Projet

Algorithmique et Complexité

Arbres couvrants

**M1 Informatique**

KIMM Raphaël

ROTON Théo **2020-2021**

Table des matières

[Questions 3](#_Toc60323604)

[Question 1 3](#_Toc60323605)

[Question 2 4](#_Toc60323606)

[Question 3 5](#_Toc60323607)

[Question 4 6](#_Toc60323608)

[Question 5 7](#_Toc60323609)

[Question 6 9](#_Toc60323610)

[Question 7 11](#_Toc60323611)

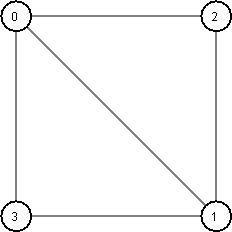
[Question 8 11](#_Toc60323612)

[Sources 13](#_Toc60323613)

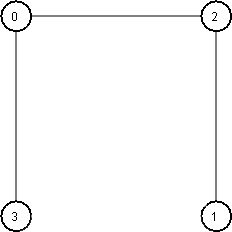
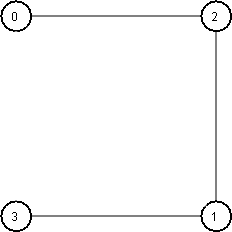
# Questions

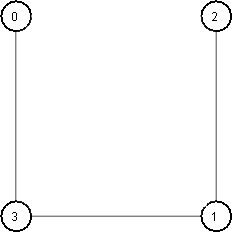
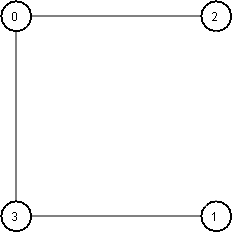
## Question 1

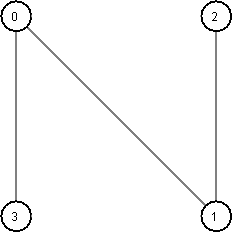
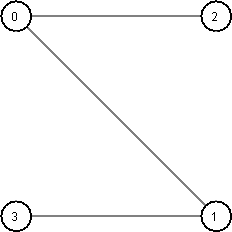
Nous avons le graphe G1 suivant :

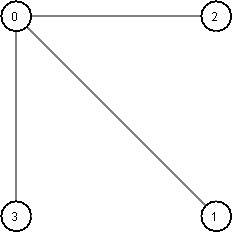
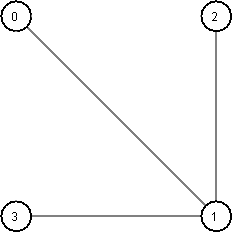


On peut trouver, pour ce graphe, 8 arbres couvrants différents. Voici les différents arbres que l’on peut avoir pour le graphe G1 :

**1.  2.

3. 4. 

5. 6. 

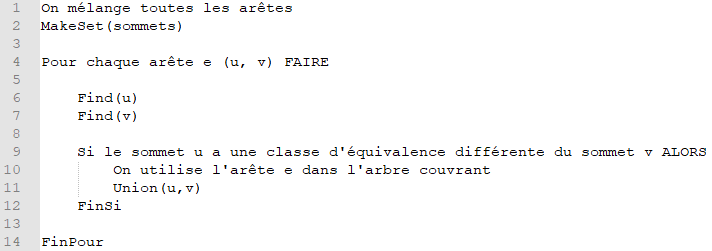
7.  8. 

Nous avons mis un numéro à chaque arbre couvrant (de 1 à 8). Ces numéros nous seront utiles dans les prochaines questions pour faire référence à ces arbres.

## Question 2

Dans cette question nous allons implémenter l’algorithme de Kruskal avec la structure « Union-Find » où on utilise un tableau d’entier pour faire la référence vers le nœud père. Nous allons utiliser 3 fonctions venant de la structure « Union-Find » :

* MakeSet(sommets) : qui initialise le tableau d’entier des références pour chaque sommet. Chaque sommet aura comme classe d’équivalence lui-même.
* Find(sommet) : qui détermine à quelle classe d’équivalence appartient le sommet.
* Union(sommet1, sommet2) : qui unit 2 classes d’équivalence en une seule.

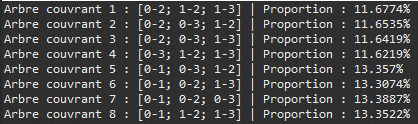
**Pseudo-algorithme :**

L’algorithme de Kruskal a été implémenté dans le fichier « Kruskal.java ».

## Question 3

Dans cette question, nous allons exécuter l’algorithme de Kruskal un million de fois sur le graphe G1 afin de voir la fréquence d’apparition de chaque arbre couvrant. Etant donné que nous avons 8 possibilités d’arbre couvrant pour le graphe G1, la fréquence d’apparition de chaque arbre devrait être d’environ 12,5%.

Après l’exécution de l’algorithme de Kruskal un million de fois, nous obtenons le résultat suivant :



Le numéro de l’arbre couvrant fait référence aux arbres couvrants trouvés dans la question 1. Pour chaque arbre, on a la liste des arêtes le formant entre crochets. Chaque arbre est représenté avec sa proportion d’apparition en utilisant l’algorithme de Kruskal. Le graphique suivant représente la fréquence d’apparition de chaque arbre couvrant :

On peut voir sur ce graphique que les arbres couvrants n’ont pas tous la même probabilité d’apparaître. Si on regarde en détail le graphique, on remarque que les 4 premiers arbres couvrants ont approximativement la même probabilité d’apparaître (≈ 11,65%). On peut aussi remarquer le même phénomène pour les 4 derniers arbres couvrants (≈ 13,35%). En utilisant l’algorithme de Kruskal, on voit donc que tous les arbres couvrants n’ont pas la même probabilité d’être trouvé.

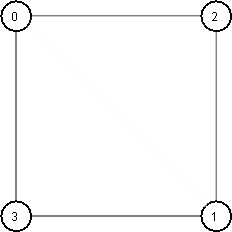
## Question 4

Nous pouvons remarquer, dans la question précédente, que les 4 arbres ayant une probabilité plus élevée d’apparaître possèdent tous l’arête 0-1 (l’arête en diagonale).

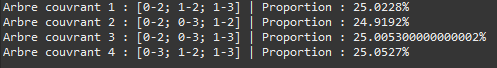
En regardant cette arête, on remarque qu’elle possède 4 arêtes voisines. Les arêtes voisines d’une arête « e » sont toutes les autres arêtes reliées aux 2 sommets de l’arête « e ». Les autres arêtes du graphe G1 possèdent quant à elles 3 arêtes voisines. Nous pouvons aussi noter que chaque arête du graphe G1 a pour arête voisine l’arête 0-1.

Par conséquent, lors de l’exécution de l’algorithme, la probabilité d’unir l’ensemble contenant l’arête 0-1 et un autre ensemble est donc beaucoup plus élevée. C’est pour cela que les arbres couvrants possédant l’arête 0-1 ont une plus grande probabilité d’apparaître lors de l’exécution de l’algorithme.

Prenons pour comparaison le graphe G2 suivant, qui est le même graphe que le graphe G1 mais sans l’arête 0-1 :



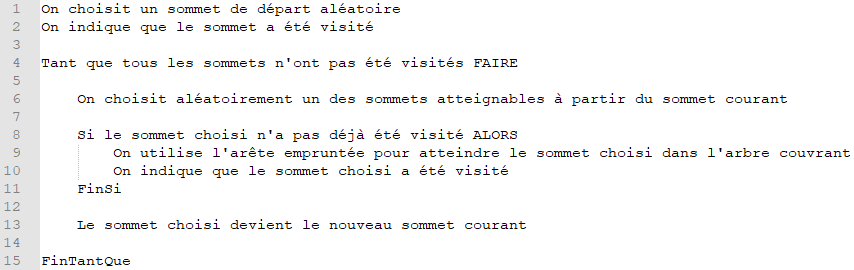
Pour ce graphe, nous pouvons obtenir 4 arbres couvrants différents (les 4 premiers arbres couvrants du graphe G1). Si nous exécutions l’algorithme de Kruskal un million de fois sur ce graphe, nous obtenons le résultat suivant :



Pour le graphe G2, nous pouvons obtenir chaque arbre couvrant avec approximativement la même probabilité (≈ 25%) . Ici, vu que l’on a enlevé l’arête 0-1, toutes les arêtes ont le même nombre d’arêtes voisines soit 2. Et donc la probabilité d’unir 2 ensembles lors de l’exécution de l’algorithme est la même à chaque étape.

## Question 5

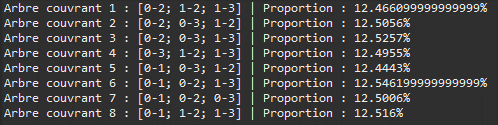
Dans cette question nous allons implémenter l’algorithme d’Aldous-Broder. Cet algorithme repose sur la notion de marche aléatoire : à chaque étape, on choisit un sommet atteignable aléatoirement (chaque sommet atteignable ayant la même probabilité d’être choisis).

**Pseudo-algorithme :**

L’algorithme d’Aldous-Broder a été implémenté dans le fichier « AldousBroder.java ».

Nous allons maintenant exécuter l’algorithme d’Aldous-Broder un million de fois sur le graphe G1 afin de voir la fréquence d’apparition de chaque arbre couvrant. Etant donné que nous avons 8 possibilités d’arbre couvrant pour le graphe G1, la fréquence d’apparition de chaque arbre devrait être d’environ 12,5%.

Après l’exécution de l’algorithme d’Aldous-Broder un million de fois, nous obtenons le résultat suivant :

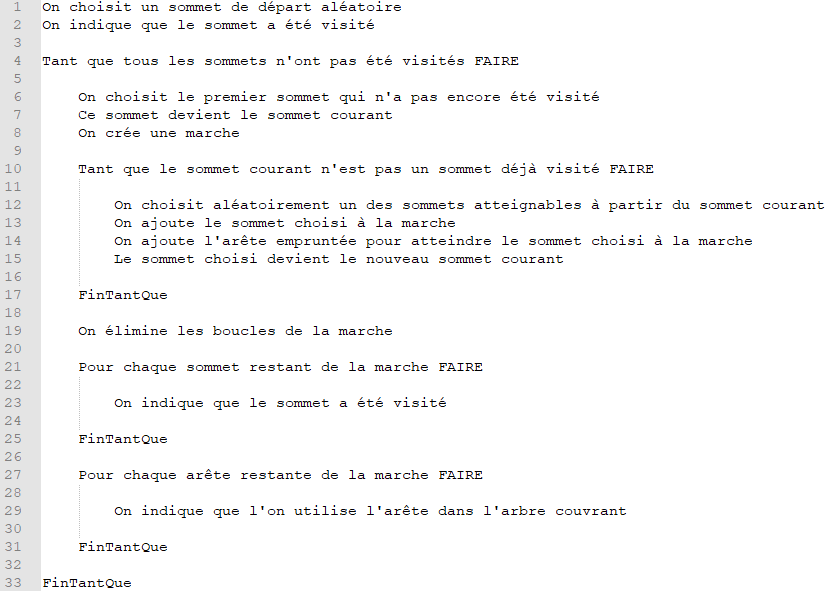


Le numéro de l’arbre couvrant fait référence aux arbres couvrants trouvés dans la question 1. Pour chaque arbre, on a la liste des arêtes le formant entre crochets. Chaque arbre est représenté avec sa proportion d’apparition en utilisant l’algorithme d’Aldous-Broder. Le graphique suivant représente la fréquence d’apparition de chaque arbre couvrant :

On peut voir sur ce graphique que les arbres couvrants ont approximativement la même probabilité (≈ 12,5%) d’apparaître. En utilisant l’algorithme d’Aldous-Broder, on voit donc que tous les arbres couvrants ont (approximativement) la même probabilité d’être trouvé.

## Question 6

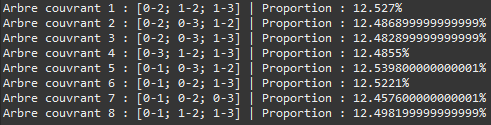
Dans cette question nous allons implémenter l’algorithme de Wilson. Cet algorithme repose aussi sur la notion de marche aléatoire, mais ici, l’algorithme va faire en sorte d’éliminer les boucles de la marche parcourue.

**Pseudo-algorithme :**

L’algorithme de Wilson a été implémenté dans le fichier « Wilson.java ».

Nous allons maintenant exécuter l’algorithme de Wilson un million de fois sur le graphe G1 afin de voir la fréquence d’apparition de chaque arbre couvrant. Etant donné que nous avons 8 possibilités d’arbre couvrant pour le graphe G1, la fréquence d’apparition de chaque arbre devrait être d’environ 12,5%.

Après l’exécution de l’algorithme de Wilson un million de fois, nous obtenons le résultat suivant :



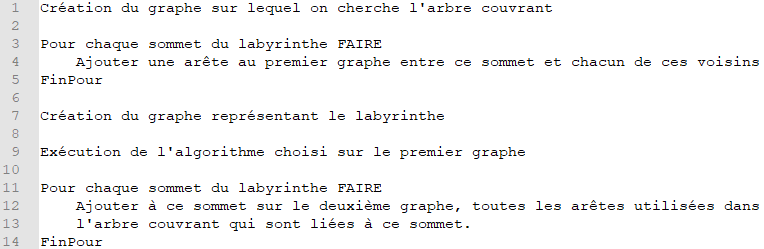
Le numéro de l’arbre couvrant fait référence aux arbres couvrants trouvés dans la question 1. Pour chaque arbre, on a la liste des arêtes le formant entre crochets. Chaque arbre est représenté avec sa proportion d’apparition en utilisant l’algorithme de Wilson. Le graphique suivant représente la fréquence d’apparition de chaque arbre couvrant :

On peut voir sur ce graphique que les arbres couvrants ont approximativement la même probabilité (≈ 12,5%) d’apparaître. En utilisant l’algorithme de Wilson, on voit donc que tous les arbres couvrants ont (approximativement) la même probabilité d’être trouvé.

## Question 7

Dans cette question, il nous est demandé d’écrire un algorithme qui crée un labyrinthe à partir des algorithmes de Kruskal et d’Aldous-Broder. Pour ce faire, nous allons utiliser 2 graphes :

* Le premier sera un graphe où tous les sommets sont reliés avec leurs voisins et sur lequel on va exécuter un des 2 algorithmes pour trouver l’arbre couvrant du graphe.
* Le deuxième sera le graphe qui représentera le labyrinthe et qui aura comme arête toutes les arêtes utilisées dans l’arbre couvrant. Les sommets du graphe représenteront les cases du labyrinthe et les arêtes les murs entre les cases.
* Chacun des 2 graphes aura le même nombre de sommets (N \* N pour un labyrinthe de taille N).

**Pseudo-algorithme :**

La création de labyrinthe a été implémenté dans le fichier « Labyrinthe.java ».

## Question 8

Nous allons maintenant comparer les 2 méthodes de génération de labyrinthes. Nous allons comparer le nombre moyen de culs de sac et la distance moyenne entre l’entrée et la sortie pour ces 2 algorithmes. Dans notre implémentation, comme on utilise un graphe pour représenter le labyrinthe, cela revient à :

* Nombre de culs de sac : compter le nombre de sommets qui ont une et une seule arête (sans compter l’entrée et la sortie).
* Distance entre l’entrée et la sortie : le nombre de sommet entre le sommet « d’entrée » (sommet en bas à gauche) et le sommet de « sortie » (sommet en haut à droite). Pour la distance nous utilisons du « Backtracking ».

Nous avons effectué 5 essais où nous générons 1000 labyrinthes de taille 20\*20 avec l’algorithme de Kruskal et l’algorithme d’Aldous-Broder. Pour ces 5 essais nous obtenons les résultats suivants :

Sur ces graphiques, nous pouvons voir que :

* L’algorithme de Kruskal génère plus de culs de sac que l’algorithme d’Aldous-Broder.
* La distance entre l’entrée et la sortie du labyrinthe est beaucoup plus grande pour l’algorithme d’Aldous-Broder que pour l’algorithme de Kruskal.

En résumé, utilisé l’algorithme de Kruskal permettra d’avoir des labyrinthes avec une plus courte distance entre l’entrée et la sortie mais avec plus de culs de sac, tandis qu’utilisé l’algorithme d’Aldous-Broder permettra d’avoir des labyrinthes avec moins de culs de sac mais une plus grande distance entre l’entrée et la sortie.

# Sources

Nous avons réalisé le projet en utilisant Java et les classes fournies :

* Classe **Graph** : ajout de 2 méthodes :
  + Méthode « resetArbreCouvrant » qui réinitialise l’arbre couvrant du graphe.
  + Méthode « aretesArbresCouvrant » qui renvoie toutes les arêtes utilisées dans l’arbre couvrant.
* Classe **Edge**: ajout de la méthode « equals » qui indique pour 2 sommets u et v si l’arête est égale à l’arête (u -> v).
* Classe **Kruskal**: implémentation de l’algorithme de Kruskal :
  + Méthode « arbreCouvrant » qui trouve, pour un graphe donné, un de ses arbres couvrant avec l’algorithme de Kruskal.
  + Méthode « testG1 » qui permet de tester N fois l’algorithme de Kruskal sur le graphe G1.
  + Méthode « testG2 » qui permet de tester N fois l’algorithme de Kruskal sur le graphe G2.
* Classe **AldousBroder** : implémentation de l’algorithme d’Aldous-Broder :
  + Méthode « arbreCouvrant » qui trouve, pour un graphe donné, un de ses arbres couvrant avec l’algorithme d’Aldous-Broder.
  + Méthode « testG1 » qui permet de tester N fois l’algorithme d’Aldous-Broder sur le graphe G1.
* Classe **Wilson** : implémentation de l’algorithme de Wilson :
  + Méthode « arbreCouvrant » qui trouve, pour un graphe donné, un de ses arbres couvrant avec l’algorithme de Wilson.
  + Méthode « testG1 » qui permet de tester N fois l’algorithme de wilson sur le graphe G1.
* Classe **Labyrinthe :** création de labyrinthe :
  + Méthode « lancerTest » qui permet de lancer le test du nombre de culs de sac et de la distance entrée/sortie.
* Classe **TestLabyrinthe**: classe de test du labyrinthe :
  + Méthode « lancerTest » qui permet d’effectuer N générations de labyrinthes de taille t\*t et de comparer les résultats entre l’algorithme de Kruskal et l’algorithme d’Aldous-Broder.
* Classe **JPanelLabyrinthe**: classe qui affiche le labyrinthe.
* Classe **Main**: lance la génération d’un labyrinthe et l’affiche le labyrinthe dans une fenêtre.