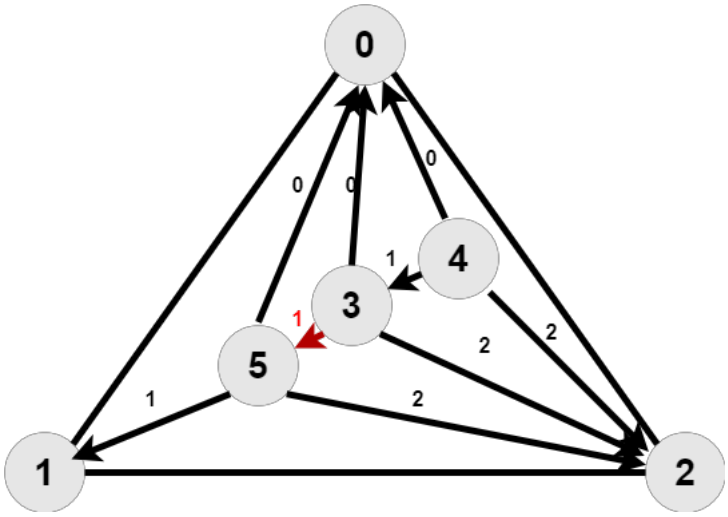



FIGURE 5.12 – Réalisation finale

Enfin, on observe le sommet $v5$, on peut constater qu'il y a la création d'un arc dirigé $(3, 5)$ car les labels des deux angles des faces ayant l'arête $(3, 5)$ en commun sont identiques et valent 1. Le réalisateur final après le passage de l'algorithme de cet exemple est donc représenté à la figure 5.12.

## 5.5 Calcul des coordonnées

*La dernière étape de notre algorithme consiste à calculer les coordonnées des différents sommets de notre graphe à l'aide du réalisateur.*

L'algorithme a maintenant le réalisateur $T_1$, $T_2$ et $T_3$ pour chaque nœud et ces réalisateurs coupent le graphe en trois régions par un i-path. Pour un sommet interne $v$ de $G$ on dit que $v_i$ est le nombre de triangles élémentaires de la région $R_i(v)$.

Reprenons notre exemple à la figure 5.4a, nous avons donc $(3_1, 3_2, 3_3) = (2, 3, 2)$. On peut étendre cette définition pour les sommets externes tels que $v_i = 2n - 5$, $v_{i+1} = v_{i+2} = 0$ pour les racines des $T_i$.
*On aura donc pour notre exemple :* $(0_0, 0_1, 0_2) = (7, 0, 0)$, $(1_0, 1_1, 1_2) = (0, 7, 0)$, $(2_0, 2_1, 2_2) = (0, 0, 7)$.

Montrons maintenant que nous avons une représentation barycentrique de $G$ :

**Théorème 5.5.1.** *La fonction $f : c \in V(G) \rightarrow \frac{1}{2n-5}(v_1, v_2, v_3)$ est la représentation barycentrique de $G$ et la labélisation de $G$ induite par $f$ est identique à la labélisation donnée par $G$.*

*Démonstration.* La première condition de la définition est clairement satisfaite, il faut donc juste prouver la deuxième condition.
Considérons une arête $\{x, y\}$ et un sommet $z$ n'appartenant pas à cette arrête. Si $z$ est un sommet externe alors il vérifie la condition $z_k = 2n-5 > x_k, y_k$ où $k$ est le réalisateur $T_k$ ayant pour racine $z$. Sinon $z$ est un sommet interne et $x, y \in R_k(z)$ pour un certain $k$ et donc on a bien que $z_k > x_k, y_k$.

On a bien la confirmation que la deuxième condition est respectée et donc f est bien la représentation barycentrique de $G$.

□

En s'appuyant sur la propriété 5.2.1, on en déduit que :

**Corollaire 5.5.1.** *Soit $a, b, c$ les racines de $T_1, T_2, T_3$, pour tous choix de positions non colinéaires de $a, betc$, $f : v \rightarrow \frac{1}{2n-5}(v_1 a, v_2 b, v_3 c)$ est l'embedding planaire linéaire de $G$ dans le plan couvert par $a, b, c$.*

En choisissant en particulier les points sur la grille $a = (2n-5, 0), b = (0, 2n-5), c = (0, 0)$, on obtient que l'application de $v \in V(G) \rightarrow (v_1, v_2)$ est l'embedding linéaire de $G$ sur une grille de $(2n-5) \times (2n-5)$.

Schnyder a ensuite voulu rendre cette représentation sur une grille plus compacte de taille $(n-2) \times (n-2)$. Pour ce faire, il a utilisé la représentation barycentrique faible.

Pour ce faire, considérons toujours notre graphe $G$ comme précédemment ainsi que $T_1, T_2, T_3$ et $v'$ le nombre de sommets de la région $R_i(v)$ dont le (i-1)-path commençant en $v$ a été supprimé.

Sur notre exemple de la figure 5.4a, nous aurions donc que $(3_1, 3_2, 3_3) = (2, 2, 2)$. On peut étendre cette définition aux sommets extérieurs de telle sorte que $v'_i = n-2$, $v'_{i+1} = 1$, $v'_{i+2} = 0$.

Montrons le théorème suivant :

**Théorème 5.5.2.** *Soit $u$ et $v$ deux sommets distincts. Si $v$ est un sommet interne et $u \in R_i(v)$ alors $(u'_i, u'_{i+1}) <_{lex} (v'_i, v'_{i+1})$.*

*Démonstration.* Prenons d'abord l'implication $u \in R_k(v) - P_{k-1}(v) \Rightarrow u'_k < v'_k$. C'est trivial si $u$ est un sommet externe alors $u$ est la racine de $T_{k+1}$ et $u'_k = 0$ alors que $v'_k \geq 1$.
Si par contre $u$ est un sommet interne, on suppose que $u \in R_i(v)$ ce qui implique que $u'_i \leq v'_i$. Si $u \notin P_{i-1}(v)$ on a $u'_i < v'_i$. Sinon $u \in R_{i+1}(v) - P_i(v)$ et $u'_{i+1} < v'_{i+1}$

□

Ce théorème implique que la fonction $v \in V(G) \rightarrow (v'_1, v'_2, v'_3)$ est injective et par l'argument similaire à précédemment on a que $v \in \frac{1}{n-1}(v'_1, v'_2, v'_3)$ est une représentation barycentrique faible. On peut ainsi en déduire le théorème suivant :

**Théorème 5.5.3.** *L'application $v \in V(G) \rightarrow (v'_1, v'_2)$ est un embedding linéaire de $G$ sur une grille de taille $(n-2) \times (n-2)$.*

## Idée générale de l'algorithme

Cet algorithme calcule les coordonnées de chaque sommet d'un graphe planaire donné, une fois que la labélisation de Schnyder a été réalisée.

L'algorithme procède de la façon suivante :

- Les sommets de la face externe sont triés et stockés. On extrait et trie les trois arbres associés aux sommets de cette face.
- On initialise les coordonnées des sommets de la face externe. Le premier sommet est placé à ($num\_nodes$ - 2, 1), le deuxième à (0, $num\_nodes$ - 2) et le troisième à (1, 0).
- Pour chaque sommet du graphe (qui n'est pas un sommet de la face externe), les coordonnées sont calculées comme suit :
  1. Pour chaque arbre (i=0, 1, 2), le code calcule combien de descendants le sommet a dans les deux autres arbres. C'est la somme des tailles des sous-arbres des descendants dans l'arbre $i$. Cette somme est stockée dans $n\_values$[i].
  2. Les coordonnées du sommet sont ensuite mises à jour pour être ($n\_values$[0], $n\_values$[1]).

La fonction retourne un dictionnaire $coordinates$ qui, pour chaque sommet du graphe, donne ses coordonnées dans la représentation de Schnyder. La représentation de Schnyder d'un graphe planaire est une disposition des sommets du graphe dans le plan, où chaque sommet est représenté par un point dans le plan, et chaque arête par un segment de droite reliant deux points.

## Pseudo-code

*Voir l'algorithme 8 $calculate\_coords()$*

## Fonctionnement

*Voici comment cet algorithme fonctionne étape par étape :*

**Initialisation** : L'algorithme trie les sommets de la face externe du graphe et extrait les nœuds correspondants de chaque arborescence dans $tree\_nodes\_arr$. Ensuite, il initialise les coordonnées de ces sommets de la face externe, leurs coordonnées seront initialisées comme suit :

1. Le premier sommet aura les coordonnées ($num\_nodes$ - 2, 1),
2. Le deuxième sommet aura les coordonnées (0, $num\_nodes$ - 2),
3. Le troisième sommet aura les coordonnées (1, 0).

**Calcul des coordonnées des autres sommets** : Pour chaque sommet node qui n'est pas sur la face externe, l'algorithme calcule ses coordonnées de la manière suivante :

1. Il initialise un tableau $n\_values$ à [0, 0, 0]. Ce tableau sera utilisé pour stocker les calculs intermédiaires nécessaires pour obtenir les coordonnées du sommet.
2. Il parcourt ensuite chaque arborescence pour ce sommet node (noté par l'indice i). Pour chaque arborescence, il parcourt les deux autres arborescences (notées par les indices $(i + 1)\%3$ et $(i - 1)\%3$) et incrémente $n\_values[i]$ par la taille du sous-arbre de chaque sommet rencontré dans ces deux arborescences.
3. Après avoir parcouru toutes les arborescences pour ce sommet node, il ajuste $n\_values[i]$ en le diminuant par la taille du sous-arbre du sommet node dans l'arborescence i et la profondeur du sommet node dans l'arborescence $(i - 1)\%3$.
4. Finalement, il met à jour les coordonnées du sommet node en utilisant $n\_values[0]$ et $n\_values[1]$.

**Algorithm 8** calculate_coords

**Require:** $G$ une graphe du module NetworkX
**Require:** $tree\_node$ un dictionnaire sommet, triplet d'arbres associés
**Require:** $ef$ la face externe du graphe
**Ensure:** Un dictionnaire qui associe le sommet à sa coordonnée
  Tri de la face externe
  Extraction des nœuds correspondants de l'arbre
  Initialisation de la coordonnées du premier sommet de la face externe à (n-2,1)
  Initialisation de la coordonnées du second sommet de la face externe à (0,n-2)
  Initialisation de la coordonnées du troisième sommet de la face externe à (1,0)
  **for** chaque sommet **do**
    **if** sommet dans face externe **then**
      Continuer
    **end if**
    Initialisation $n\_value$ comme un triplet de 0
    **for** i de 0 à 2 **do**
      **for** $t\_node$ dans $[(i+1)\%3, (i+1)\%3]$ **do**
        Initialisation de $current\_node$ à $tree\_nodes[node][t_node]$
        **while** $current\_node$ non nul **do**
          Incrémentation de $n\_value[i]$ de la taille du sous-arbre de $tree\_nodes$
          $current\_node$ devient parent de $current\_node$
        **end while**
      **end for**
      Décrémentation de $n\_values[i]$ de la taille du sous-arbre de $tree\_nodes[node][i]$
      Décrémentation de $n\_values[i]$ de la profondeur de $tree\_nodes[node][(i-1)\%3]$
    **end for**
    Mise à jour de la coordonnée du sommet
  **end for**
  **return** Le dictionnaire de coordonnées

**Renvoie des résultats** : Après avoir calculé les coordonnées pour tous les sommets, l'algorithme renvoie le dictionnaire $coordinates$ qui contient les coordonnées de tous les sommets du graphe.

Notez que la représentation barycentrique est utilisée ici pour obtenir une représentation en deux dimensions du graphe planaire triangulé. Chaque sommet est représenté par un point dans le plan et chaque arête est représentée par un segment de droite reliant les points correspondants.

## Preuve d'exactitude

*Prouvons maintenant l'exactitude de notre algorithme :*

**Initialisation des coordonnées** : L'algorithme initialise les coordonnées des sommets de la face externe à des valeurs fixes. Ces valeurs correspondent à une représentation barycentrique du triangle formé par ces trois sommets.

**Calcul des coordonnées** : Pour chaque sommet interne, l'algorithme calcule ses coordonnées en fonction des tailles des sous-arbres dans les arborescences de Schnyder et de leurs profondeurs. Ces calculs sont basés sur les propriétés de la représentation barycentrique et assurent que chaque sommet est correctement positionné par rapport à ses voisins dans le graphe.

**Propriétés de la représentation barycentrique** : La représentation barycentrique assure que chaque sommet est représenté par un point unique dans le plan et que chaque arête est représentée par un segment de droite reliant les points correspondants. De plus, elle garantit que si deux sommets sont adjacents dans le graphe, leurs points correspondants sont également adjacents dans la représentation barycentrique.

Ces éléments nous permettent de conclure que l'algorithme est exact, car il calcule une représentation barycentrique correcte du graphe planaire triangulé à partir d'une labélisation de Schnyder correcte.

### Preuve de complexité

**Tri des nœuds de l'arbre** : Cette opération coûte **O(1)**, car l'on trie les nœuds de la face externe. Puisque la face externe contient généralement un nombre constant de nœuds (3 pour un graphe planaire triangulé), cette opération est considérée

comme O(1) dans la pratique au lieu d'O(n log n).

**Initialisation des coordonnées** : Cette opération est **O(1)**, car elle affecte simplement une valeur fixe à chaque nœud de la face externe.

**Parcours des nœuds du graphe** : Cette opération est **O(n)**, car elle itère une fois sur chaque nœud du graphe.

**Calcul des coordonnées pour chaque nœud** : À l'intérieur de cette boucle, se trouve une autre boucle qui parcourt chaque arête du nœud en cours. Dans le pire des cas, ce coût peut être considéré comme O(d), où d est le degré du nœud (le nombre de voisins). Dans un graphe planaire, le degré moyen est lié à la taille du graphe par la formule $d = \frac{2E}{n}$, où E est le nombre d'arêtes. Cependant, comme chaque arête est considérée deux fois (une fois pour chaque nœud à ses extrémités), le coût total est **O(n)**. Ensuite, pour chaque arête, on parcourt l'arborescence jusqu'à la racine, ce qui coûte O(log n) dans le pire des cas pour les arbres équilibrés. Par conséquent, le coût total pour cette partie est **O(n log n)**.

La complexité totale de notre algorithme est donc **O(n log n)**.

## Exemple

*Reprenons notre exemple précédent en partant du réalisateur de la figure 5.12.*

Tout d'abord, il est assez simple d'initialiser nos trois premiers sommets aux coordonnées respectives suivantes : (4,1),(0,4),(1,0).

Commençons donc à regarder le sommet $v3$. Pour la coordonnée x, après le parcours des nœuds de l'arbre, nous avons notre $n\_values$ incrémentée à 5. En soustrayant la taille du sous-arbre, on obtient 4, on retire ensuite la profondeur du i-path(i-1) et l'on obtient la coordonnée 2. On réitère l'opération pour y et l'on obtient à nouveau un $n\_values$ valant 2. On a donc la coordonnée (2,2) pour notre sommet $v3$.

On réitère l'opération pour le sommet $v4$, on obtient ainsi un $n\_values$ de 3 pour x et de 1 pour y. On a donc les coordonnées (3,1) pour notre sommet $v4$.

Enfin, nous traitons le sommet $v5$ et l'on obtient les coordonnées (1,3) pour ce sommet.

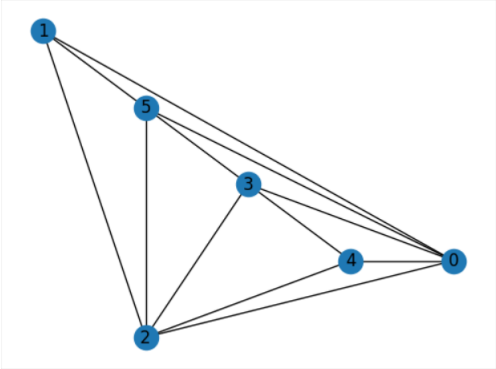



FIGURE 5.13 – Embedding planaire de notre exemple

Nous avons donc toutes les coordonnées des sommets de notre graphe comprises sur une grille de taille $(n-2) \times (n-2)$. Nous pouvons donc représenter notre graphe comme un embedding planaire linéaire représenté à la figure 5.13.

# Conclusion

Ce mémoire fournit une base de connaissance sur les graphes planaires et plusieurs de leurs dérivées, ainsi que sur l'algorithme de Schnyder permettant la représentation de ceux-ci. Ceci aura permis la conception d'un algorithme fonctionnel prouvant le théorème de Schnyder de 1990 ainsi qu'un étoffement de l'article scientifique si rapportant.

Ce document aura donc montrer qu'à partir d'un graphe planaire connexe, il est toujours possible d'obtenir une représentation planaire linéaire de celui-ci sur une grille de $(n-2) \times (n-2)$ à l'aide de l'algorithme de Schnyder. Cette conception peut tout de même encore être optimisée comme lors de l'étape du calcul de coordonnées qui a une complexité temporelle en O(n log n) au lieu de O(n).

L'implémentation pourrait être plus complète en résolvant le problème de labélisation lors de l'étape si rapportant et empêchant donc certains types de graphe d'être compatible avec l'algorithme.

Finalement, ce mémoire et cette implémentation sont une bonne base pour toutes personnes voulant s'intéresser à la représentation de graphe planaire et qui ne sont pas expertes dans ce domaine. Ils seront aidés par nombreux exemples illustrant des concepts complexes.

# Bibliographie


[1] *Takao Nishizeki et Md. Saidur Rahman : "Planar Graph Drawing" - chapitre 5.4 - 2004*

[2] *Takao Nishizeki et Md. Saidur Rahman : "Planar Graph Drawing" - chapitre 4.3 - 2004*

[3] `https://networkx.org` *- consulté le 15/07/2023*

[4] *Walter Schnyder : "Embedding Planar Graphs on the Grid" - 1990*

[5] *Martin Mader : "Planar Graph Drawing" - 2008*

[6] `https://jupyter.org/` *- consulté le 15/07/2023*

[7] *Antonios Symvonis et Chrysanthi Raftopoulou : "Visualisation of graphs Planar straight-line drawings Schnyder realiser" - 2020*

[8] *Luca Castelli Aleardi et Steve Oudot : "Introduction à la géométrie algorithmique et ses applications" - 2020*

[9] *W. T. TUTTE : "A THEORY OF 3-CONNECTED GRAPHS" - 1961*

[10] *David Eppstein : "Walter Schnyder's Grid-Embedding Algorithm" - 2022*

[11] *Stephen Kobourov : "Canonical Orders and Schnyder Realizers" - 2015*

[12] *Hadrien Mélot et Véronique Bruyère : "Structures de données 1" - Chapitre 6 et 7 - 2018*

[13] *Daniel Tuyttens : "Théorie des graphes et Optimisation combinatoire" - Chapitre 1,3,4 - 2019*

[14] *Nicolas BONICHON : "Quelques algorithmes entre le monde des graphes et les nuages de points." - Chapitre 2 - 2013*

[15] *H. de Fraysseix, J. Path, R. Pollack : "Small sets supporting F&y embeddings of planar graphs" - 1988*

[16] *J. E. Hopcroft and R. E. Tarjan : " Dividing a graph into triconnected components." - 1973*

[17] *W. Schnyder : "Planar graphs and poser dimension" - 1989*