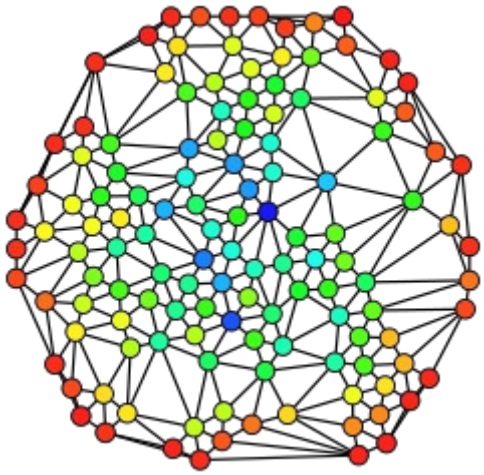
Université Claude Bernard Lyon 1

Mesurer la centralité de proximité dans un graphe de De Bruijn

Cours de Gustavo Sacomoto



Pauline Auffret – Anaïs Marino – Kevin Parent

M1 Ecosciences, Microbiologie – option MIV

2012/2013

Introduction

Le problème des sept ponts de la ville de Königsberg, qui consiste à rechercher un chemin passant une et une seule fois par chaque pont en revenant à son point de départ, fut résolu en 1736 par Euler qui énonça qu'il n'existait en fait pas de solution. Ce problème donna naissance à la théorie des graphes, qui permet de modéliser des réseaux par un ensemble de nœuds et d'arêtes, orientées ou non . Cette théorie possède de nombreuses applications dans des domaines divers utilisant la notion de réseaux. En particulier, la théorie des graphes a été appliquée par Nicolaas De Bruijn : ce mathématicien néerlandais a défini des graphes orientés *B(k,n)* qui permettent de représenter les chevauchements de longueur *n-1* entre tous les mots de longueur *n* sur un alphabet donné de *k* lettres. Les graphes de De Bruijn sont notamment utilisés en génomique dans le cadre de l'assemblage des lectures issues des séquençages haut débit. D'autre part, la représentation de réseaux (réseaux sociaux, réseaux métaboliques...) sous forme de graphe permet d'associer à chaque nœud des mesures telles que degré entrant, degré sortant ou centralité : la centralité permet de classer les nœuds selon leur position dans le réseau, et elle est interprétée par l'importance du nœud dans le réseau.

L'objectif de ce projet est de développer un programme permettant de calculer la centralité de chaque nœud dans un graphe de De Bruijn formé à partir de lectures issues d'un séquençage, et de voir si cela met en évidence des motifs pertinents. La mise en œuvre de ce travail nécessitera l'utilisation d'un programme préexistant transformant une liste de lectures au format fasta en un graphe de De Bruijn, ainsi que l'implémentation d'n algorithme de recherche du plus court chemin. Plusieurs algorithmes ont été décrits dans la littérature pour résoudre ce problème, dont l'algorithme de Dijkstra[1] qui sera utilisé dans ce projet et implémenté dans le langage de programmation informatique Python. Enfin, un calcul de centralité sera appliqué à chaque nœud et le graphe sera représenté en associant une couleur proportionnelle à la centralité de chaque nœud via l'utilisation d'un module Python.

I. Matériels et méthodes

I.1. Lectures et graphe de De Bruijn

Le fichier de départ contient des lectures, issues d'un séquençage haut débit, au format fasta (son extension est *.fa*). Ce format impose, pour chaque lecture, une ligne de commentaire (identification de la lecture) commençant par le caractère '>' puis sur la/les ligne(s) suivante(s) la séquence avec 60 caractères maximum par ligne.

Ce fichier est ensuite transformé en graphe de De Bruijn via le programme debruijn3 écrit dans le langage de programmation C++ et mis à disposition par M. Sacomoto. Ce programme a été exécuté avec en paramètre d'entrée le fichier fasta, la taille du kmer souhaitée (-k), le nom et le format du fichier sortie (-o et -g). Dans le cadre de ce projet, le format DOT a été choisi pour le fichier sortie, qui sera utilisé par la suite. La commande suivante a été utilisée :

```
> ./debruijn3 fichier_reads -k 3 -o fichier_sortie -g 0"
```

En fait, l'option -g 0 produit un fichier au format DOT qui portera l'extension *.graph* et un deuxième fichier portant l'extension *.raw_edges* qui ne sera pas utilisé dans le cadre de ce projet. Dans un graphe de De Brujn au format DOT, les nœuds du graphe sont d'abord listés avec leurs noms puis vient l'énumération des arêtes de cette façon :

```
digraph debuijn {
0 [label="AAA / TTT"];
8 [label="CAA / TTG"];
13 [label="GAA / TTC"];
```

```
0 -> 0 [label="FF" weight=16484];
13 -> 0 [label="FF" weight=9484];
0 -> 13 [label="RR" weight=9484];
}
```

Les graphes de De Bruijn appliqués aux données de séquençage contiennent les séquences des nœuds mais aussi les reverses compléments, le graphe les considère comme un ensemble. De plus chaque arête possède une étiquette (label) indiquant s'il faut considérer la séquence F (forward) ou R (reverse) des nœuds impliqués.

I.2. Python et Pydot

Le programme conçu pour répondre aux objectifs de ce projet est écrit en langage de programmation Python, choisi pour sa prise en main plus rapide pour les non-informaticiens que des langages tels que C++ ou Java. Les graphes seront représentés graphiquement grâce au module Pydot qui permet entre autres de colorer les nœuds et d'enregistrer le graphe dans un fichier image au format png.

I.3. Algorithme du plus court chemin

Différents algorithmes (Kruskal, Floyd-Warshall, Dijkstra …) existent pour calculer le plus court chemin existant dans un graphe étant donné un nœud de départ. Dans le cadre de ce projet, le choix s'est porté sur l'algorithme de Dijkstra, plus facile à implémenter. De plus, il a été étudié en cours d'Informatique dispensé par B. Sinaimeri[2] (voir annexe 1), et est donc mieux connu. Il construit progressivement, à partir des données initiales, un sous-graphe dans lequel sont classés les différents sommets par ordre croissant de leur distance minimale au sommet source. La distance correspond à la somme des poids des arêtes empruntées. Pour ce projet, le poids des arêtes sera considéré comme égal à 1. L'avantage de cet algorithme est qu'il permet de garder en mémoire les nœuds déjà visités en les marquant : cela évite à l'algorithme de revenir en arrière ce qui permet d'améliorer sa performance. De plus, dès que le point d'arrivée est traité le plus court chemin du sommet source à ce point est trouvé puisque le chemin construit jusqu'ici est un chemin minimal.

I.4. Centralité de proximité

Différentes mesures de centralité existent, dans le cadre de ce projet c'est la centralité de proximité qui sera utilisée. Cette mesure définit la centralité en utilisant la notion de proximité ou de distance. Par exemple, si Xi est un sommet central qui interagit facilement avec les autres sommets, sa distance avec les autres doit être courte (d'où l'intérêt de calculer la plus courte distance entre chaque paire de nœuds par l'algorithme de Dijkstra). Pour calculer un degré de centralité associé à un nœud (compris entre 0 et 1), on utilisera la formule suivante :

$$C(i)=\frac{n-1}{\sum(di,j)}$$

Avec n → nombre total de nœuds du graphe ;
di,j → distance la plus courte à partir de l'acteur i vers l'acteur j
$\sum(di,j)$ → somme de j=1 à n

Dans me script implémenté pour ce projet, une petite modification est faite pour cette mesure. En effet, le degré de centralité doit être compris entre 0 et 1, or parfois la distance $d_{i,j}$ est égale à +∞ (dans l'implémentation on utilisera integer.max_value) quand il n'existe pas de chemin entre i et j. Cela pose problème quand par exemple, en considérant un graphe de 4 nœuds i, k, l et m, on évalue la distance du nœud i aux nœuds k, l et m : si di,i=0 ; di,k=3 ; di,l=+∞ et di,m=+∞ alors C(i)= (4-1)/(0+3+∞+∞), en effet ce calcul donnera un résultat proche de zéro, ce qui sera le cas dès qu'une des distances est égale à +∞. Pour contourner ce problème, on va considérer que s'il n'existe pas de chemin entre deux nœud, celle-ci sera égale à n car la distance maximale entre deux nœuds est n-1 (dans le cas où le poids des arêtes est de 1).

I.5. Graphes dirigés et non dirigés

A partir du fichier DOT que produit le programme debruijn3, il faut tout d'abord transformer le graphe produit en graphe dirigé d'une part, et en graphe non dirigé d'autre part, sous forme de liste d'adjacence. En effet, le programme associe une séquence et leur complément sur le même nœud.

Pour créer un graphe non dirigé, il suffit d'ignorer les étiquettes des arêtes et de laisser comme nom aux nœuds l'association des séquences F et R. Par exemple, considérons les deux nœuds et l'arête suivants :
0 [label="AAA / TTT"];
13 [label="GAA / TTC"]
13 -> 0 [label="FF" weight=9484];
Le nœud 0 portera le nom « AAA/TTT », le nœud 13 portera le nom « GAA/TTC », et l'arête 13 → 0 reliera « AAA/TTT » → « GAA/TTC ».

Pour créer un graphe dirigé, il faut séparer chaque nœud en deux dont l'un sera la séquence F et l'autre sera la séquence R. Il faudra alors veiller à prendre en compte les étiquettes des arêtes pour bien reconstruire les liens entre nœuds. Prenons le même exemple que précédemment : le nœud 0 donnera naissance au nœud « AAA » et au nœud« TTT ». De la même façon, le nœud 13 donnera naissance à « GAA » d'une part et à « TTC » d'autre part. L'arête 13 → 0, de label « FF », nous indique qu'il faut considérer la séquence F de 13 et la séquence F de 0. Ainsi, on aura « AAA » → « GAA ».

Ces transformations seront réalisées à l'aide de deux scripts Python, directedGraph.py et undirectedGraph.py, écrits pour ce projet. Ces scripts prennent en entrée le graphe au format DOT produit par le programme debruijn3, et produiront chacun un fichier texte contenant le graphe dirigé ou non sous forme de liste d'adjacence.

I.6. Script Project_main.py

Le script Python Project_main.py prend en entrée 3 arguments : le nom du fichier contenant les lectures, la taille des kmers souhaitée et enfin, un entier indiquant si on veut faire un graphe dirigé (1) ou non dirigé (0). La commande pour exécuter le script sera donc du type :

```
> python Project_main.py reads200.fa  6  1
```

Avec reads200.fa le fichier contenant les lectures, une taille de kmer de 6 et en choisissant un graphe dirigé.

En fonction du paramètre 3, le script va appeler le script Python shortestPathDirected.py pour un graphe dirigé, et le script shortestPathUndirected.py pour un graphe non dirigé. On se place dans le cas du graphe dirigé. L'algorithme utilisé dans shortestPathDirected.py comporte différentes étapes afin d'arriver au résultat (seules les étapes 2 et 3 seront différentes pour un graphe non dirigé :

- Etape 0 : lecture des arguments et ouverture des fichiers. Les noms des fichiers sont stockés

dans les variables adéquates, et les fichiers correspondants sont ouvert en lecture ou en écriture.

- Etape 1 : conversion de la liste de lectures (.fa) en graphe de De Bruijn. Le programme C++ debruijn3 est appelé et le fichier DOT est produit.
- Etape 2 : conversion du graphe de De Bruijn en graphe dirigé. Le script directedGraph.py est appelé sur le fichier DOT, et un fichier texte contenant la liste d'adjacence du graphe dirigé correspondant est créé.
- Etape 3 : parcours de la liste d'adjacence et stockage des nœuds dans un objet de type liste (nodes), et des arêtes dans un objet de type dictionnaire (edges) tel que nœud père (clé) => nœud fils (valeur).
- Etape 4 : dessin du graphe originel, d'après les informations contenues les objets nodes et edges.
- Etape 5 : pour tous les nœuds s du graphe, exécution de la fonction dijkstra qui calcule la plus courte distance de s à tous les autres nœuds du graphe. La fonction renvoie un dictionnaire dist contenant toutes les distances calculées pour s. Tous les dist issus des nœuds du graphe sont stockés dans un dictionnaire dico_final (nœud s => {liste des distances à tous les autres nœuds})
- Etape 6 : calcul de la centralité de chaque nœud et stockage dans un dictionnaire centrality tel que nœud s => degré de centralité.
- Etape 7 : dessin du graphe en associant à chaque nœud, une couleur en fonction de sa centralité.

Le script Project_main.py est donc capable, à partir d'un fichier de lectures au format fasta, de faire appel à d'autres scripts spécialement conçus pour le projets ou à des programmes préexistants, de dessiner le graphe de De Bruijn correspondant, étant donnée une taille de kmers, en affichant une couleur de nœud proportionnelle à leur centralité.

III. Résultats

III.1. Tests

Le projet a d'abord été testé sur un fichier créé « à la main », contenant un nombre restreint de petites lectures (voir annexe 2). Les résultats suivants ont été obtenus en utilisant un kmer égal à 3.

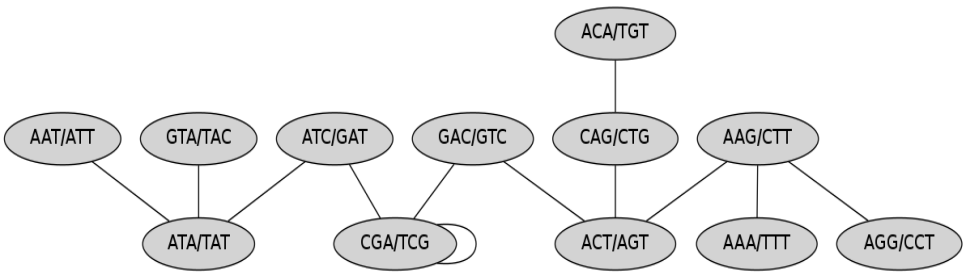


Figure 1 : Graphe non orienté dessiné par le programme avant calcul des centralités

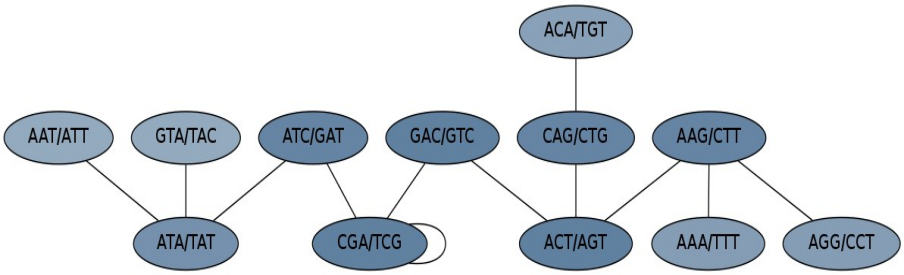


Figure 2 : Graphe non orienté dessiné par le programme après calcul des centralités

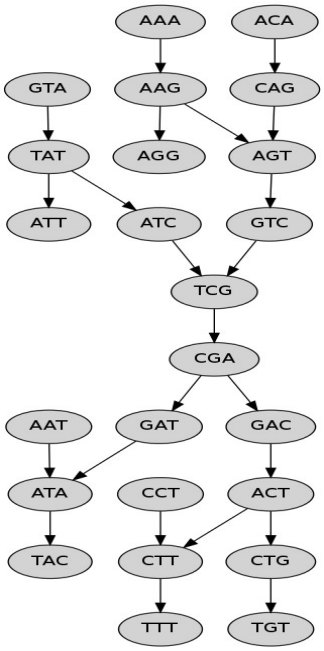


Figure 4 : Graphe orienté dessiné par le programme avant calcul des centralités

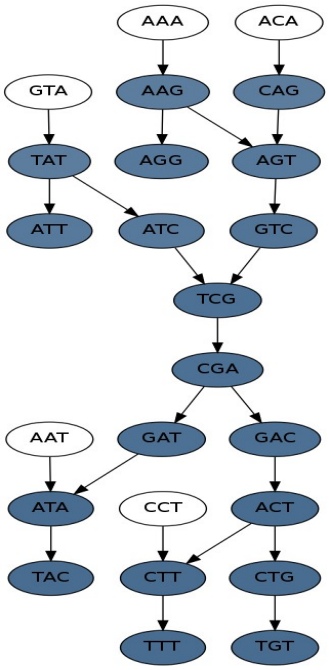


Figure 5 Graphe orienté dessiné par le programme après calcul des centralités

III.2. Application aux vraies données

Le projet a ensuite été testé sur des jeux de données plus larges, provenant de vrais séquençages (fichier HBM75brain_100000.fasta fournit par M. Sacomoto). Dans ce cas, les programme tourne bien et calcule bien les centralités, mais l'outil utilisé pour la représentation graphique ne permet pas de dessiner d'aussi grands graphes, quel que soit le kmer.

IV. Discussion et perspectives

Un programme en Python a été conçu pour lire un fichier de lectures et produire un graphe orienté et un graphe non orienté issus du graphe de De Bruijn associé. Ce programme permet également de calculer les degrés de centralité de chaque nœud, et de tracer ces graphes. Cette dernière étape ne fonctionne pas sur des graphes de taille moyenne à grande. On peut alors se demander si le bon outil a été utilisé pour la représentation graphique, ou s'il a été correctement paramétré. D'autre part, les résultats obtenus pour le petit fichier testé montrent que le dégradé de

couleur en fonction de la centralité est trop subtil et pas assez marqué. Une fonction de conversion du degré de centralité en couleur hexadécimale pourrait être envisagée pour palier ce problème.

Un autre problème soulevé est celui de la complexité du programme, qui n'a pas été calculée. En effet, cela pourrait également expliquer la difficulté du programme à tracer les graphes de taille importante ; on sait en particulier que l'algorithme de Dijkstra demande une certaine masse de traitements quand le graphe devient grand. Certains points pourraient être affinés, comme par exemple supprimer la redondance qui existe entre les deux scripts shortestPathDirected.py et shirtestPathUndirected.py.

Enfin, dans le cadre de ce projet, le poids des arêtes est considéré à 1 : on peut s'interroger sur les résultats qui auraient été produits en pondérant les arêtes. De plus, seulement la centralité de proximité a été utilisée, mais il pourrait être judicieux de calculer d'autres mesures de centralité et de comparer les résultats.

Conclusion

La centralité est un indice qui permet de caractériser un réseau et d'en identifier les sommets les plus importants. Au sein d'un réseau métabolique par exemple, un sommet de fort centralité pourrait être considéré comme intervenant dans un grand nombre de réactions chimiques. Dans un graphe de De Bruijn, cette centralité pourrait permettre de repérer les séquences répétées.

Bibliographie

[1] : **DIJKSTRA, E. W**., 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1: 269–271.
[2] : **SINAIMERI Blérina**, 2012. *Finding the Shortest path in a graph with Dijkstra algorithm*. UCBL1.

Ressources complémentaires

Sites internet

• *Algorithme de Dijkstra*. In Wikipedia [en ligne]. <http://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra> (consultation le 05/11/12)
• *Algorithme de Floyd-Warshall*. In Wikipedia [en ligne]. <http://fr.wikipedia.org/wiki/Algorithme_de_Floyd-Warshall> (consultation le 05/11/12)
• *Le pathfinding avec dijkstra*. In Le Site Du Zero [en ligne]. <http://www.siteduzero.com/tutoriel-3-35706-le-pathfinding-avec-dijkstra.html> (consultation le 05/11/12)
• **Duong-Khang NGUYEN, Julien POLO**. *Opencity : recherche du plus court chemin* [en ligne]. <http://opencity.sourceforge.net/pdf/AG41_OpenCity_ShortestPath_Report.pdf> (consultation le 05/11/12)
• **Hélène RENARD**. *Le problème du plus court chemin : algorithme de Dijkstra* [en ligne]. <http://users.polytech.unice.fr/~gaetano/asd/pdf/Slides_Dijkstra.pdf> (consultation le 05/11/12)
• *Pydot 1.0.2*. In Python [en ligne]. <http://pypi.python.org/pypi/pydot/1.0.2> (consultation le 30/11/12)
• *Calculateur de dégradé de couleurs*. In Skymac [en ligne]. < http://www.skymac.org/colorgradient.htm> (consultation le 05/12/12)

Enseignements dispensés au cour du M1 Ecosciences, Microbiologie, mention MIV à l'UCBL1 en 2012

• **LACROIX Vincent**, 2012. *Introduction to biological networks*. UCBL1
• **MIELLE Vincent**, 2012. *Réseaux et classification*. UCBL1

Annexe 1 : Algorithme de Dijkstra
Pseudo-code vu au cours de B. Sinaimeri

```
Dijkstra(G, s, w)
     FOR each v in V DO
          d[u] ← +∞           //d[v] is the distance estimated for
     the path from s to v
     ENDFOR

     d[s] ← 0
     S ← {s}
     Q ← V

     WHILE Q is not empty DO     //Q is a priority queue
          u ← extractMin(Q)       //Values are stored according to
     d[]
          S ← S U {u}
          FOR each v in AdjOut(u) DO
               IF d[v] > d[u] + w(u,v) THEN
                    d[v] ← d[u] + w(u,v)
               ENDIF
          ENDFOR
     ENDWHILE

     return d[ ]
END Dijkstra
```

Annexe 2 : Fichier fasta utilisé pour tester le programme

>r1
ATCGA
>r2
CGACT
>r3
ACTGT
>r4
ACTTT
>r5
GTATC
>r6
ACTTT
>r7
CCTTT
>r8
GTATT