Rapport Devoir Algorithmique

Partie 1 : Algorithmes de création d’arbres couvrants

LANUEL Charlotte

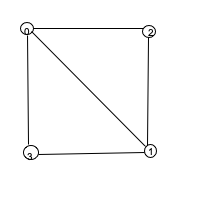
SCHWAB Lucas

M1 Informatique

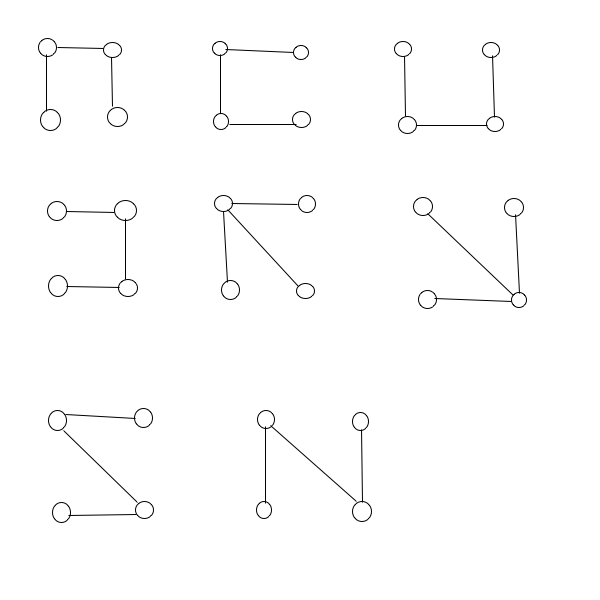
Tout le travail fourni pour ce projet a été réalisé en binôme. Le rapport a été écrit par Charlotte LANUEL.

# Question 1

Les huit arbres couvrants qui ressortent du graphe suivant :



Sont :



# Question 2 : Kruskal

L’algorithme de Kruskal randomisé a été au départ un peu difficile à implémenter. Bien sûr, on commence forcément par appliquer Collections.shuffle à la liste d’arêtes du graphe, puis on les ajoute une par une dans la liste du résultant en vérifiant si à chaque fois elle ne fait pas de cycle.

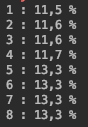
Ce qui a posé problème en premier lieu fut la détection de cycles. Nous avons d’abord créé une méthode estCyclique qui renvoie un booléen si les sommets de l’arête qu’on veut ajouter à la liste font déjà partie d’une arête qui est dans la liste de résultat. Mais cette fonction a été compliquée à mettre en place et nous avons finalement opté pour l’utilisation d’une structure Union-Find.

Cette structure nous a permis de séparer les sommets et lorsqu’on ajoute une arête à la liste du résultat, on vérifie au préalable que c’est possible : les sommets n’ont pas encore été reliés dans l’Union-Find donc aucun cycle ne peut être créé.

Pour détecter un cycle avec la structure Union-Find (déclarée statique dans le programme), on appelle la méthode union qui va tester pour deux points distincts s’ils ont la même racine à l’aide de la méthode récursive find. Si les deux sommets ont la même racine, union renvoie faux : ça engendrera un cycle. Sinon, pas de problèmes, on peut unir les deux sommets par une arête et l’ajouter à la liste des arêtes de l’arbre couvrant.

## Montrer expérimentalement que tous les arbres n’ont pas la même probabilité d’apparaître.

Une fois le programme exécuté un million de fois, on obtient les résultats suivants :



On voit clairement que les arbres couvrants qui sont fait à partir de l’arête diagonale (5,6,7 et 8) apparaissent à deux pour cent plus souvent que les autres.

## Montrer rigoureusement que tous les arbres couvrants n’ont pas la même probabilité d’apparaître.

On rappelle d’abord que le graphe sur lequel on a testé les algos contient quatre sommets et cinq arêtes. Pour faire un arbre couvrant qui passe par les quatre points, il faut trois arrêtes.

Donc : pour qu’une arête soit choisie en premier, elle a chances.

Il y a seulement 4 arbres couvrants produits avec l’arête diagonale.

Chaque arête qui forme le carré produit 5 arbres couvrants.

Donc : chaque arête a d’être prise en premier.

Si c’est l’arête diagonale, chaque arbre couvrant a chance d’être produit plus chance d’être produit avec une des deux autres arête qui le composent plus chance qu’il soit produit avec la dernière arête qui le compose (toujours choisi en premier).

Si ce n’est pas l’arête diagonale, chaque arbre couvrant qui peut être fait avec l’arête choisie a chance d’être produit.

Ce qui donne :

Pour les arbres couvrants produits avec l’arête diagonale (5,6,7,8) :

Pour les arbres couvrants n’utilisant pas l’arête diagonale (1 ,2,3,4) :

On retrouve les résultats précédents (arrondis). Tous les arbres couvrants n’ont pas la même probabilité d’être produits.

# Question 3 : Aldous-Broder

Pour Aldous-Broder, nous avons récupéré le nombre de sommet en premier lieu et avons créé un tableau de booléens de cette taille. Chaque fois qu’un sommet sera visité, on mettra le booléen associé à true.

A chaque fois qu’on visite un sommet, on réduit un compteur size. S’il est égal à zéro, on a visité tous les points et l’arbre couvrant est déterminé.

On choisit ensuite le sommet de départ aléatoirement parmi tous les sommets et on récupère les arêtes qui lui sont adjacentes.

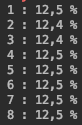
On choisit au hasard une des arêtes et on la parcourt. Nous avons bien pris garde à vérifier si le point où on arrive fait partie du départ ou de l’arrivé (bien que le graphe soit non orienté). Confondre les deux peut entraîner des erreurs. Si le sommet est déjà marqué à true dans le tableau, on n’ajoute pas l’arête dans l’arbre couvrant : soit on l’a déjà ajouté donc on a pas besoin de la prendre à nouveau, soit elle n’est pas marqué mais elle créé un cycle, donc on ne la prend pas.

On réitère cette étape tant que tous les sommets ne sont pas marqués à true, et donc que size est supérieur strictement à 0.

Choisir un tableau de booléens pour cet algorithme nous a semblé bien plus judicieux étant donné que l’algorithme lui-même se focalise sur le marquage de sommet et non des arêtes. C’est la structure la plus adaptée qui nous permet de ne pas faire appel à Union-Find et à prévenir l’apparition de cycle tout en gardant un code clair et fonctionnel.

## Prouver expérimentalement que tous les graphes ont la même chance d’apparaître.

Après avoir exécuté le programme un million de fois sur le graphe G1, il en ressort ces statistiques :



Pas de doute, tous les arbres couvrants ont exactement la même chance d’être produits.

# Question 4 : Wilson

La programmation de l’algorithme de Wilson s’est basé en partie sur celle d’Aldous-Broder.

L’exécution de l’algorithme se fait en trois temps :

* genNewUncoveredPoint est une fonction qui va générer une liste de points qu’on a pas encore parcouru. On choisit un nouveau point au hasard dans cette liste.
* CheminRetour est une procédure qui à partir du point choisi aléatoirement précédemment va construire un chemin jusqu’au prochain point qu’on a déjà parcouru.

C’est ici toute la subtilité de Wilson : on peut très bien faire demi-tour et créer des cycles, ils seront supprimés plus tard.

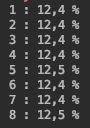
Les arêtes choisies pour constituer le chemin sont stockées dans une liste. Il sera plus facile par la suite de les retirer.

* enleverBoucle est la procédure la plus complexe. Elle se base sur la liste cheminRetour et sur un tableau d’entier nommé indice\_du\_premier\_depart qui va être rempli par des -1 en premier lieu, car c’est un indice qui ne peut pas exister dans un tableau. On va ensuite parcourir ce tableau : si l’indice est différent de -1 alors on va supprimer à l’aide d’une boucle toutes les arêtes de cheminRetour entre l’indice contenu dans le tableau et le compteur.

Si l’indice est à -1 alors on parcourt ce sommet pour la première fois et on va changer la valeur du tableau pour y mettre la valeur du compteur.

* Une fois cette étape terminée on appelle ajouterChemin qui va récupérer l’arbre retour une fois nettoyé et marqué tous les sommets à true.

Cet algorithme donne la même probabilité pour tous les arbres couvrants, expérimentalement, voici les résultat de l’exécution de l’algorithme un million de fois :



# Question 5 : Création de labyrinthes

La création des mille labyrinthes a nécessité un patron : un quadrillage 20x20 avec une entrée tout en bas à gauche et la sortie en haut à droite.

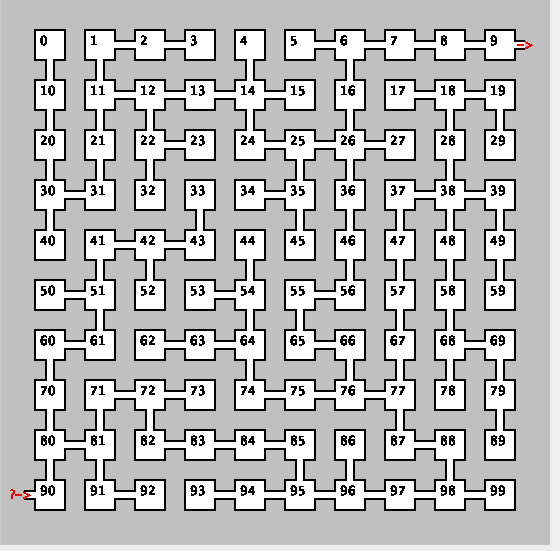


Figure 1 : Exemple de labyrinthe de taille 10

Le code qui a été ajouté concerne le calcul du nombre de cul-de-sac et la distance de l’entrée à la sortie. Ces deux méthodes ont été implémentées dans une classe à part des classes qui créent le labyrinthe.

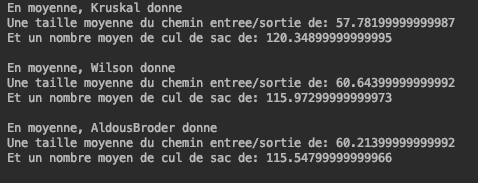
La méthode identifier() va prendre le tableau vu précédemment : point\_visites. Pour chaque point qui est visité, on part du principe que c’est un cul de sac, et on appelle la méthode récursive chercherAutour() qui va rechercher pour chaque arête non visitée le prochain sommet qui est un cul de sac.

Pour chaque arête qui est utilisée dans le labyrinthe, on instancie le pointFutur avec l’autre bout de cette arête et le booléen cds (comme cul de sac) est placé à false si le pointFutur n’a pas encore été visité.

On incrémente la variable cul\_de\_sac à chaque fois qu’on termine une boucle avec un csd à true.

Chercher autour incrémente aussi, à chaque traversée d’une arête, la variable chemin (int). Cette variable compte la distance entre l’entrée du labyrinthe et la sortie.

La moyenne des cul-de-sac et des distances entre l’entrée et la sortie est la suivante :



On constate que le nombre de cul-de-sac est supérieur pour Kruskal à l’inverse de sa distance de l’entrée vers la sortie. Les algorithmes de Wilson et Aldous-Broder étant très proches au niveau de l’implémentation, on retrouve la même moyenne pour les deux paramètres étudiés.