

Génération de graphes planaires

Directeur de Projet :

Hadrien Melot

Étudiant :
Foucoux Noa

MAB1 Sciences Informatiques

Année Académique 2022-2023

Faculté des Sciences, Université de Mons



Contents

1	Introduction	3
2	Rappels sur la théorie des graphes	4
2.1	Les graphes	4
2.1.1	Sous-graphe	4
2.1.2	Sous-graphe induit	5
2.2	Connexité des graphes	5
2.2.1	Graphe connexe	5
2.2.2	Composante connexe	5
2.3	Isomorphisme de graphes	5
2.3.1	Graphes isomorphes	5
2.3.2	Invariants de graphe	5
2.4	Planarité des graphes	5
2.4.1	Graphe planaire	5
2.4.2	Déférence entre un graphe plane et un graphe planaire	6
3	Analyses des graphes de ville	6
3.1	Méthodes d'analyses de graphe	7
3.1.1	Utilisation de la librairie OSMNX	7
3.2	Analyse de la répartition des degrés	10
3.3	Sélection de graphes de taille équivalente pour chaque ville	13
3.4	Diamètre	14
3.5	Rayon	15
3.6	Plus court chemin moyen	16
3.7	La connectivité moyenne des degrés d'un noeud	17
3.8	Irrégularités	19
3.9	Répartition des noeuds par kilomètre carré	21
3.10	Analyse des arcs	21
4	Algorithme de génération de graphe planaire	23
4.1	Exemple de fonctionnement de l'algorithme	23
4.2	Fonction <code>graph_generation</code>	26
4.2.1	Paramètres de <code>graph_generation</code>	26
4.2.2	Nombre de noeuds	27
4.2.3	Matrice d'adjacence Numpy	27
4.2.4	Position des noeuds	27
4.2.5	Graphe NetworkX	27
4.2.6	Plus grande composante connexe	27
4.2.7	Suppression des noeuds de degré 2 et 0 pour l'affichage	28
4.2.8	Complexité en temps	28
4.3	Node Generation	30
4.3.1	Paramètres de <code>node_generation</code>	30
4.3.2	Description	31
4.3.3	Complexité	31

4.4	Edges Generation	33
4.4.1	Positions possibles des arêtes	33
4.4.2	Complexité	34
5	Autres fonctions utilisées dans le projet	35
5.1	Fonction <code>fct</code>	35
5.2	Fonction <code>cumulative</code>	36
5.3	Fonction <code>inverse</code>	36
5.4	Fonction <code>position_in_adj</code>	37
5.5	Fonction <code>position_from_adj</code>	37
6	Complexité globale de l'algorithme de génération de graphe.	38
7	Résultats	39
7.1	Cas standard	39
7.2	Nombre faible de noeuds	41
7.3	Grand nombre de noeuds	42
7.4	Paramètre <code>usa</code>	43
7.5	Paramètre <code>color</code>	45
7.6	Comparaison	45
8	Perspectives futures	46
8.1	Complexité en espace	46
8.1.1	Langage	46
8.1.2	Librairie	47
8.1.3	Utilisation de matrice creuse	47
8.2	Génération géométrique	47
8.3	Parcours en largeur avant NetworkX	47
8.4	Génération d'un pays	48
9	Conclusion	49

1 Introduction

La génération naïve de graphes aléatoires peut paraître un problème simple mais en réalité si nous voulons respecter leur distribution, si nous ajoutons des contraintes telle que la planarité, il devient plus complexe. La génération aléatoire d'un grand graphe planaire l'est d'autant plus, car plus le nombre de noeuds augmente, plus la planarité est difficile à respecter.

Il existe de nombreux algorithmes de génération de graphe planaire comme les différentes méthodes présentées dans l'article [3]. Cependant, dans ce projet, nous devons aussi respecter les caractéristiques d'un environnement urbain, ce qui ajoute des contraintes à la génération et à nos algorithmes.

Les graphes planaires sont utilisés dans de nombreux domaines, tels que :

- Les systèmes de transport : pour modéliser les réseaux routiers ou les réseaux ferroviaires.
- L'urbanisme : pour modéliser les plans de villes, les réseaux de transports en commun et les réseaux d'alimentation en eau.
- Les circuits électroniques : les graphes planaires sont utilisés pour représenter les composants électroniques et leurs connexions dans un circuit électrique.
- Les réseaux sociaux : pour modéliser les relations entre les différents membres d'un groupe.

Dans ce projet, nous allons nous concentrer sur le système de transport routier en milieu urbain, mais il pourrait être étendu à d'autres infrastructures similaires. La génération aléatoire permettrait de simuler un monde imaginaire et d'y effectuer des tests de trafic, d'emplacement d'arrêts de transport en commun [6], ... afin d'améliorer l'organisation de nos villes futures.

Dans mon pré-rapport, j'avais émis l'hypothèse de rendre les graphes paramétrables et ainsi permettre de comparer différents environnements du monde réel. Nous pourrions penser que les graphes routiers de type nord-américain [8] et européens seraient aussi différents en apparence qu'en invariants (2.3.2) de graphes. Cependant, vous observerez dans la section 3 qu'aucun d'entre eux n'a permis de différencier les différentes organisations urbaines.

Afin de mener à bien ce projet, j'ai développé un algorithme de génération aléatoire de graphe planaire en milieu urbain respectant des contraintes prédéfinies que nous aborderons dans la section 4.2.1.

Cet algorithme utilise les résultats des analyses pour créer une répartition des noeuds similaires à celle de la réalité. De plus, une fois le graphe généré, on ne garde que la plus grande composante connexe, ce qui rend celui-ci d'autant plus réaliste. En effet, dans un environnement urbain, toutes les routes sont reliées entre elles.

Après avoir introduit l'importance des graphes planaires dans la modélisation des systèmes de transport urbain et avant d'approfondir notre discussion sur la génération aléatoire de graphes planaires en milieu urbain, il est essentiel de rappeler les concepts fondamentaux de la théorie des graphes.

2 Rappels sur la théorie des graphes

Tout d'abord, faisons un petit rappel sur différentes définitions qui nous seront utiles dans l'analyses des villes dans la section 3.

2.1 Les graphes

Un graphe G [2] est une paire telle que :

- $G = (V, E)$ avec $E \subseteq [V]^2$;
- V est l'ensemble des nœuds $\{v_1, v_2, v_3, \dots, v_n\}$;
- E est l'ensemble des arcs $\{(v_i, v_j), \dots\}$.

Le degré d'un nœud est le nombre d'arcs associés à ce nœud. Les figures 1 et 2 illustrent 2 graphes aléatoires, H et I .

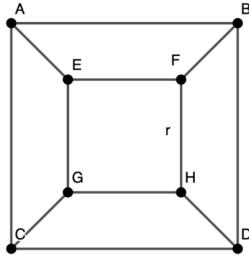


Figure 1: Graphe H

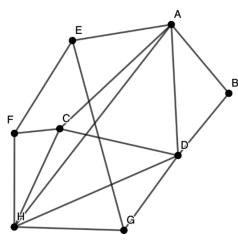
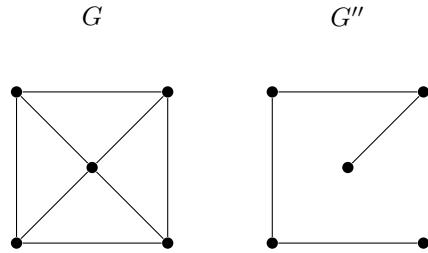


Figure 2: Graphe I

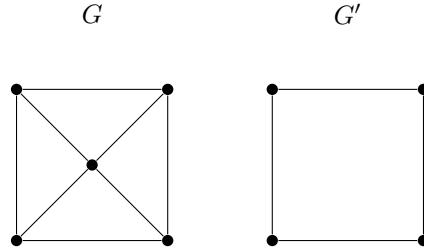
2.1.1 Sous-graphe

Un graphe G'' [2] est un sous-graphe de G et s'écrit $G'' \subseteq G$. C'est-à-dire que G contient les arcs et les nœuds de G'' .



2.1.2 Sous-graphe induit

G' est un sous-graphe induit de G [2] car $G' \subseteq G$ et G' contient tous les arcs $xy \in E$ avec $x, y \in V'$.



2.2 Connexité des graphes

2.2.1 Graphe connexe

Un graphe est connexe [4] si pour toutes paires de nœuds u, v , il existe un chemin entre u et v .

2.2.2 Composante connexe

Une composante connexe G' de G [2] est un sous-graphe induit maximal connexe.

2.3 Isomorphisme de graphes

2.3.1 Graphes isomorphes

Deux graphes $G = (V, E)$ et $G' = (V', E')$ sont isomorphes [2] s'il existe une bijection $f : V \rightarrow V'$ telle que $(v, v') \in E \iff (f(v), f(v')) \in E'$

Les graphes G et H des figures 3 et 4 sont isomorphes.

2.3.2 Invariants de graphe

Un invariant de graphe [4] est une valeur qui est préservée par isomorphisme.

Exemples

Par exemple, le degré des nœuds est un invariants qui va nous être utile dans la section 3 pour analyser les graphes de différentes villes.

2.4 Planarité des graphes

2.4.1 Graphe planaire

Un graphe planaire [2] est une paire (V, E) finie avec les propriétés suivantes :

1. $V \subseteq \mathbb{R}^2$;

2. chaque arc est relié entre 2 noeuds ;
3. 2 arcs sont différents si leur ensemble est constitué d'au moins 1 noeud différent ;
4. il existe un graphe isomorphe pour lequel l'intérieur d'un arc ne contient pas de noeuds et pas de points d'un autre arc.

Le graphe J de la figure 4 est un graphe planaire.

2.4.2 Différence entre un graphe plane et un graphe planaire

Un graphe plane est un graphe planaire représenté de manière plane. Un graphe planaire est un graphe possédant une représentation sous forme d'un graphe plane. En réalité, dans ce projet, je vais réaliser une génération aléatoire de graphes planes. Les graphes planes étant des graphes planaires par définition, nous allons appeler ce projet "Génération aléatoire de graphe planaire".

Prenons comme exemple deux graphes isomorphes que nous appellerons H et J . Ces graphes sont planaires car il existe un graphe isomorphe où aucune arête n'est intersectée par une autre arête ou noeud (n'appartenant pas aux extrémités de cette arête). Le graphe H de la figure 3 est une représentation plane. Le graphe J de la figure 4 étant isomorphe à H à un renommage des sommets près, il est planaire.

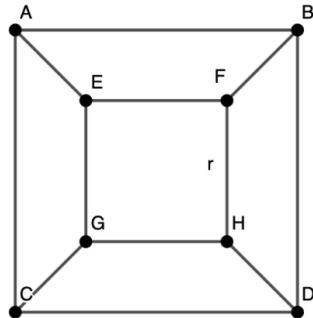


Figure 3: Graphe plane H

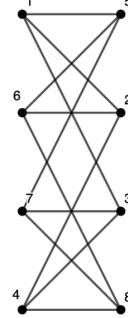


Figure 4: Graphe planaire J

3 Analyses des graphes de ville

L'analyse de graphes de villes existantes va permettre d'obtenir des paramètres et des caractéristiques utilisables dans l'algorithme de génération aléatoire de graphe planaire.

Les analyses sont disponibles dans le Jupyter Notebook appelé `plot_analysis.ipynb` dans le projet mais également sur le [GitHub](#). Il contient un dictionnaire nommé `pos_dict` permettant de changer ou d'ajouter des villes aux analyses et ainsi comparer d'autres organisations urbaines. Les tableaux des figures 5 et 6

représentent les paramètres utilisés dans les analyses pour comparer chacune des villes.

Ville	Latitude	Longitude	taille d'un côté du carré
Mons	50.453	3.95	900
New York	40.7443	-73.9903	4000
Washington	38.9064	-77.0295	3000
Charleroi	50.4116	4.4448	900
Dallas	32.9096	-96.9506	3000
Barcelone	41.3899	2.1701	2000

Figure 5: paramètres des graphes des différentes villes

Latitude 1	Latitude 2	Longitude 1	Longitude 2
25.72	25.39	-80.0252	-80.517

Figure 6: paramètres du graphe de Miami

3.1 Méthodes d'analyses de graphe

Il existe différentes méthodes pour générer et analyser les graphes des villes. La première serait de les générer à la main en prenant les coordonnées de chacun des nœuds et recréer ainsi le graphe. Malheureusement, cette méthode est terriblement fastidieuse pour des grandes villes. En testant différentes librairies de génération de graphes à partir de carte existante, [OSMNX](#) a retenu notre attention.

3.1.1 Utilisation de la librairie [OSMNX](#)

[OSMNX](#) est une bibliothèque [Python](#) qui permet de télécharger des données cartographiques provenant de [OpenStreetMap](#). Elle permet de sélectionner un carré de $2Nx2N$ mètres autour d'un point central en utilisant les coordonnées géographiques.

Grâce à cette librairie, la création et l'analyse des graphes de grandes villes a été plus facile. De plus, [OSMNX](#) génère des graphes compatibles avec [NetworkX](#). On a donc pu utiliser cette librairie pour générer une représentation de ces graphes. Sur les figure 7, 8 et 9, nous avons les graphes des villes générés via [OSMNX](#) et [NetworkX](#) grâce au paramètre des tableaux 5 et 6.

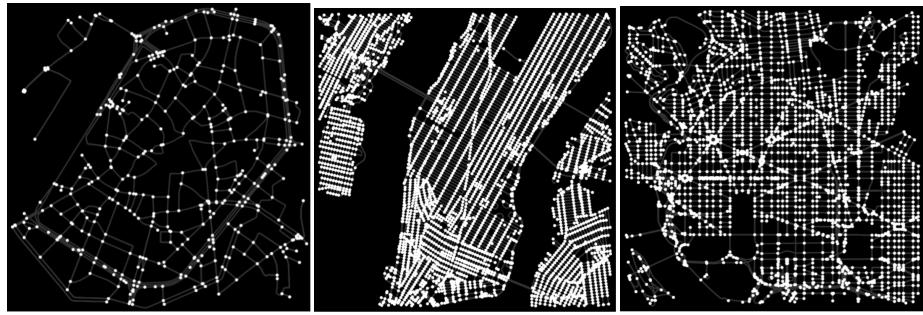


Figure 7: Graphes des villes de Mons, New York et Washington



Figure 8: Graphes des villes de Charleroi, Dallas et Miami

De plus, [NetworkX](#) possède déjà de nombreuses analyses d'invariants ce qui permet de les analyser facilement. On peut apercevoir sur les images des figures [7](#) et [8](#) que leur organisation est assez différente et représentative de 2 endroits du globe à analyser.

On peut également remarquer sur la figure [9](#) que Barcelone était également une 7ème ville à analyser, mais elle n'a pas été prise en compte dans les analyses comparatives. En effet, la plupart des villes américaines ayant une organisation hippodamienne [\[8\]](#) et Barcelone l'étant aussi, cela aurait pu entraîner des erreurs dans les analyses.

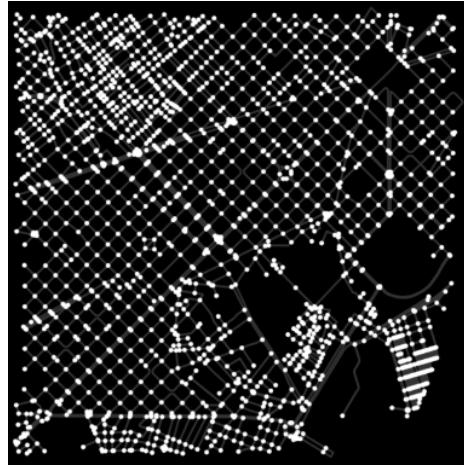


Figure 9: Graphe de la ville de Barcelone

3.2 Analyse de la répartition des degrés

Rappel

Le degré $\deg(v)$ d'un nœud est le nombre de voisins de ce nœud (le nombre de sommets adjacents).[\[4\]](#)

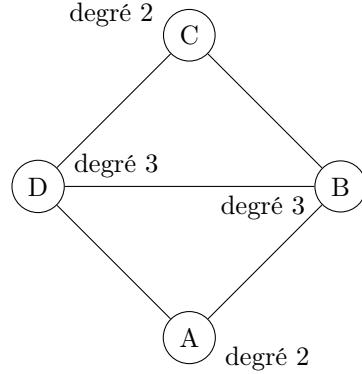


Figure 10: Graphique avec 4 nœuds et leurs degrés notés.

Lors de l'analyse de la répartition des degrés des nœuds des graphes des différentes villes, on a pu apercevoir une répartition assez inattendue pour les villes nord-américaines. En effet, on peut s'attendre à une répartition plus importante des nœuds de degré 4 que des nœuds de degré 3, mais à l'exception de New York, qui possède effectivement cette caractéristique, toutes les autres villes analysées ont une quantité de nœuds de degré 3 plus importante.

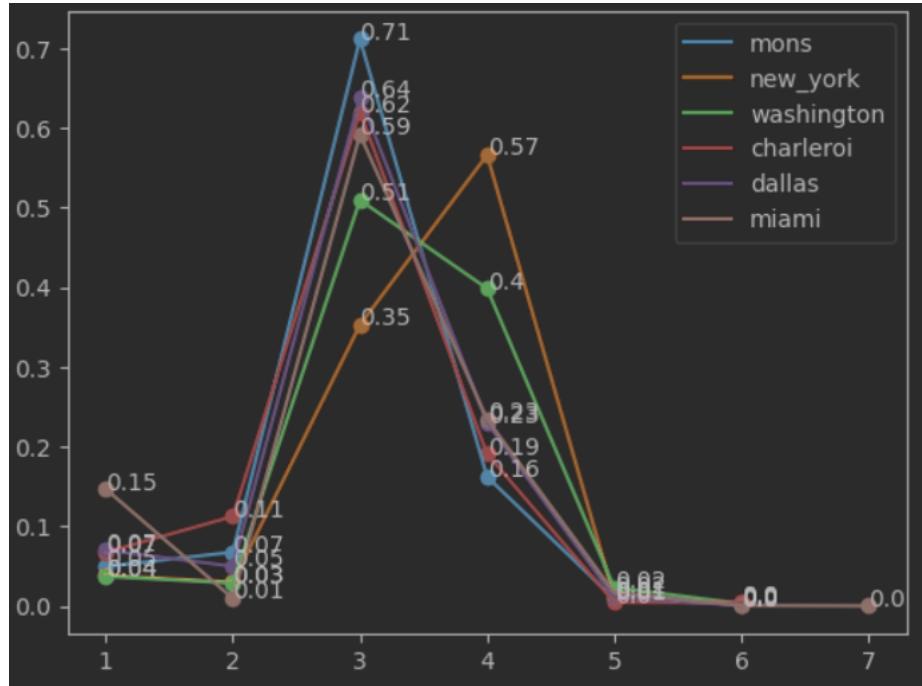


Figure 11: Répartition des degrés des villes

Analyse des nœuds à partir du degré 3

Sur le graphique obtenu, nous pouvons apercevoir que la courbe ressemble à une exponentielle décroissante. L'utilisation de la fonction `curve_fit` de [Scipy](#) a permis d'obtenir une répartition des nœuds à partir du degré 3.

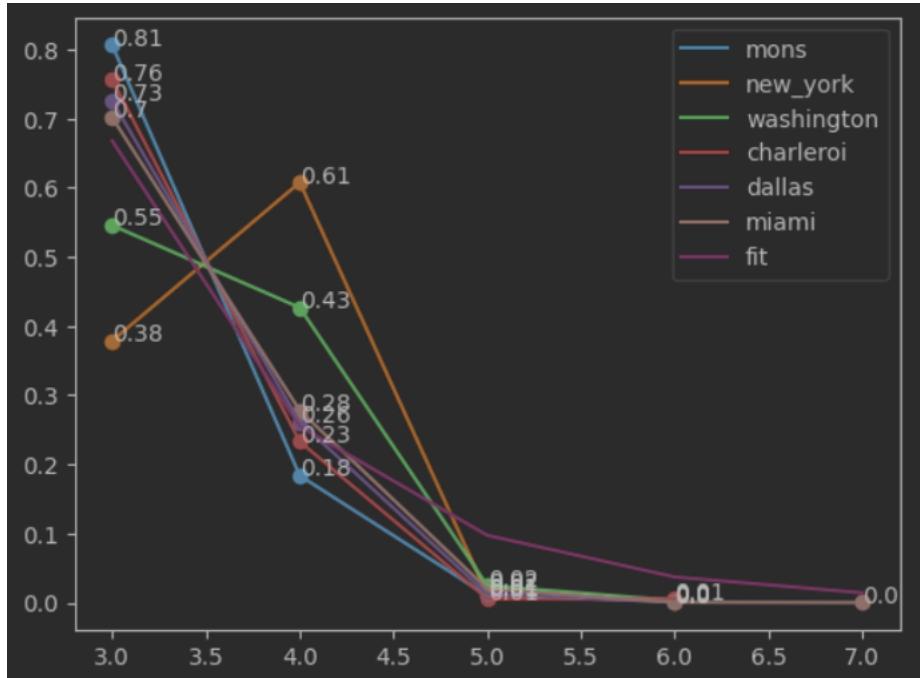


Figure 12: Répartition des degrés supérieurs à 3 des villes

Ville	Nombre de noeuds
Mons	392
New York	3752
Washington	2597
Charleroi	365
Dallas	1309
Miami	133925

Figure 13: Nombre de noeuds de degré > 3 par ville

Les résultats étant très satisfaisants, la fonction obtenue a été conservée pour le générateur. Une autre analyse a également été effectuée en éliminant New York et Washington qui ont des répartitions assez différentes par rapport aux autres villes, et la courbe obtenue est également très représentative des villes ayant des caractéristiques similaires en termes de noeuds.

Ayant obtenu deux fonctions différentes en fonction des villes sélectionnées, la fonction de répartition des noeuds sera paramétrable dans le générateur.

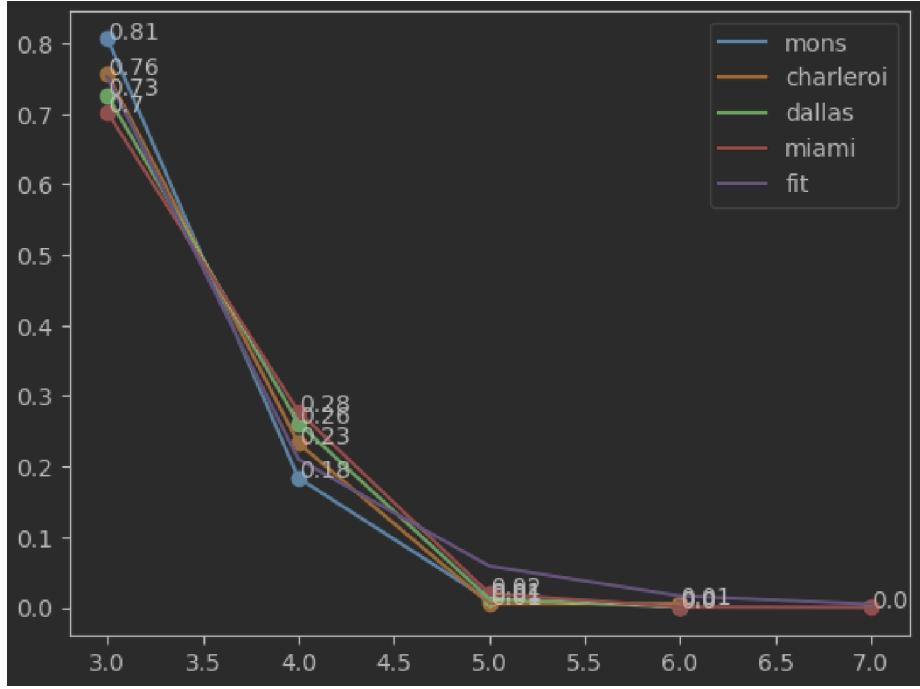


Figure 14: Répartition des degrés supérieurs à 3 des villes sans New York et Washington

Résultat général	Résultat simplifié
$12.01009375101943 \times e^{-0.9631486813699162x}$	$34.3793401927484 \times e^{-1.2743522467865194x}$

3.3 Sélection de graphes de taille équivalente pour chaque ville

Étant donné que de nombreux invariants de graphe sont très sensibles à la taille du graphe et qu'on peut apercevoir sur la figure 15 que les villes analysées ont toutes des tailles différentes, j'ai décidé de créer un graphe pour chaque ville ayant une taille équivalente. Pour cela, j'ai sélectionné un carré de sélection de taille égale au plus petit carré de sélection utilisé dans le tableau 5, c'est-à-dire le carré de sélection de la plus petite ville analysée. Sélectionner des graphes de taille égale pour chaque ville est très difficile et peut entraîner des erreurs pour les analyses ultérieures. En comparant le nombre de noeuds à la figure 16, on peut voir que le nombre de noeuds est comparable pour chacune des villes.

Ville	Nombre de nœuds
Mons	444
New York	4026
Washington	2780
Charleroi	1487
Miami	158724

Figure 15: Nombre de nœuds par ville

Ville	Nombre de nœuds
Mons	392
New York	209
Washington	340
Charleroi	365
Dallas	182

Figure 16: Nombre de nœuds pour les graphes de tailles équivalentes

Malheureusement, la manière dont Miami est organisée ne permet pas de faire cette modification. Elle a donc été exclue des analyses.

3.4 Diamètre

Rappel

Le diamètre d'un graphe G , noté $diam(G)$, est la plus grande distance entre 2 nœuds.[\[2\]](#)

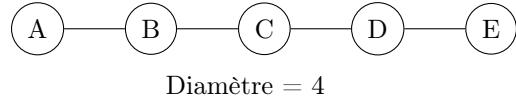


Figure 17: Graphe de type chemin avec son diamètre

Le diamètre est très sensible à la taille du graphe et les résultats obtenus sur des tailles de graphes différentes ne sont pas représentatifs. En revanche, pour des graphes de tailles égales, le diamètre est assez similaire entre chaque ville. Par conséquent, nous ne pouvons pas conclure de paramètres pour le générateur en ce qui concerne le diamètre.

Ville	Diamètre (Graphe classique)	Diamètre (Graphe de taille équivalente)
Mons	35	35
New York	82	32
Washington	66	29
Charleroi	29	29
Dallas	54	26

Figure 18: Diamètre des différents graphes de villes

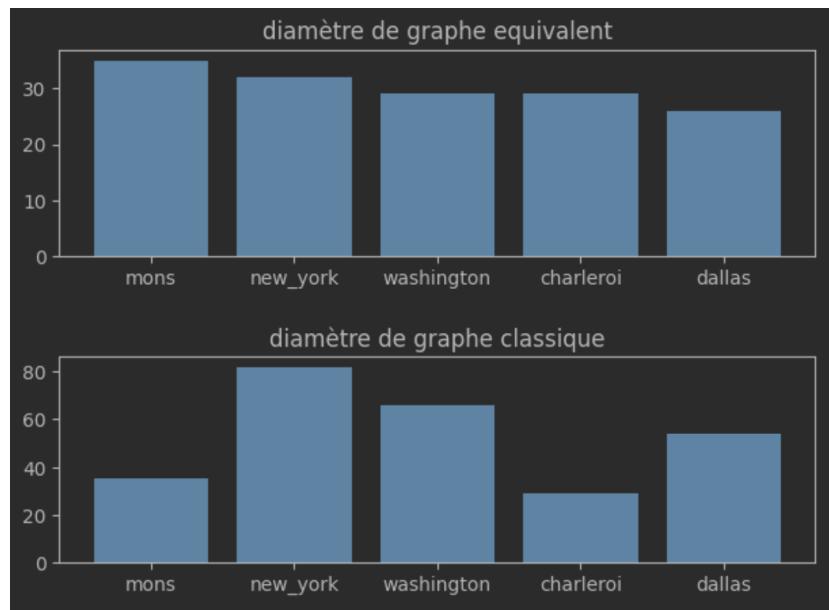


Figure 19: Graphique du diamètre des différents graphes de villes

3.5 Rayon

Rappel

Le rayon [2], noté $rad(G)$, est noté comme ceci :

$$rad(G) = \min_{x \in V(G)} \max_{y \in V(G)} d_G(x, y)$$

$$rad(G) \leq diam(G) \leq 2rad(G)$$

De la même manière, le rayon étant lui aussi très sensible à la taille du graphe, l'analyse ne peut être effectuée que sur des graphes équivalents en taille. Cependant, aucune différence notable ne ressort de cette comparaison.

Ville	Rayon (Graphe classique)	Rayon (Graphe de taille équivalente)
Mons	18	18
New York	41	37
Washington	37	18
Charleroi	18	28
Dallas	16	15

Figure 20: Rayon des différents graphes de villes

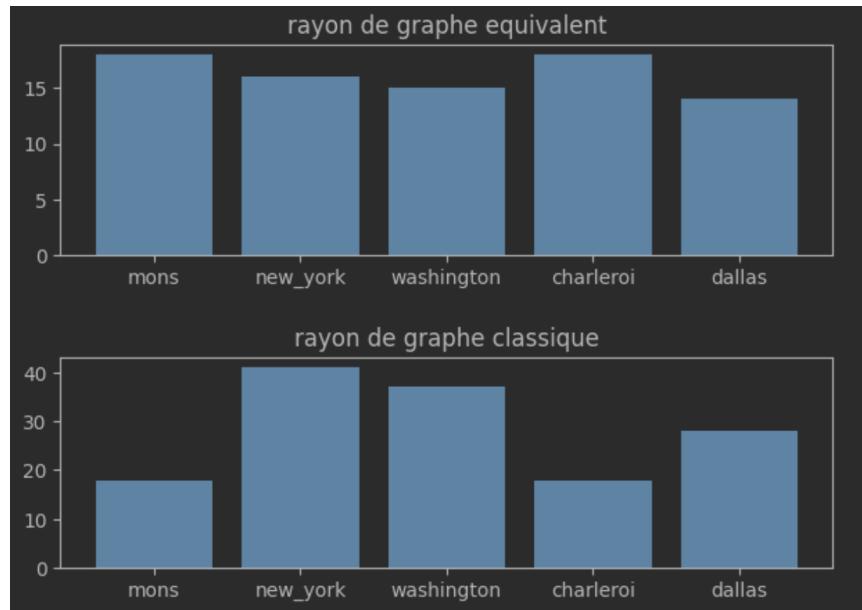


Figure 21: Graphique du rayon des différents graphes de villes

3.6 Plus court chemin moyen

Cet invariant est lui aussi très sensible à la taille du graphe et très peu de différences sont observées sur les figures 22 et 23.

Ville	Plus court chemin moyen (Graphe classique)	Plus court chemin moyen (Graphe de taille équivalente)
Mons	12.292223374616151	12.292223374616151
New York	32.16739454415862	25.76031490029279
Washington	25.76031490029279	12.741431318959409
Charleroi	12.741431318959409	21.238538396022594
Dallas	21.238538396022594	12.292223374616151

Figure 22: Plus court chemin moyen des différents graphes de villes

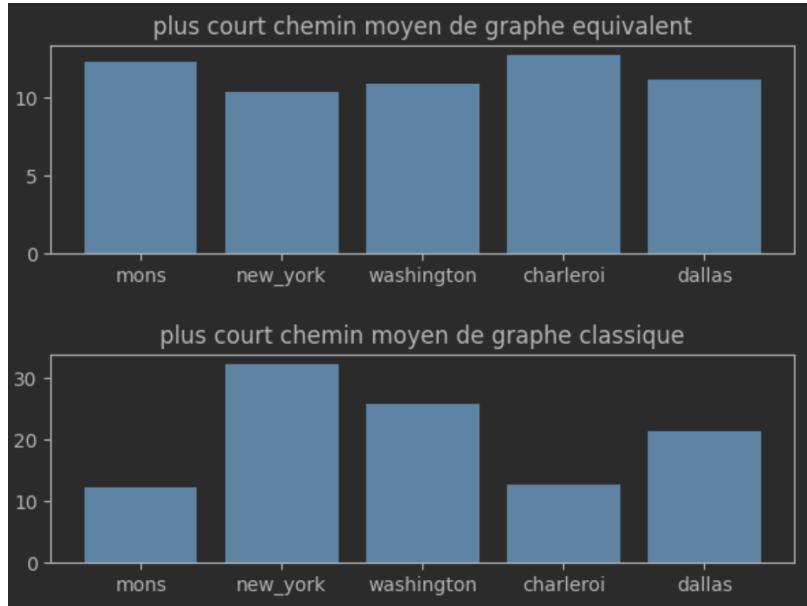


Figure 23: Graphique du plus court chemin des différents graphes de villes

3.7 La connectivité moyenne des degrés d'un nœud

Rappel

La connectivité moyenne des degrés d'un nœud est la traduction que j'ai choisie pour *Average degree connectivity*. En effet, dans la littérature, cet invariant n'est que très peu cité en français, je n'ai pas trouvé de traduction définie pour celui-ci. Il s'agit en réalité de la moyenne des degrés des plus proches voisins d'un nœud de degré k .^[5]

Prenons un exemple :



Figure 24: Exemple de graphique illustrant la moyenne des degrés de connectivité

La connectivité moyenne des degrés d'un nœud fait référence à la moyenne des degrés de ses voisins directs. Dans cet exemple, prenons le noeud B . Ses voisins directs sont les nœuds A et C . Le degré du noeud A est 1 et le degré du noeud C est 2. Ainsi, la connectivité moyenne des degrés du nœud B est $(1 + 2)/2 = 1.5$. La connectivité moyenne des nœuds de degré 2 est égale à 1.5 et La connectivité moyenne des nœuds de degré 1 est égale à 2.

Cet invariant n'est pas sensible à la taille du graphe, et nous pouvons observer de nombreuses similarités entre les différents graphes pour les nœuds de

degré 1 à 4. À partir de 5, la répartition des valeurs est très différente, mais il n'y a pas de différence notable entre les graphes nord-américains et européens. La répartition de ces valeurs est sans doute due au fait que le nombre de noeuds de degré supérieur à 5 est très faible.

Villes	Degré 1	Degré 2	Degré 3	Degré 4	Degré 5	Degré 6
Mons	3.1364	2.9500	3.1424	3.2882	3.3000	-
New York	3.4416	3.1542	3.4145	3.7697	3.4844	3.4444
Washington	3.3981	3.0375	3.3738	3.6552	3.4563	3.5625
Charleroi	3.4667	2.8300	3.0894	3.3265	3.200	2.5833
Dallas	3.0673	2.8311	3.1551	3.3933	2.7500	2.8333

Figure 25: Tableau des Average Degree Connectivity pour des graphes de taille différente

Villes	Degré 1	Degré 2	Degré 3	Degré 4	Degré 5	Degré 6
Mons	3.1364	2.9500	3.1424	3.2882	3.3000	-
New York	3.3333	3.2857	3.2864	3.7663	-	-
Washington	3.3125	3.2917	3.4291	3.6561	3.5750	2.8333
Charleroi	3.4667	2.8300	3.0894	3.3265	3.2000	2.5833
Dallas	2.7647	2.7903	3.0711	3.3056	1.8000	-

Figure 26: Tableau des Average Degree Connectivity pour des graphes de taille équivalente

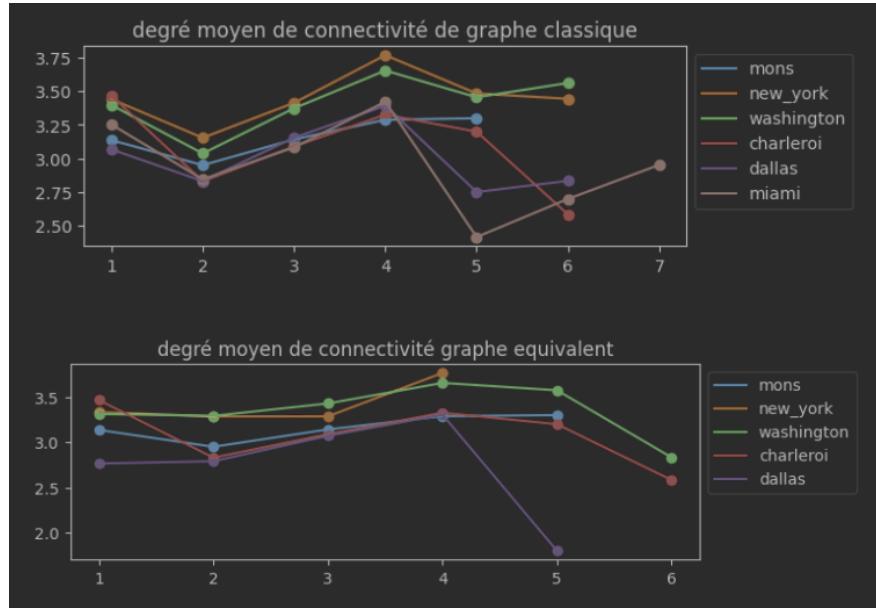


Figure 27: Graphique du degré moyen de connectivité des différents graphes de villes

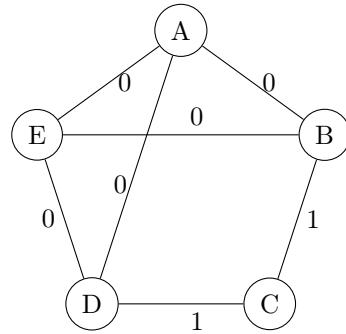
3.8 Irrégularités

Rappel

L'irrégularité d'un graphe G [4], noté $A(G)$, est :

$$A(G) = \sum_{\{i,j\} \in E} |d_i - d_j|$$

La figure 28 illustre un graphe ayant une irrégularité de 2.



$$\text{Irregularité} = 2$$

Figure 28: Graphe avec 5 nœuds formant un pentagone et une irrégularité de 2.

Cet invariant aurait pu être celui qui différencierait les graphes. Cependant, nous obtenons une fois de plus des valeurs très similaires.

Ville	Irregularité (Taille classique)	Irregularité (Taille équivalente)
Mons	2.2580061517129337	2.2580061517129337
New York	2.193963823247846	2.193963823247846
Washington	2.162247543224071	2.162247543224071
Charleroi	2.169754023686608	2.169754023686608
Dallas	2.1972047980631673	2.1972047980631673

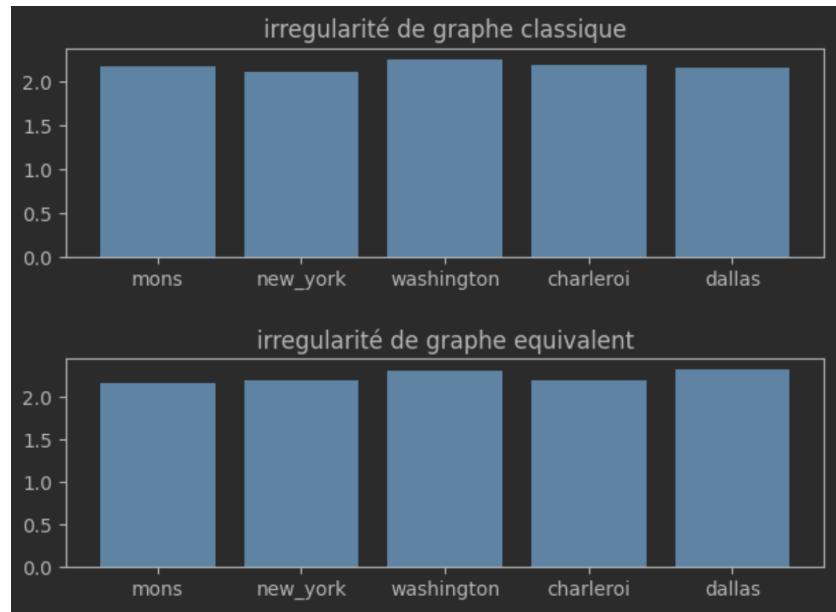


Figure 29: Graphique des irrégularités des différents graphes de villes

3.9 Répartition des nœuds par kilomètre carré

Le nombre de noeuds par kilomètre carré est aussi un invariant analysé. Il n'est normalement pas sensible à la taille du graphe. Le fait que Washington ait une valeur différente entre les 2 graphes est lié à l'emplacement où le carré de sélection pour les graphes de tailles équivalente de 1800 mètres de côté a été placé à Washington. Nous pouvons remarquer des différences assez importantes entre New York, Dallas, Washington et les autres graphes. Cela est assez logique car ces villes sont plus grandes et plus étendues, ce qui signifie qu'il y a moins de carrefour par kilomètre carré et donc moins de nœuds. Ce paramètre peut être utilisable dans le générateur.

Ville	Nœuds par km (Classique)	Nœuds par km (Équivalent)
Mons	88.2564	88.2564
New York	47.9914	55.7939
Washington	60.5946	93.2820
Charleroi	88.7771	88.7771
Dallas	34.9793	61.2184

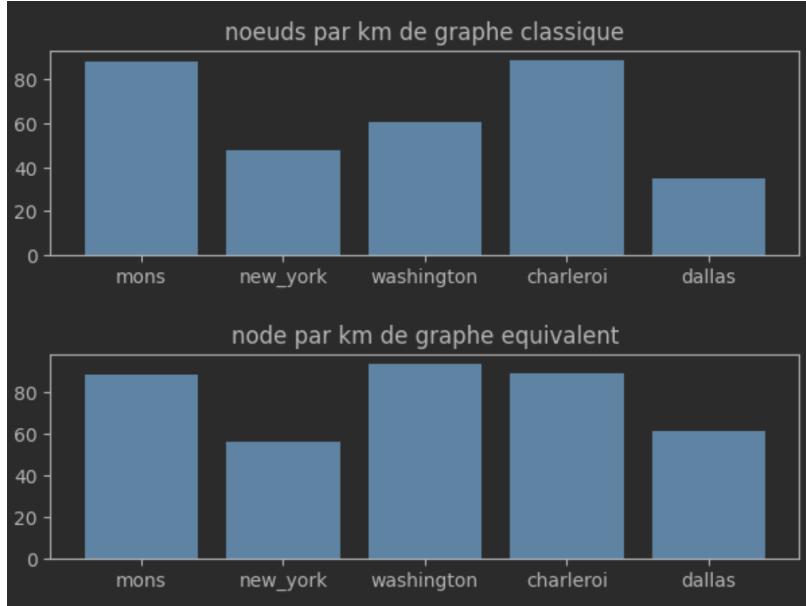


Figure 30: Graphique de la répartition des nœuds par kilomètre carré des différents graphes de villes

3.10 Analyse des arcs

Après avoir analysé les noeuds, j'ai dû à analyser la formation des arcs. [NetworkX](#) utilise des « LineString » pour former ses arcs. Un arc est donc constitué d'un ensemble de segments, ce qui va créer leur courbure. J'ai décidé d'analyser

le nombre de segments moyens constituant un arc. Après cela, j'ai considéré que chaque extrémité de segment peut être considéré comme un nœud de degré 2. Cela m'a donc permis d'obtenir le nombre de nœuds de degré 2 séparant 2 nœuds de degré plus grand que 2. Voici un exemple :

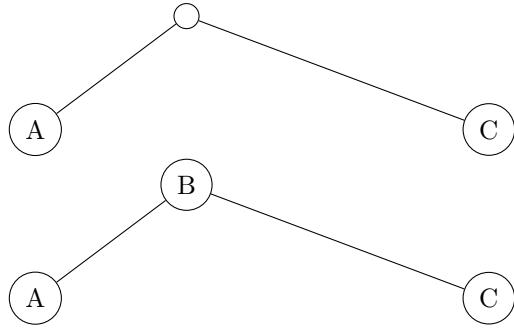


Figure 31: Exemple de formation des arcs avec des nœuds

Élimination des angles trop grands

Malheureusement, au départ mes résultats n'étaient pas très concluant. Cela était dû à des segments ayant des angles très proche de 180. J'ai donc analyser la répartition des nœuds et j'ai éliminé les noeuds de plus de 176 degrés (la valeur moyenne du plus bas quartile de nos différentes villes) et qui représentait plus de 50% des angles.

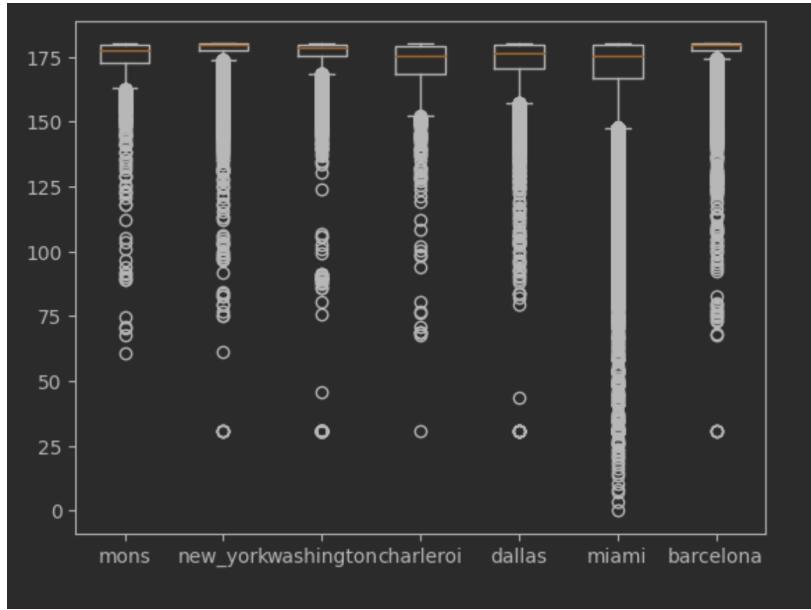


Figure 32: Graphique de la répartition des degrés des angles entre les différents segments formant un arc

Rapport entre le nombre nœuds de degré 2 et le nombre de nœuds de degré plus grand ou égale à 2

Après toutes ces analyses, j'ai obtenu un ratio assez convaincant de 3.64 nœuds de degré 2 séparant 2 nœuds de degré supérieur.

4 Algorithme de génération de graphe planaire

Après avoir défini les caractéristiques de nos graphes, passons à l'implémentation de notre algorithme en respectant ces caractéristiques.

Le code est subdivisé en 3 fonctions :

- `graph_generation`
- `nodes_generation`
- `edges_generation`

4.1 Exemple de fonctionnement de l'algorithme

Les algorithmes étant assez longs, voici un exemple simplifié de la génération d'un graphe. Il s'agit d'une simulation étape par étape de la façon dont les graphes sont générés sur un petit graphe composé de 9 nœuds.

La génération se fait sur un quadrillage, ce qui permet de vérifier facilement la planarité. À la fin de l'algorithme, un bruit aléatoire est ajouté.

Chaque noeud est situé dans un point du quadrillage 3×3 .

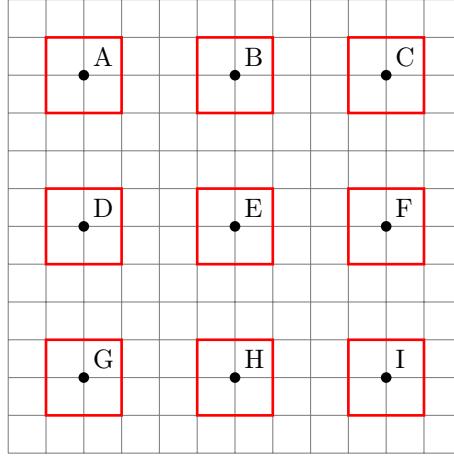


Figure 33: Génération des points sur le quadrillage

Pour chaque noeud, l'algorithme `nodes_generation` va définir son degré. Prenons le cas où le noeud *A* est défini comme un noeud de degré 3. L'algorithme `edges_generation` va choisir 3 noeuds parmi les 3 possibilités en vérifiant la planarité à chaque fois.

Ensuite, si le noeud *B* est défini comme un noeud de degré 2, l'algorithme `edges_generation` va vérifier les arêtes appartenant déjà au noeud *B*. Dans ce cas, l'arête *AB* existe déjà. La fonction va donc ajouter 1 arête à *B*. La possibilité de relier *B* à *D* est supprimée car elle va entraîner la non-planarité du graphe.

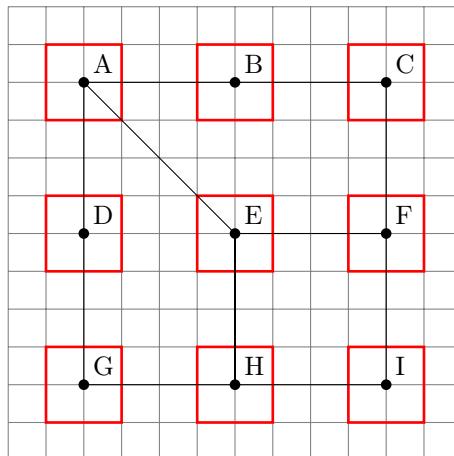


Figure 34: Génération des arcs dans le quadrillage

Ensuite, un bruit compris entre -0.25 et 0.25 est ajouté à chaque nœud pour permettre un emplacement aléatoire tout en maintenant la planarité.

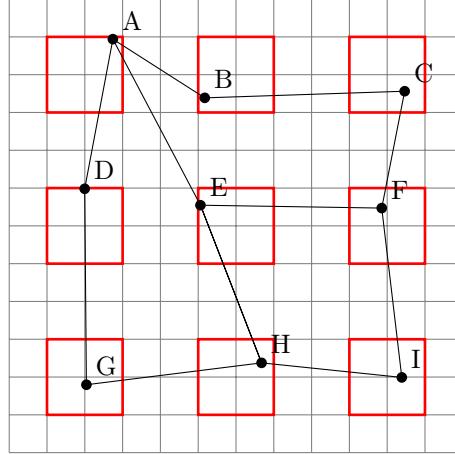


Figure 35: Ajout d'aléatoire sur l'emplacement des nœuds

En effet, si A est en $(0,1)$, A ne peut pas dépasser son carré compris entre $(-0.25, 0.25)$ de largeur et de hauteur. Le point B , situé en $(1,1)$, ne peut pas entrer dans ce carré puisque la valeur du bruit est comprise entre $[-0.25; 0.25[$. Prenons le cas extrême où A est situé en $(0.25, 1.25)$, B en $(0.75, 0.75)$ et C en $(1.25, 0.25)$. Ces 3 points sont impossibles car l'ensemble $[-0.25; 0.25[$ est ouvert mais supposons l'ensemble fermé en tant que cas "extrême". AC passe par B et le graphe n'est donc pas planaire mais étant donné que l'abscisse de B est strictement plus grand que 0.75, AC ne passera jamais par B . De plus, L'arc BD va être interdit et les 2 arcs AB et BC conserveront la planarité.

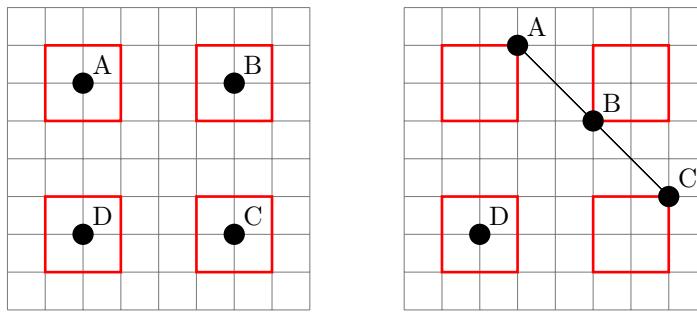


Figure 36: Intersection entre un point et un arc

Enfin, cette génération aléatoire peut entraîner la création de plusieurs composantes connexes différentes. Nous ne gardons que la plus grande composante connexe, ce qui rend le graphe encore plus réaliste puisqu'une ville n'est jamais exactement carrée.

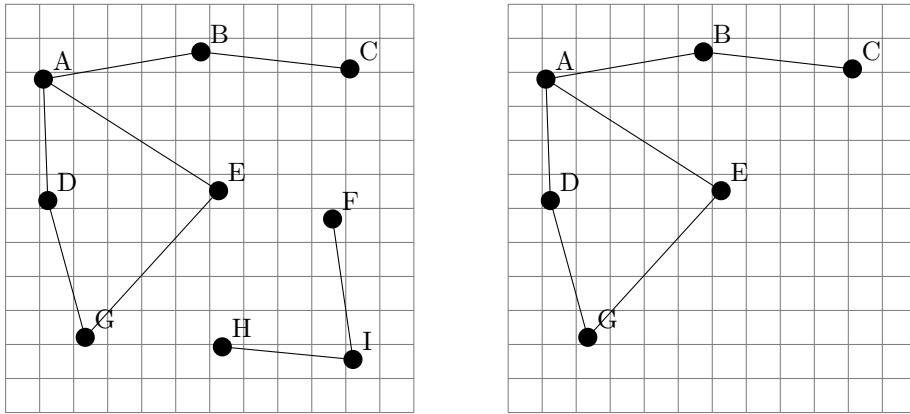


Figure 37: Suppression des composantes connexes inutiles

4.2 Fonction `graph_generation`

4.2.1 Paramètres de `graph_generation`

La fonction `graph_generation` accepte plusieurs paramètres qui contrôlent le processus de génération du graphe :

- `Nbr_node` : le nombre de nœuds dans le graphe (l'ordre de grandeur réel varie en fonction de la génération).
- `Funct_deg` : un tableau de taille 2 contenant les valeurs a et b de l'exponentielle décroissante prédéfinie lors de l'analyse de la répartition des degrés des nœuds.
- `Pourc_2` : le pourcentage de nœuds de degré 2 constituant le graphe (la valeur par défaut est de 65%).
- `Pourc_0` : le pourcentage de nœuds de degré 0 constituant le graphe (la valeur par défaut est de 60%).
- `Noise` : le bruit aléatoire ajouté à la position des nœuds pour permettre une répartition aléatoire tout en maintenant la planarité.
- `Reduce_deg_1` : ce paramètre permet de réduire la quantité de nœuds de degré 1 dans le graphe.
- `Usa` : si vrai, le graphe ressemble à un graphe américain avec moins de bruit et une probabilité plus élevée d'avoir des nœuds avec des arcs verticaux et horizontaux.
- `Color` : ce paramètre permet de différencier les nœuds en fonction de leur degré.

Ces paramètres permettent de contrôler la taille, la structure et l'apparence visuelle du graphe généré.

4.2.2 Nombre de nœuds

Le nombre réel de nœuds dans le graphe généré est calculé à l'aide de la formule suivante :

$$\text{nbr_node} = \frac{\text{nbr_node}}{(1 - \text{pourc_2}) \times (1 - \text{pourc_0})}$$

Cette formule tient compte du fait que les nœuds de degré 2 et 0 sont générés lors du processus de génération avant d'être supprimés.

4.2.3 Matrice d'adjacence Numpy

Une matrice d'adjacence de taille `nbr_node × nbr_node` est utilisée pour stocker les arcs du graphe. Cette matrice est créée et manipulée dans la fonction `nodes_generation`, qui est appelée par `graph_generation`. La matrice d'adjacence permet de représenter les relations entre les nœuds du graphe.

4.2.4 Position des nœuds

La position des nœuds est prédéterminée en fonction de son emplacement dans la matrice d'adjacence. Les nœuds sont placés sur un quadrillage de taille $\sqrt{\text{nbr_node}} \times \sqrt{\text{nbr_node}}$. Pour permettre une répartition aléatoire tout en maintenant la planarité, un bruit aléatoire compris entre -0.25 et 0.25 est ajouté à la position de chaque noeud. Ainsi, chaque nœud se voit attribuer une position légèrement décalée par rapport à la position initiale sur le quadrillage. Cela permet d'obtenir une répartition plus aléatoire des nœuds.

La position des nœuds est stockée dans un dictionnaire appelé `dict_pos`, où la clé est l'identifiant du nœud et la valeur est un tableau contenant les coordonnées (x, y) du nœud dans le plan.

4.2.5 Graphe NetworkX

La fonction `graph_generation` crée également un graphe `NetworkX` correspondant au graphe généré. Le graphe `NetworkX` est construit en ajoutant les nœuds et les arêtes du graphe généré à l'aide des données de la matrice d'adjacence et des positions des nœuds.

4.2.6 Plus grande composante connexe

Après la génération aléatoire du graphe, il est possible que plusieurs composantes connexes [2.2.1](#) distinctes se forment. Dans la fonction `graph_generation`, seule la plus grande composante connexe est conservée. Cela signifie que seuls les nœuds et les arêtes de la plus grande composante connexe sont ajoutés au graphe `NetworkX` final.

Cette étape permet de rendre le graphe plus réaliste, car dans un contexte urbain, par exemple, une ville n'est généralement pas constituée d'ilots complètement isolés, mais plutôt d'une structure connectée.

4.2.7 Suppression des nœuds de degré 2 et 0 pour l'affichage

Avant d'afficher le graphe avec [NetworkX](#), une étape de suppression des nœuds de degré 2 et 0 est effectuée.

La suppression des nœuds de degré 2 et 0 se fait en parcourant l'ensemble des nœuds du graphe et en vérifiant leur degré. Si un nœud a un degré égal à 2 ou à 0, il est caché lors de l'affichage. Ces nœuds ne seront pas représentés graphiquement dans la visualisation finale du graphe.

L'objectif de cette suppression est de simplifier la représentation visuelle du graphe en éliminant les nœuds qui ont peu d'influence sur sa structure globale. Cela permet de mettre davantage l'accent sur les nœuds les plus importants et les connexions significatives du graphe.

Il est important de noter que la suppression des nœuds de degré 2 et 0 pour l'affichage n'affecte pas la structure réelle du graphe lui-même. Elle est uniquement réalisée dans le contexte de la visualisation graphique du graphe à l'aide de la bibliothèque [NetworkX](#).

4.2.8 Complexité en temps

Après la description de chaque étape de l'algorithme, calculons sa complexité en temps étape par étape.

1. La première ligne de la fonction divise `nbr_node` par $(1 - \text{pourc_2}) \times (1 - \text{pourc_0})$. Cela nécessite des opérations de multiplication et d'addition, qui ont une complexité constante. Donc, la complexité de cette étape est $\mathcal{O}(1)$.
2. Le calcul de la variable `side` en prenant la racine carrée de `nbr_node` et en ajoutant 1 nécessite des opérations de racine carrée et d'addition, qui ont également une complexité constante. Donc, la complexité de cette étape est $\mathcal{O}(1)$.
3. La création de `tab_graph` en tant que matrice de nombres aléatoires de taille $[\text{side} \times \text{side}]$ nécessite d'initialiser chaque élément de la matrice avec un nombre aléatoire. Cela prend un temps proportionnel à la taille de la matrice, donc la complexité de cette étape est $\mathcal{O}(\text{side}^2)$.
4. La création de `tab_adj` en tant que matrice de zéros de taille $[\text{nbr_node} \times \text{nbr_node}]$ nécessite d'initialiser chaque élément de la matrice avec zéro. Cela prend un temps proportionnel à la taille de la matrice, donc la complexité de cette étape est $\mathcal{O}(\text{nbr_node}^2)$.
5. L'appel à la fonction `node_generation`, nous donnerons sa complexité dans l'analyse 4.3.3 mais nous supposons sa complexité en $\mathcal{O}(T)$
6. La boucle `for` qui itère sur `nbr_node` et effectue des opérations constantes à chaque itération a une complexité de $\mathcal{O}(\text{nbr_node})$.

7. L'appel à `NetworkX.generate_graph()` dépend de l'implémentation spécifique de cette fonction dans `NetworkX`. Cependant, nous pouvons la borner en $\mathcal{O}(\text{nbr_node}^2)$ car nous lui apportons un matrice d'adjacence de taille $[\text{nbr_node} \times \text{nbr_node}]$.
8. L'appel à `BFS_keep_largest_component(G)` dépend de l'implémentation spécifique de la fonction BFS dans `NetworkX`. Mais nous savons que la complexité du BFS est égale à $\mathcal{O}(|E| + |V|)$, $|E|$ étant le nombre d'arcs et $|V|$ le nombre de noeuds. Le nombre d'arêtes maximal par noeuds étant 8 (une pour chaque direction dans le quadrillage), nous savons que le nombre d'arêtes est borné par `nbr_node * 8` et le nombre de noeuds étant égal à `nbr_node`. La complexité est donc en $\mathcal{O}(9 * \text{nbr_node})$ et donc $\mathcal{O}(\text{nbr_node})$.
9. L'appel à `color_nodes(G, color)` est fait pour chaque itération sur `tab_ajd` ($\mathcal{O}(\text{nbr_node})$), une recherche dans une liste de taille maximale égale à `nbr_node`. La complexité est donc égale à $\mathcal{O}(\text{nbr_node}^2)$.
10. L'appel à `draw_graph(G)` dépend de l'implémentation spécifique de cette fonction dans `NetworkX`. Sa complexité sera donc en $\mathcal{O}(|E| + |V|)$ car l'algorithme dessine chaque noeud et chaque arc les uns après les autres. $\mathcal{O}(|E| + |V|)$ a déjà été simplifié en $\mathcal{O}(\text{nbr_node})$ au point 8.

Voici le résumé corrigé de la complexité en temps de chaque étape de l'algorithme :

- Étape 1 : $\mathcal{O}(1)$
- Étape 2 : $\mathcal{O}(1)$
- Étape 3 : $\mathcal{O}(\text{side}^2)$
- Étape 4 : $\mathcal{O}(\text{nbr_node}^2)$
- Étape 5 : La complexité dépend de l'implémentation spécifique de la fonction `node_generation` donnée ci-dessous 4.3.3. Supposons la complexité égale à $\mathcal{O}(T)$
- Étape 6 : $\mathcal{O}(\text{nbr_node})$
- Étape 7 : $\mathcal{O}(\text{nbr_node}^2)$
- Étape 8 : $\mathcal{O}(\text{nbr_node})$
- Étape 9 : $\mathcal{O}(\text{nbr_node}^2)$
- Étape 10 : $\mathcal{O}(\text{nbr_node})$
- Étape 11 : $\mathcal{O}(\text{nbr_node}^2)$
- Étape 12 : $\mathcal{O}(\text{nbr_node})$

La complexité est donc égale à $\mathcal{O}(\text{side}^2) + (\text{nbr_node}^2) + T + (\text{nbr_node}) + (\text{nbr_node}^2) + (\text{nbr_node}) + \text{nbr_node}^2 + \text{nbr_node} + \text{nbr_node}^2 + \text{nbr_node}$. Sachant que $\text{nbr_node} = \text{side}^2$, on sait que $\text{nbr_node} > \text{side}$. On peut conclure que la complexité est égale à $\mathcal{O}(5 * (\text{nbr_node}^2)) + \mathcal{O}(T)$

Algorithm 1 graph_generation

Require: nbr_node: int, funct_deg: tuple, pourc_2: float, pourc_0: float, noise: float, reduce_deg_1: int, usa: bool, color: bool
Ensure: G: graph

```

1: function GRAPH_GENERATION(nbr_node, funct_deg, pourc_2, pourc_0,
   noise, reduce_deg_1, usa, color)
2:   nbr_node ← nbr_node ÷ ((1 - pourc_2) × (1 - pourc_0))
3:   side ← Integer( $\sqrt{\text{nbr\_node}}$ ) + 1
4:   nbr_node ← side × side
5:   tab_graph ← matrix of random numbers [0, 1] of size [side ×
   side]
6:   tab_adj ← matrix of zeros of size [nbr_node × nbr_node]
7:   NODE_GENERATION(tab_graph, side, funct_deg, pourc_2, pourc_0,
   usa)
8:   dict_pos ← empty dictionary
9:   for i in range(nbr_node) do
10:    pos ← POSITION_FROM_ADJ(i, side)
11:    x ← pos[0] + random number between ] - 0.25; 0.25[
12:    y ← pos[1] + random number between ] - 0.25; 0.25[
13:    dict_pos[i] ← [x, y]
14:   end for
15:   G ← NetworkX.generate_graph()
16:   BFS_KEEP_LARGEST_COMPONENT(G)
17:   COLOR_NODES(G, color)
18:   DRAW_GRAPH(G)
19:   return G
20: end function

```

4.3 Node Generation

4.3.1 Paramètres de node_generation

La fonction node_generation prend les paramètres suivants :

- **tab_graph** : Une matrice représentant les nombres aléatoires entre 0 et 1. Cette matrice est utilisée pour influencer la génération des arêtes.
- **side** : La taille du quadrillage. Il s'agit d'un entier indiquant le nombre de noeuds sur chaque côté du quadrillage.
- **funct_deg** : un tableau de taille 2 contenant les valeurs a et b de l'exponentielle décroissante prédéfinie lors de l'analyse de la répartition des degrés des

nœuds.

- **Pourc_2** : le pourcentage de nœuds de degré 2 constituant le graphe (la valeur par défaut est de 65%).
- **Pourc_0** : le pourcentage de nœuds de degré 0 constituant le graphe (la valeur par défaut est de 60%).
- **Usa** : si vrai, le graphe ressemble à un graphe américain avec moins de bruit et une probabilité plus élevée d'avoir des nœuds avec des arcs verticaux et horizontaux.

Ces paramètres sont utilisés par la fonction `node_generation` pour contrôler la génération des nœuds et des arêtes dans le graphique.

4.3.2 Description

La fonction `node_generation` parcourt chaque noeud et crée deux valeurs aléatoires, `random_2` et `random_0`.

Pour chaque noeud, si `random_2` est inférieur à `pourc_2`, cela signifie que le noeud aura deux arêtes. Dans ce cas, la fonction `edges_generation` est appelée avec les paramètres appropriés pour créer les deux arêtes.

Si `random_2` n'est pas inférieur à `pourc_2`, la génération d'arêtes est basée sur la valeur de `random_0`. Si `random_0` est inférieur à `pourc_0`, cela indique que le noeud ne génère aucune arête. Dans ce cas, le traitement du noeud se termine et passe au noeud suivant.

Si `random_0` n'est pas inférieur à `pourc_0`, cela signifie que le noeud doit générer un nombre d'arêtes correspondant à sa valeur dans le `tab_graph`. La fonction `edges_generation` est appelée avec les paramètres appropriés pour créer ces arêtes.

En utilisant ces mécanismes, la fonction `node_generation` génère les arêtes des nœuds du graphe en fonction des différents paramètres.

4.3.3 Complexité

- Étape 1 : La boucle extérieure parcourt chaque ligne de la matrice `tab_graph`, ce qui nécessite un temps proportionnel à `side`. La boucle intérieure parcourt chaque colonne de la matrice, également en temps proportionnel à `side`. Donc, la complexité totale de cette étape est de $\mathcal{O}(\text{side}^2)$.
- Étape 2 : La génération d'un nombre aléatoire `random_2` nécessite une opération de génération de nombre aléatoire, ce qui a une complexité constante. Donc, la complexité de cette étape est $\mathcal{O}(1)$.
- Étape 3 : La condition `random_2 < pourc_2` est évaluée en temps constant, car il s'agit simplement d'une comparaison. Donc, la complexité de cette étape est $\mathcal{O}(1)$.

- Étape 4 : Si la condition précédente est vraie, alors l'appel à la fonction `edges_generation` est effectué. Sa complexité dans l'analyse 4.4.2 mais nous supposons sa complexité en $O(U)$
- Étape 5 : Si la condition précédente est fausse, alors une nouvelle variable aléatoire `random_0` est générée, ce qui a une complexité constante. Donc, la complexité de cette étape est $\mathcal{O}(1)$.
- Étape 6 : La condition `random_0 < pourc_0` est évaluée en temps constant, car il s'agit simplement d'une comparaison. Donc, la complexité de cette étape est $\mathcal{O}(1)$.
- Étape 7 : Si la condition précédente est vraie, alors l'instruction `continue` est exécutée, ce qui a une complexité constante. Donc, la complexité de cette étape est $\mathcal{O}(1)$.
- Étape 8 : Si la condition précédente est fausse, alors une nouvelle variable aléatoire `nbr_edge` est générée, ce qui a une complexité constante. Donc, la complexité de cette étape est $\mathcal{O}(1)$.
- Étape 9 : L'appel à la fonction `edges_generation` est effectué. Supposons sa complexité en $\mathcal{O}(U)$

En résumé, la complexité en temps de chaque étape de l'algorithme `node_generation` est la suivante :

- Étape 1 : $\mathcal{O}(\text{side}^2)$
- Étapes 2-3 : $\mathcal{O}(1)$
- Étapes 4 : $\mathcal{O}(U)$
- Étapes 5-7 : $\mathcal{O}(1)$
- Étapes 8 : $\mathcal{O}(1)$
- Étapes 9 : $\mathcal{O}(U)$

La complexité est donc égale à $\mathcal{O}(\text{side}^2) * \mathcal{O}(U)$

Algorithm 2 node_generation

Require: tab_graph : matrix, $side$: int, $funct_deg$: tuple, $pourc_2$: float,
 $pourc_0$: float, usa : bool

Ensure: modify Global tab_adj

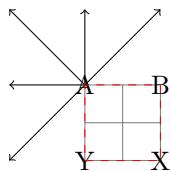
```
1: function NODE_GENERATION( $tab\_graph$ ,  $side$ ,  $funct\_deg$ ,  $pourc\_2$ ,  
     $pourc\_0$ ,  $usa$ )
2:   for lenght in range( $side$ ) do
3:     for width in range( $side$ ) do
4:       random_2  $\leftarrow$  generate random number between 0 and 1
5:       if  $random\_2 < pourc\_2$  then
6:         EDGES_GENERATION( $i$ ,  $j$ ,  $side$ ,  $nbr\_edge = 2$ ,  $usa = False$ )
7:       else
8:         random_0  $\leftarrow$  generate random number between 0 and 1
9:         if  $random\_0 < pourc\_0$  then
10:           continue
11:         else
12:            $nbr\_edge \leftarrow$  Call inverse( $tab\_graph[lenght][width]$ )
13:           EDGES_GENERATION( $i$ ,  $j$ ,  $side$ ,  $nbr\_edge$ ,  $usa$ )
14:         end if
15:       end if
16:     end for
17:   end for
18: end function
```

4.4 Edges Generation

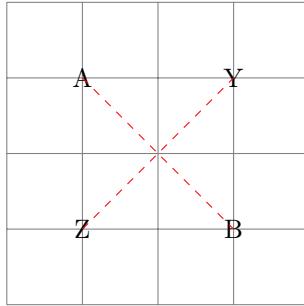
4.4.1 Positions possibles des arêtes

L'algorithme commence par déterminer les positions possibles pour les arêtes à partir de la position du noeud dans la matrice. Ces positions sont stockées dans un dictionnaire avec des clés représentant les directions possibles (par exemple, "up-left", "up", "up-right", etc.). Certaines positions peuvent être exclues en fonction de la taille de la grille et de la planarité du graphe.

Par exemple, A étant dans un coin du quadrillage (0,0), il ne peut pas être relié hors du quadrillage.



Le second exemple montre deux noeuds, A et B, avec une arête AB, ainsi qu'une autre arête YZ qui traverse l'arête AB. La condition de planarité indique que l'arête AB ne peut pas être présente si une arête comme YZ existe déjà.



4.4.2 Complexité

La complexité de l'algorithme `edges_generation` peut être analysée de la manière suivante :

- L'initialisation du dictionnaire `possible_position` et de l'ensemble `poss_pos` prend un temps constant, c'est-à-dire $\mathcal{O}(1)$.
- L'élimination des positions hors de la grille implique de vérifier les limites de la grille. Cette étape prend un temps constant, c'est-à-dire $\mathcal{O}(1)$.
- L'élimination des cas de non-planarité implique également de vérifier certaines conditions basées sur les positions. Cette étape prend un temps constant, c'est-à-dire $\mathcal{O}(1)$.
- Si le paramètre `usa` est vrai, l'algorithme augmente la probabilité des arêtes verticales et horizontales. Cette opération prend un temps constant, c'est-à-dire $\mathcal{O}(1)$.
- La boucle de 1 à `nbr_edges` itère `nbr_edges` fois. À l'intérieur de la boucle, l'ajout d'un arc dans une des positions possibles dans `tab_adj` prend un temps constant, c'est-à-dire $\mathcal{O}(1)$. Par conséquent, la complexité globale de cette boucle est $\mathcal{O}(\text{nbr_edges})$, où `nbr_edges` est le nombre d'arêtes.

Ainsi, la complexité globale de l'algorithme `edges_generation` est $\mathcal{O}(\text{nbr_edges})$.

Algorithm 3 edges_generation

Require: i : int, j : int, $side$: int, nbr_edges : int, usa : bool
Ensure: modify Global tab_adj

```
1: function EDGES_GENERATION( $i, j, side, nbr\_edges, usa$ )
2:    $possible\_position \leftarrow$  dictionary of {"up-left": False, "up": False, "up-right": False, "right": get_right( $i, j$ ), "down-right": get_down_right( $i, j$ ), "down": get_down( $i, j$ ), "down-left": get_down_left( $i, j$ ), "left": get_left( $i, j$ )}
3:    $poss\_pos \leftarrow$  set of the possible keys
4:   Elimination of the positions out of the grid
5:   Elimination of cases of non-planarity
6:   if  $usa = True$  then
7:     Increase probability of vertical and horizontal edges
8:   end if
9:    $nbr\_edges \leftarrow nbr\_edges -$  number of edges already present in the ma-
    trix of adjacency for the node
10:  for  $i$  in range( $nbr\_edges$ ) do
11:     $tab\_adj \leftarrow$  adding 1 in one of the possible positions
12:  end for
13: end function
```

5 Autres fonctions utilisées dans le projet

5.1 Fonction fct

Algorithm 4 fct

```
1: function FCT( $x, a, b$ )
2:   Input:  $x$  (variable d'entrée),  $a$  (paramètre  $a$  de la fonction  $fct$ ),  $b$ 
      (paramètre  $b$  de la fonction  $fct$ )
3:   Output:  $y$  (résultat de la fonction  $fct$ )
4:
5:    $y \leftarrow a \cdot e^{-b \cdot x}$ 
6:   return  $y$ 
7: end function
```

La fonction **fct** est une fonction exponentielle utilisée pour ajuster la distribution des degrés du graphe. Elle prend trois paramètres en entrée : x pour la variable d'entrée, a le paramètre a de la fonction **fct**, et b le paramètre b de la fonction **fct**.

La fonction **fct** est définie par l'équation $y = a \cdot e^{-b \cdot x}$, où e représente le nombre d'Euler.

La fonction **fct** renvoie la valeur y calculée, qui correspond au résultat de la fonction exponentielle pour la valeur donnée de x .

5.2 Fonction cumulative

Algorithm 5 cumulative

```
1: function CUMULATIVE( $x, a, b$ )
2:   Input:  $x$  (variable d'entrée),  $a$  (paramètre  $a$  de la fonction  $fct$ ),  $b$ 
      (paramètre  $b$  de la fonction  $fct$ )
3:   Output:  $y$  (résultat de la fonction cumulative)
4:
5:    $y \leftarrow 1 - fct(x, a, b)$ 
6:   return  $y$ 
7: end function
```

La fonction **cumulative** est la fonction cumulative de la fonction exponentielle **fct**. Elle prend trois paramètres en entrée : x pour la variable d'entrée, a le paramètre a de la fonction **fct**, et b le paramètre b de la fonction **fct**.

La fonction cumulative est définie par l'équation $y = 1 - fct(x, a, b)$, où **fct** est la fonction exponentielle précédemment.

La fonction **cumulative** renvoie la valeur y calculée, qui correspond au résultat de la fonction cumulative pour la valeur donnée de x .

5.3 Fonction inverse

Algorithm 6 inverse

```
1: function INVERSE( $r, a, b$ )
2:   Input:  $r$  (valeur pour laquelle calculer l'inverse),  $a$  (paramètre  $a$  de la
      fonction  $fct$ ),  $b$  (paramètre  $b$  de la fonction  $fct$ )
3:   Output:  $x$  (résultat de l'inverse)
4:
5:    $x \leftarrow \lfloor -\log\left(\frac{-(r-1)}{a}\right) / b \rfloor$ 
6:   return  $x$ 
7: end function
```

La fonction **inverse** calcule l'inverse de la fonction cumulative de la fonction exponentielle **fct**. Elle prend trois paramètres en entrée : r pour la valeur pour laquelle calculer l'inverse, a le paramètre a de la fonction exponentielle **fct**, et b le paramètre b de la fonction exponentielle **fct**.

L'inverse est calculé en utilisant la formule : $x = \lfloor -\log\left(\frac{-(r-1)}{a}\right) / b \rfloor$, où \log représente le logarithme népérien et $\lfloor \cdot \rfloor$ représente la fonction plancher.

L'objectif de cette fonction est de trouver la valeur de x telle que $1 - \text{cumul}(x, a, b) = r$, où **cumul** est la fonction cumulative de **fct**.

La fonction **inverse** renvoie la valeur x calculée, qui correspond à la valeur recherchée dans la distribution des degrés du graphe.

5.4 Fonction position_in_adj

Algorithm 7 position_in_adj

```
1: function POSITION_IN_ADJ(pos, side)
2:   Input: pos (position du noeud dans le graphe), side (taille du côté du
      graphe)
3:   Output: position (position du noeud dans la matrice d'adjacence)
4:
5:   position  $\leftarrow$  pos[0]  $\cdot$  side + pos[1]
6:   return position
7: end function
```

La fonction `position_in_adj` permet de donner la position du noeud dans la matrice d'adjacence en fonction de sa position dans le graphe. Elle prend deux paramètres en entrée : *pos* pour la position du noeud dans le graphe et *side* pour la taille du côté du graphe.

La position du noeud dans la matrice d'adjacence est calculée en utilisant la formule $position = pos[0] \cdot side + pos[1]$.

La fonction `position_in_adj` renvoie la valeur *position* calculée, qui correspond à la position du noeud dans la matrice d'adjacence.

5.5 Fonction position_from_adj

Algorithm 8 position_from_adj

```
1: function POSITION_FROM_ADJ(pos, side)
2:   Entrée: pos (position du noeud dans la matrice d'adjacence), side (taille
      du côté du graphe)
3:   Sortie: position (position du noeud dans le graphe)
4:
5:   i  $\leftarrow$  (side - 1) -  $\lfloor \frac{pos}{side} \rfloor$ 
6:   j  $\leftarrow$  pos%side
7:   return [i, j]
8: end function
```

La fonction `position_from_adj` permet de donner la position du noeud dans le graphe en fonction de sa position dans la matrice d'adjacence. Elle prend deux paramètres en entrée : *pos* pour la position du noeud dans la matrice d'adjacence et *side* pour la taille du côté du graphe.

La position du noeud dans le graphe est calculée en utilisant la formule $i = (side - 1) - \lfloor \frac{pos}{side} \rfloor$ et $j = pos \% side$.

La fonction `position_from_adj` renvoie la valeur *position* calculée, qui correspond à la position du noeud dans le graphe.

6 Complexité globale de l'algorithme de génération de graphe.

Après avoir analysé la complexité de chaque algorithme, nous pouvons conclure que la complexité de `edges_generation` est de l'ordre de $\mathcal{O}(\text{nbr_edges})$. Cela implique que la complexité de `node_generation` est de l'ordre de $\mathcal{O}(\text{side}^2 \times \text{nbr_edges})$. Nous savons que $\text{side}^2 = \text{nbr_node}$ et que `nbr_edges` est borné par $8 \times \text{nbr_node}$. Par conséquent, la complexité de `node_generation` peut être exprimée comme $\mathcal{O}(\text{nbr_node} + 8 \times \text{nbr_node}) = \mathcal{O}(\text{nbr_node})$. En résumé, l'algorithme `graph_generation` a une complexité temporelle de $\mathcal{O}(5 \times \text{nbr_node}^2) + \mathcal{O}(\text{nbr_node})$. Cette complexité peut être simplifiée à $\mathcal{O}(\text{nbr_node}^2)$.

La complexité dépend donc du nombre de noeuds choisi, c'est-à-dire de `nbr_node`, ainsi que des paramètres `pourc_0` et `pourc_2`, qui influent sur le nombre de noeuds réellement généré par l'algorithme. La formule qui influe sur la complexité est la suite :

$$\text{nbr_node} = \left(\sqrt{\frac{\text{nbr_node}}{(1 - \text{pourc_0}) \cdot (1 - \text{pourc_2})}} + 1 \right)^2$$

Par conséquent, la complexité finale reste dominée par le terme quadratique $\mathcal{O}(\text{nbr_node}^2)$.

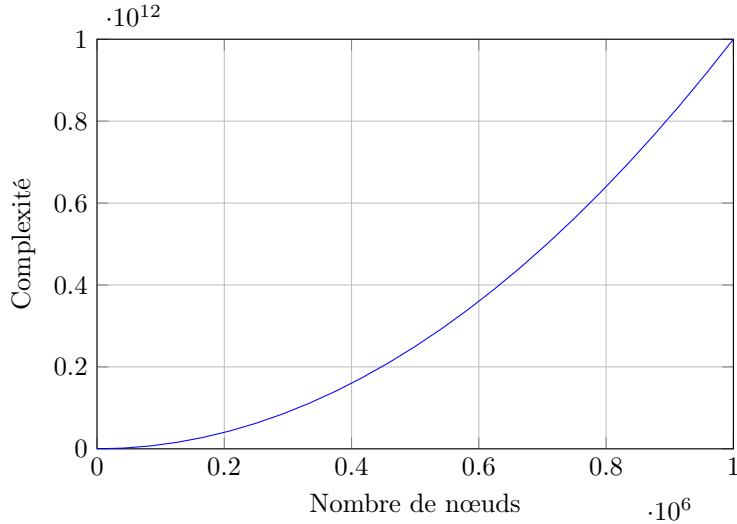


Figure 38: Complexité de l'algorithme en fonction du nombre de noeuds

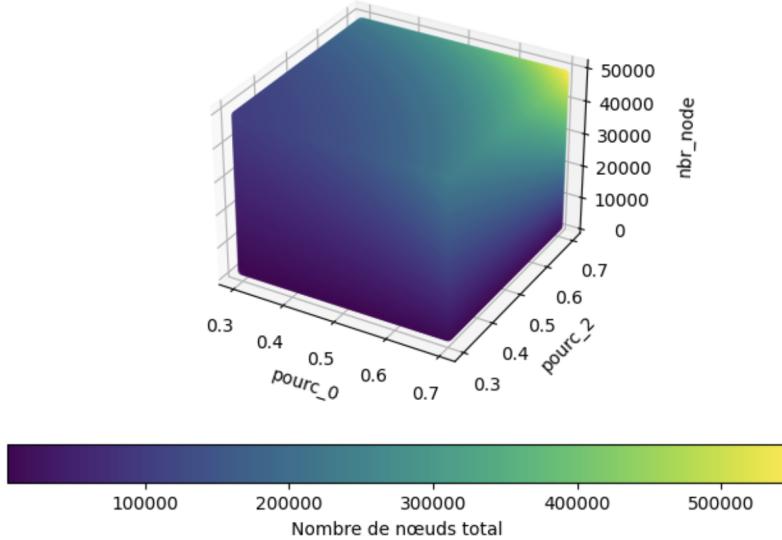


Figure 39: évolution du nombre de nœuds total en fonction de `nbr_node`, `pourc_0` et `pourc_2`

7 Résultats

Le fichier principal de ce projet est appelé `graph_generation_np_array`. Il fonctionne comme une librairie, il suffit de l'importer pour l'utiliser. Cependant, j'ai fait un fichier `main.py` s'occupant d'afficher les étapes de génération ainsi que de tracer un graphique. Les paramètres ayant déjà été expliqués à la section 4.2.1, cette section va montrer les graphes que nous obtenons en utilisant cette librairie. Je vais également expliquer les défauts de celle-ci.

7.1 Cas standard

Le cas standard (c'est-à-dire un appel à la fonction sans modification de paramètre) génère un graphe de taille ≈ 4000 nœuds. Ce nombre varie en fonction des générations. Dans cet exemple 40, nous avons un graphe de 3674 nœuds. Vous pouvez voir sur la figure 41 que la répartition des nœuds est très proche de la fonction que nous avons définie sur la figure 14 mais pas exacte. Cela est dû à l'aléatoire ajouté à certain endroit de l'algorithme.

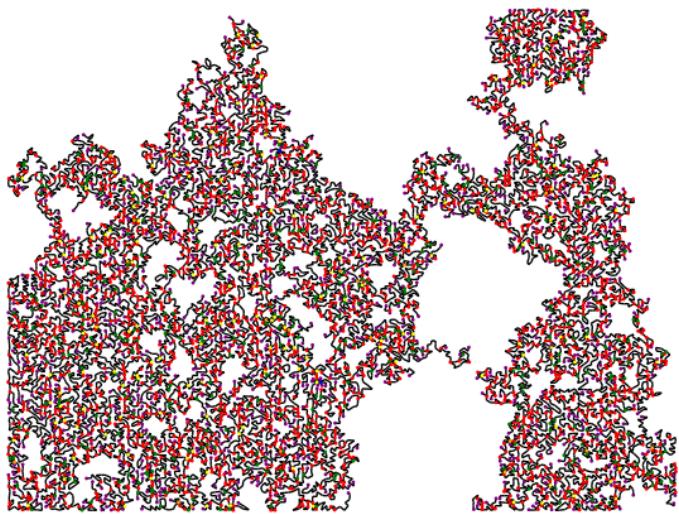


Figure 40: Cas standard de génération de graph

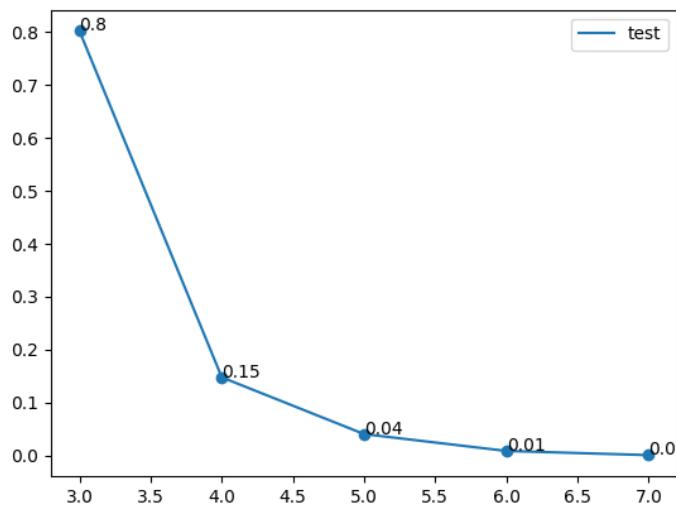


Figure 41: Graphique de la répartitions des noeuds pour le cas standard

7.2 Nombre faible de nœuds

Dans l'exemple suivant, nous allons voir un cas où le nombre de nœuds est faible. Etant donné que la construction est faite sur un quadrillage, un nombre faible de nœuds montre un géométrie très rectiligne. Pour ce genre de petit graphe, il serait assez intéressant d'augmenter le nombre de sommets de degré 2.

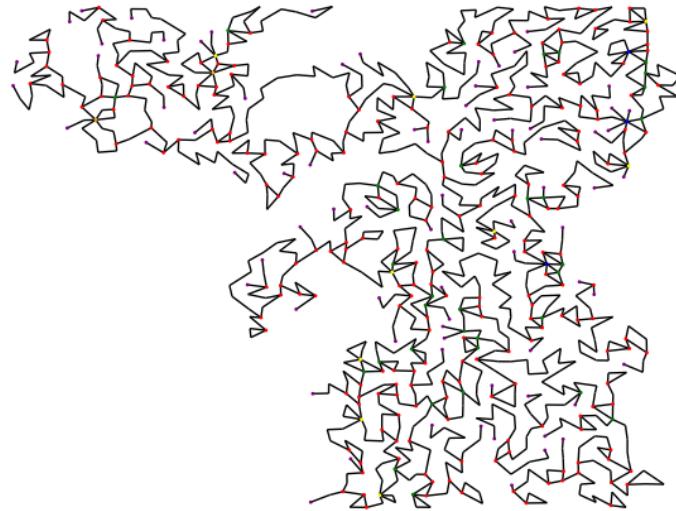


Figure 42: Nombre faible de noeuds

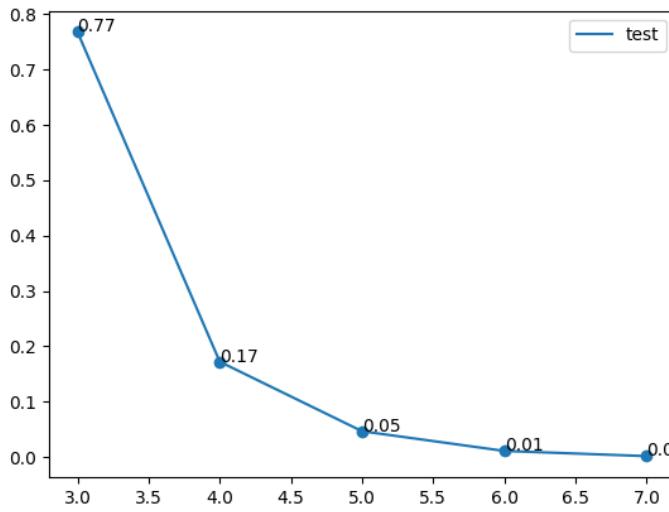


Figure 43: Graphique de la répartition des nœuds pour le nombre faible de nœuds

7.3 Grand nombre de nœuds

Dans l'exemple suivant, nous allons voir un cas où le nombre de nœuds est très grand. La génération ayant un complexité en temps en $\mathcal{O}(n^2)$, la génération serait proportionnelle à la taille du graphe. Cependant, un problème lié à de nombreux facteurs comme le langage ([Python](#)), la librairie ([NetworkX](#)), ... entraîne une explosion du stockage en mémoire. La section 8 présente différentes hypothèses pour résoudre ce problème.

Le plus grand graphe que j'ai su réaliser est un graphe composé de 27727 nœuds, ce graphe a nécessité 260 Go de mémoire.

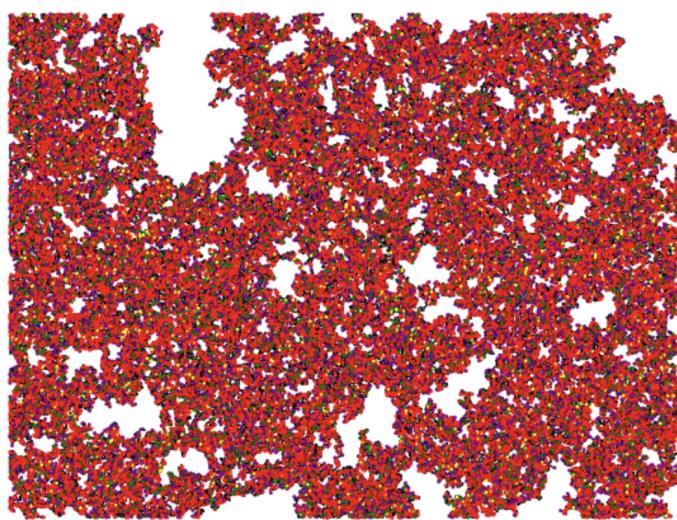


Figure 44: Grand nombre de noeuds

7.4 Paramètre usa

Les graphes ayant le paramètre usa, sont très rectilignes et ont énormément d'arcs verticaux et horizontaux.

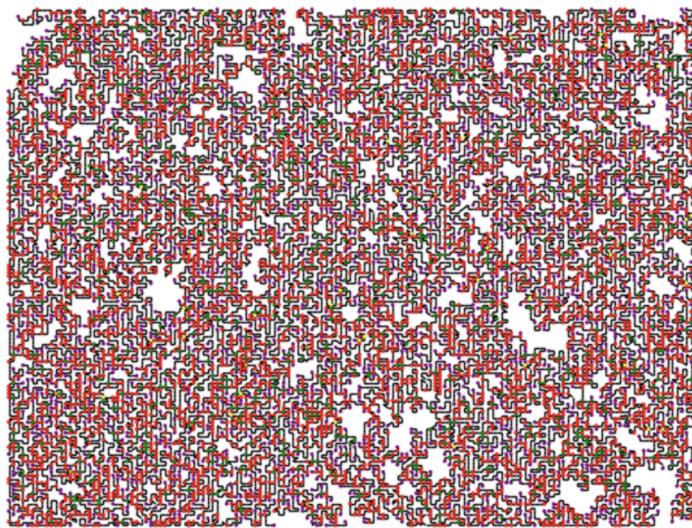


Figure 45: Utilisation paramètre usa

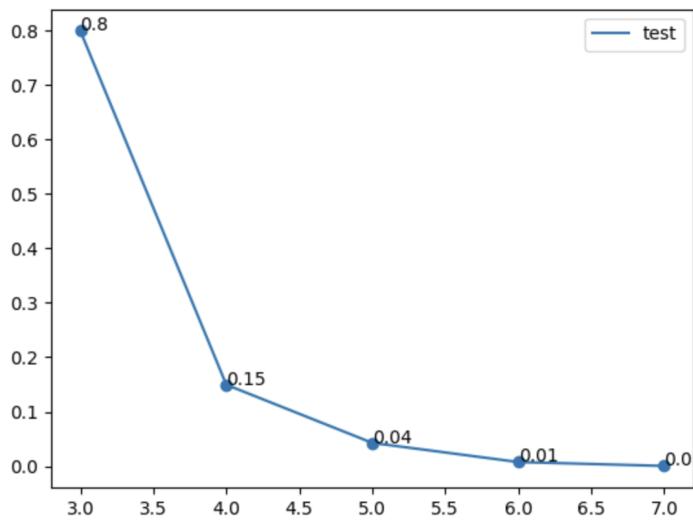


Figure 46: Graphique de la répartition des noeuds avec le paramètre usa

7.5 Paramètre `color`

Les graphes générés en ayant mis le paramètre `color` à `False`, permettent de comparer visuellement ceux-ci à des villes réelles.

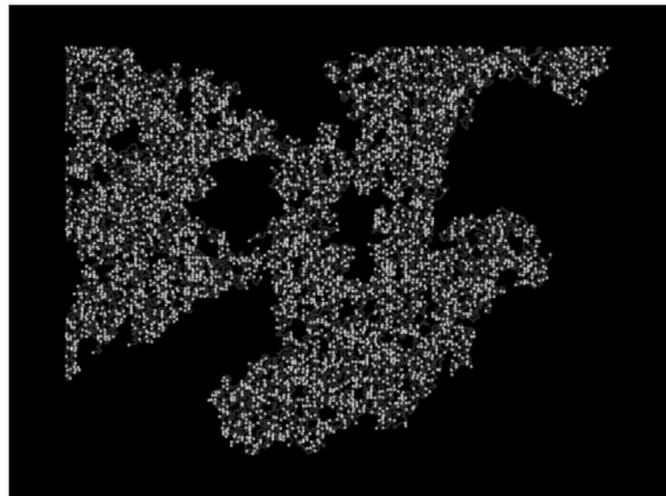


Figure 47: Utilisation paramètre `color`

7.6 Comparaison

On peut voir que le graphe généré aléatoirement correspond très fortement à nos graphes de villes analysés précédemment. Le nombre de noeuds de degrés 1 peut être réduit grâce au paramètre `reduce_deg_1` mais ajoute une fois de plus une complexité en $\mathcal{O}(n^2)$ à l'algorithme.

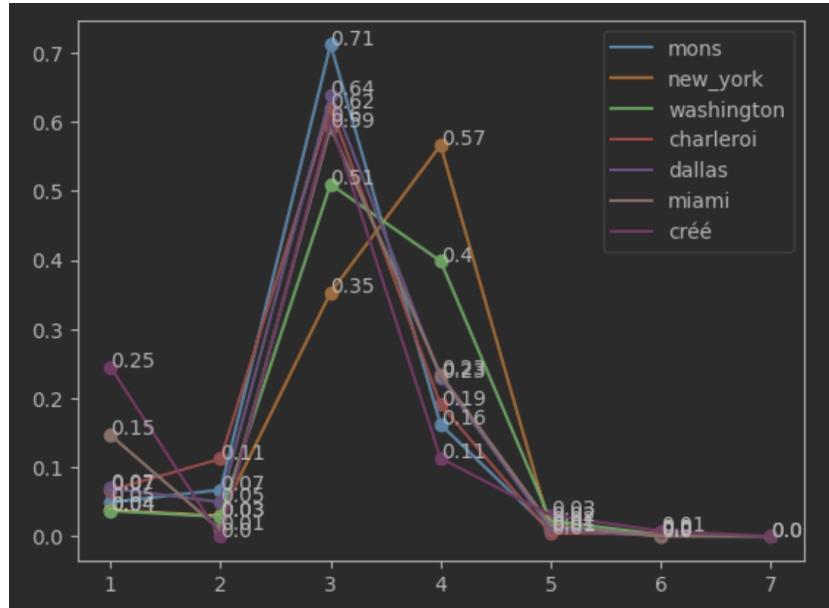


Figure 48: Graphique comparant les villes analysées à un graphe créé

8 Perspectives futures

Le projet pourrait être amélioré sur divers points que je vais aborder dans cette section. De plus, il pourrait être étendu à d'autres emplois mais également utilisé pour tester d'autres projets.

8.1 Complexité en espace

Cette amélioration est la plus importante si nous voulons étendre ce projet sur la génération de pays qui est abordé dans la section 8.4. Les sous-points suivants sont les hypothèses que j'ai concernant cette explosion de la complexité en espace.⁴

8.1.1 Langage

Python étant un langage de très haut niveau, il ne permet pas de gérer chaque allocation mémoire. Il est donc possible que malgré l'utilisation de variables globales, Python ajoute plusieurs fois la matrice d'adjacence en mémoire. La matrice étant de taille très importante (200000x200000) dans les cas limites, le stockage multiple de celle-ci pourrait entraîner cette utilisation de mémoire intensive.

8.1.2 Librairie

[NetworkX](#) est une librairie externe et elle génère un nombre assez important de données lors de la génération d'un graphe, il est donc également possible que cette librairie entraîne en partie cette surutilisation de la mémoire.

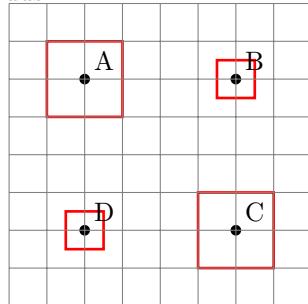
8.1.3 Utilisation de matrice creuse

La matrice d'adjacence étant une matrice remplie en grande partie de 0, elle pourrait être simplifiée en matrice creuse (sparse matrix en anglais) ce qui pourrait diminuer sa place en mémoire. Je ne l'ai pas réalisé car [NetworkX](#) a besoin d'une matrice d'adjacence classique en entrée et donc une transformation de la matrice creuse en matrice d'adjacence était nécessaire. Puisque [NetworkX](#) est aussi une cause de cette utilisation assez importante de la mémoire, cette matrice creuse n'aurait pas entièrement résolu le problème d'espace.

8.2 Génération géométrique

Cette partie est plus liée au problème des graphes de petites tailles de la section [7.2](#). Il serait peut-être intéressant de créer un algorithme plus avancé pour le placement des noeuds. pour ajouter une taille aléatoire de carré limite, ...

Prenons un exemple [8.2](#), Nous avons les point A , B , C , D sur le même quadrillage qu'au point [4.1](#) mais les bruits aléatoires sont différents entre les noeuds.



Il existe beaucoup de possibilités à explorer pour améliorer l'apparence des graphes. Néanmoins, le projet étant utilisé pour faire des test d'algorithmes, l'apparence visuelle n'influe pas ceux-ci.

8.3 Parcours en largeur avant [NetworkX](#)

Le parcours en largeur du graphe (BFS en anglais) nécessaire à la suppression des petites composantes connexe pourrait être réalisé avant l'appel à [NetworkX](#), cela permettrait de diminuer le nombre de noeuds donné à [NetworkX](#) et ainsi diminuer la complexité en espace de l'algorithme en général. Cependant, je ne l'ai pas réalisé car [NetworkX](#) utilise un algorithme très rapide de parcours en largeur de graphe ce qui améliorait significativement le temps de génération de nos graphes.

8.4 Génération d'un pays

Cette perspective nécessiterait beaucoup d'ajustement technique. En effet, un pays possède en général beaucoup de zones vides avec quelque zones très concentrées reliées les unes aux autres par 1 ou 2 grandes route principale. Il faudrait adapter cela pour que l'algorithme décide de terminer un zone "urbaine" et d'entrer dans une zone "rurale". De plus, la taille du graphe d'un pays est bien plus important, il faudrait donc améliorer la complexité en espace de nos algorithmes pour les créer.

9 Conclusion

Notre analyse approfondie a porté principalement sur les graphes de cinq villes : Mons, New York, Washington, Charleroi et Dallas. Malgré les différences visuelles entre ces villes situées en Europe et en Amérique du Nord, nos études ont révélé des caractéristiques et des invariants similaires dans leurs graphes urbains. Cette constatation soulève l'intérêt d'étendre nos analyses à d'autres types d'organisations urbaines pour vérifier si ces caractéristiques restent cohérentes. Si des différences significatives sont observées, il serait alors envisageable de modifier les paramètres de notre algorithme de génération de graphes afin de créer des modèles adaptés à ces nouvelles configurations urbaines.

L'élaboration de notre algorithme, qui génère des graphes urbains à partir de noeuds disposés dans une grille, nous a également poussé à réfléchir à l'incorporation de bruit au niveau de l'emplacement des noeuds tout en préservant la planarité. L'amélioration de ce processus de bruitage pourrait rendre les graphes encore plus uniques et aléatoires, ajoutant ainsi une dimension de réalisme supplémentaire à nos modèles.

Il est important de souligner que l'analyse des graphes est un domaine de recherche en perpétuelle évolution, et de nouveaux invariants non traités dans ce rapport pourraient révéler des différences entre les graphes analysés. Ce projet constitue donc une ouverture vers d'autres perspectives de recherche, non seulement dans le domaine des graphes urbains, mais également dans l'analyse de graphes à plus grande échelle, tels que ceux représentant des pays ou des continents. Ces analyses pourraient nous aider à mieux comprendre les structures complexes qui sous-tendent nos environnements urbains.

En considérant l'efficacité de nos algorithmes, une complexité en $\mathcal{O}(\text{nbr_node}^2)$ pour la génération de grands graphes reste acceptable dans des délais raisonnables, étant donné que la limitation principale réside dans la complexité spatiale. Cependant, il serait intéressant d'explorer les différentes hypothèses émises dans la section 8 afin de résoudre cette complexité en espace et ainsi permettre la génération efficace de graphes urbains de plus grande envergure.

Une piste d'extension serait d'analyser les réseaux de transport ferroviaire pour déterminer si, une fois encore, les caractéristiques des réseaux de différents endroits du monde présentent des similitudes. Cette analyse permettrait d'envisager des améliorations et des optimisations pour un transport public plus efficace et adapté aux besoins des populations.

Enfin, les graphes que nous avons obtenus pourraient être exploités pour appliquer divers algorithmes de simulation de la vie urbaine. Ces simulations pourraient nous aider à évaluer les effets de différentes caractéristiques de graphes sur l'avenir de nos environnements urbains en termes de population, d'efficacité énergétique et d'accessibilité aux services. Par exemple, en modélisant la croissance démographique et les déplacements des habitants dans un graphe urbain donné, nous pourrions estimer l'impact de l'ajout de nouvelles infrastructures de transports, tels que des lignes de métro supplémentaires ou des pistes cyclables, sur la congestion routière et la durée des trajets quotidiens.

En conclusion, notre projet de recherche sur l'analyse des graphes urbains

a permis de mettre en évidence des caractéristiques et des invariants communs dans les villes étudiées. Cependant, de nombreuses perspectives de recherche restent à explorer, notamment en analysant d'autres types d'organisations urbaines et en tenant compte de l'impact des caractéristiques géographiques sur la structure des graphes urbains. De plus, l'utilisation des graphes générés pour appliquer des algorithmes de simulation de la vie urbaine ouvre de nouvelles possibilités pour évaluer l'efficacité des infrastructures urbaines et prendre des décisions en matière de développement urbain. Ces travaux contribuent ainsi à une meilleure compréhension des villes et offrent des opportunités à l'amélioration de nos villes futures.

References

- [1] V. Bruyère. *Calculabilité et complexité*, chapter 5 : complexité. 2022.
- [2] Reinhard Diestel. *Graph Theory*. Springer, 2017.
- [3] ERIC FUSY. Uniform random sampling of planar graphs in linear time. 2008.
- [4] H. Mélot. *Graphs & Artificial Intelligence*. 2022.
- [5] NetworkX Developers. Networkx documentation: average_degree_connectivity. February 2023.
- [6] C. Samain. Comparaison d'heuristique pour l'optimisation de l'emplacement des arrêts de transport en commun, 2023.
- [7] W. T. Tutte. A theorem on planar graphs. *American Mathematical Society*, 1956.
- [8] Wikipédia. Plan hippodamien, April 2023.