Année 2018 - 2019

BENNOUR Alexandre

KLEIN Christopher

Rapport du projet d’Algorithmes et Complexité

Ce rapport rend compte du projet d’algorithmes et complexité de Master Informatique 1 de BENNOUR Alexandre et KLEIN Christopher. Ce projet consistait à l’implémentation de différents algorithmes choisissant un arbre couvrant d’un graphe. De plus, nous utilisons ces algorithmes pour une application ludique, les labyrinthes.

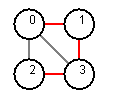
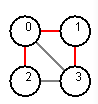
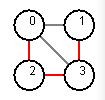
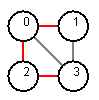
Ce rapport contient les réponses aux différentes questions posés par nos enseignants ainsi qu’une explication de la répartition du travail dans le groupe.

Vous pouvez retrouvez les sources java de notre projet sur le dépôt git suivant :

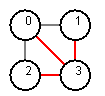
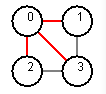
<https://github.com/christo57/Algo>

Question 1 (BENNOUR Alexandre):

Notre graphe représente un carré avec une diagonale. Donc on peut déjà obtenir 4 arbres couvrants en retirant la diagonale et une arrête du carré à chaque fois.



On a ensuite deux arbres couvrants en prenant la diagonale et deux arrêtes adjacentes.

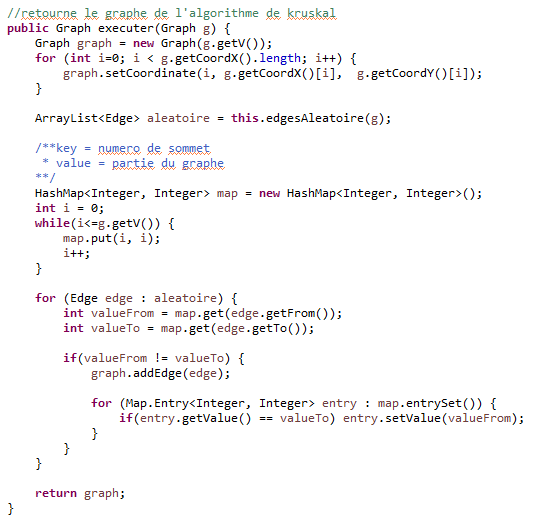


Et pour finir deux autres en prenant la diagonale et les arrêtes opposées.

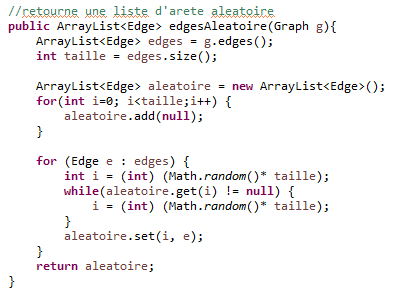


Ce qui nous fait un totale de 8 arbres couvrants pour le graphe G1

Question 2 (KLEIN Christopher):

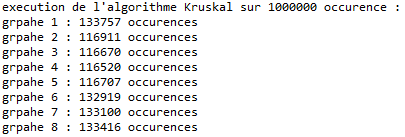
Voici l’implémentation de l’algorithme de Kruskal : 

La fonction edgesAleatoire(Graph) est une fonction que nous avons écrit pour obtenir une liste aléatoire d’arrêtes qui sont dans le Graph passé en paramètre :



Question 3 (BENNOUR Alexandre):

En testant l’algorithme un million de fois sur le graph G1 et en comptant l’apparition de chaque arbre couvrant obtenu, on obtient les résultats suivants :

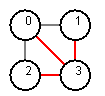
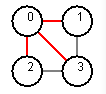


Donc sur les 8 arbres couvrants obtenus on voit bien que les fréquences d’apparitions ne sont pas les même, ils n’ont pas tous la même probabilité d’apparaitre.

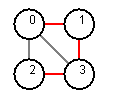
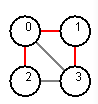
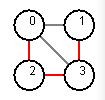
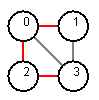
Question 4 (BENNOUR Alexandre – KLEIN Christopher):

Dans ce graphe, nous constatons qu’il y a 8 arbre couvrants possibles, vu ci-dessus. Mais nous voyons aussi que 4 arbres utilisent l’arête diagonal et que 4 arbres ne l’utilise pas :

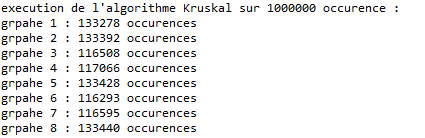
Voici les arbres couvrants utilisant l’arête diagonal :

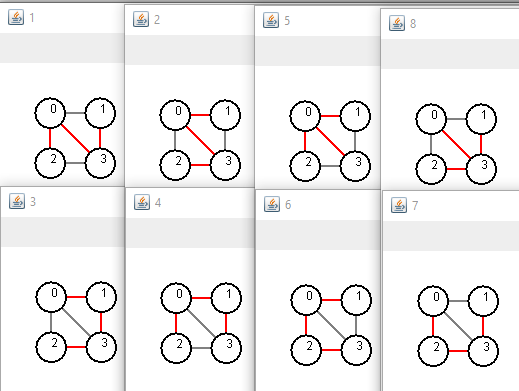


Et voici ceux ne l’utilisant pas :



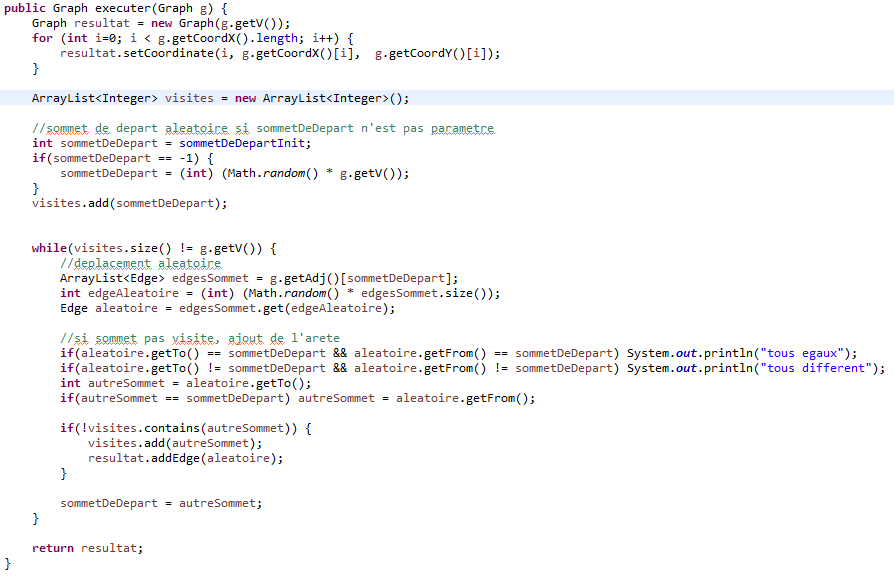
On remarque que chaque arête est contenue 5 fois dans l’ensemble des arbres couvrant, sauf l’arête diagonal, qui n’est présente que 4 fois. Donc lorsque l’on choisi l’arête diagonal, on supprime directement la moitié des arbres couvrant disponible. Nous avons donc plus de chance de tomber sur un arbre contenant l’arête diagonal plutôt qu’un arbre ne la contenant pas. Avec un exemple concret, on obtient ce résultat : on peut observer que les arbres contenant l’arête diagonal (1,2,5 et 8) ont une fréquence d’apparition de 13000 a peu près alors que les arbres ne la contenant pas ont une fréquence d’apparition d’a peu près 11500.



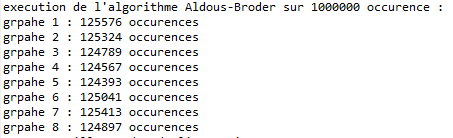


Question 5 (BENNOUR Alexandre – KLEIN Christopher):

Voici l’implémentation de l’algorithme d’Aldous-Broder :



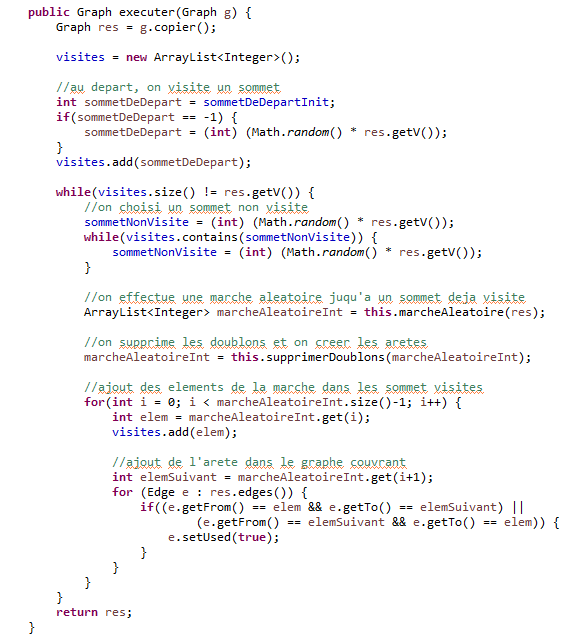
En testant l’algorithme un million de fois sur le graph G1 et en comptant l’apparition de chaque arbre couvrant obtenu, on obtient les résultats suivants :



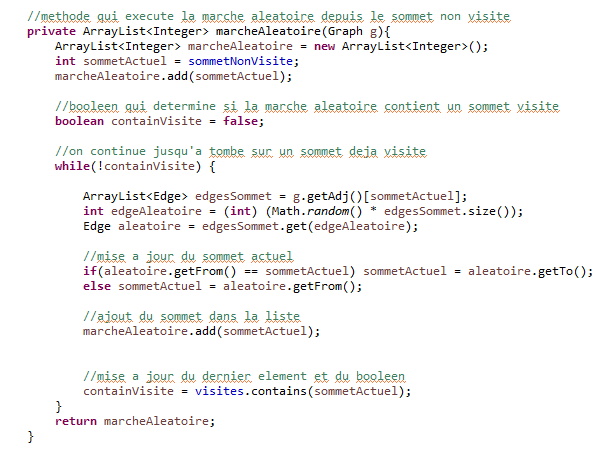
Donc sur les 8 arbres couvrants obtenus on voit bien que les fréquences d’apparitions sont sensiblement les mêmes, donc avec l’algorithme d’Aldous-Broder, on obtient des arbres couvrant avec une probabilité équivalente d’apparaître.

Question 6 (KLEIN Christopher):

Voici l’implémentation de l’algorithme de Wilson :



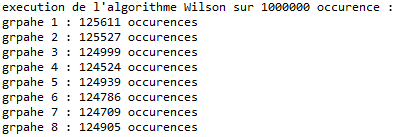
Cette méthode utilise plusieurs autres fonctions : la fonction marcheAleatoire(Graph g) qui permet d’exécuter une marche aléatoire depuis le sommet non visité jusqu’à un sommet déjà visité :



De plus, on utilise aussi une autre fonction, supprimerDoublons(ArrayList<Integer> liste) qui permet de supprimer tous les doublons d’une liste d’entier. Cette fonction utilise aussi une autre fonction, getDoublon(ArrayList<Integer> liste) qui permet de récupérer le premier doublons de la liste entrée en paramètres.



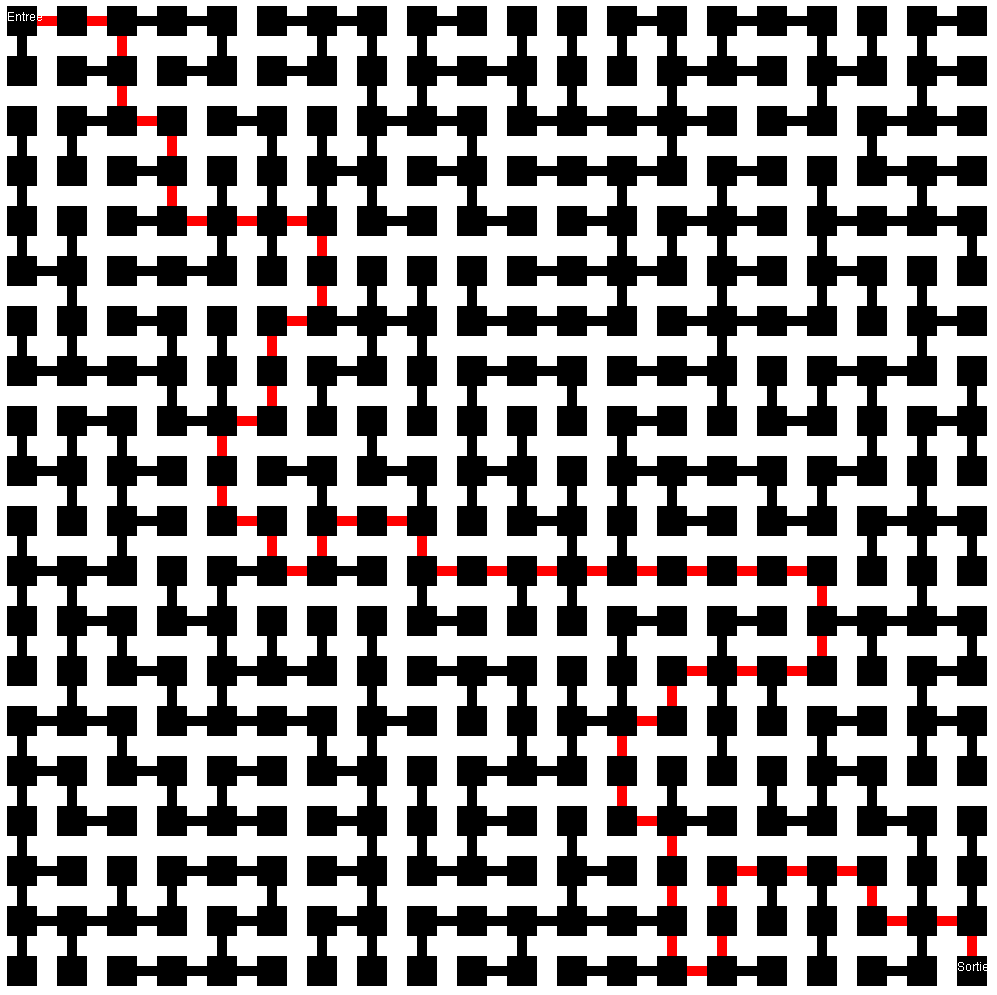
En testant l’algorithme un million de fois sur le graph G1 et en comptant l’apparition de chaque arbre couvrant obtenu, on obtient les résultats suivants :



Donc sur les 8 arbres couvrants obtenus on voit bien que les fréquences d’apparitions sont sensiblement les mêmes, donc avec l’algorithme de Wilson, on obtient des arbres couvrant avec une probabilité équivalente d’apparaître.

Question 7 (BENNOUR Alexandre):

Voici un labyrinthe généré à partir d’un arbre couvrant d’un graphe de 20x20 avec l’algorithme de Kruskal :

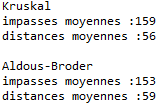
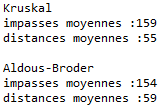
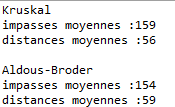


Le chemin rouge correspond au chemin entre l’entrée et la sortie.

L’algorithme qui permet de compter le nombre d’impasses et la distance vers la sortie consiste à dans un premier temps supprimer les premières impasses et les compter. Ces impasses correspondent aux nœuds (sommets du graph) qui n’ont qu’une seule arrête sans prendre en compte l’entrée et la sortie. Ensuite tant qu’il reste ce type de nœuds on continu à les supprimer, cette fois ci sans le compter car ils font partie des premières impasses dénombrées. Une fois toutes les impasses supprimées, on obtient le chemin qui mène de l’entrée à la sortie et il suffit de compter le nombre d’arêtes pour avoir la distance.

Question 8 (BENNOUR Alexandre):

En testant avec 1000 labyrinthes générés par l’algorithme de Kruskal et 1000 autres générés par l’algorithme d’Aldous-Broder, on obtient les moyennes suivantes :



Kruskal donne une distance inferieur à Aldous-Broder mais fournit plus d’impasses.

Partie 2 :

Question 9 (BENNOUR Alexandre – KLEIN Christopher) :

Il y a combinaisons de couleurs secrètes.

Question 10 (BENNOUR Alexandre – KLEIN Christopher) :

Si (b, 0) est la réponse correspondant à la proposition (p1, . . . , pN ) alors le joueur 2 peut envisager :

combinaisons secrètes.

Question 14 (BENNOUR Alexandre – KLEIN Christopher) :

Combinaison (entier k, entier n, entier b, entier m) : entier

Début

si n==b

alors retourne 0

sinon

entier bienPlace

entier malPlace

entier absent

si b==0

alors bienPlace = 0

sinon

bienPlace = 1 + combinaison(k-1,n-1,b-1,m)

fsi

si m==0

alors malPlace = 0

sinon

malPlace = 1 + combinaisons(k,n,b,m-1)

fsi

si n==k

alors absent = 0

sinon

absent = 1 + combinaison(k-1,n,b,m)

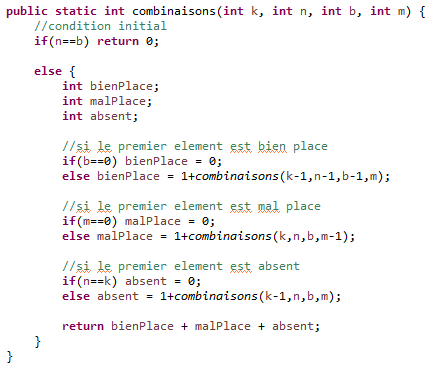
fsi

retourne bienPlace + malPlace + absent

fsi

Fin

Question 15 (KLEIN Christopher) :

Voici l’implémentation en Java de l’algorithme décrit ci-dessus : 

Avec ce programme, nous obtenons, pour N=4, K=6, b=1 et m=2, un résultat de 89 combinaisons :

