

Algorithmes et Complexité

Projet

Le devoir doit être effectué par groupe de 1 à 2 personnes, chaque personne appartenant à exactement un groupe.

1 Travail à effectuer

Description

Le devoir est à rendre sur **arche** pour le **4 janvier 2021 à 8h**.

Vous devez rendre :

- Les sources de vos programmes
- Un rapport répondant aux différentes questions, et qui explique en particulier la répartition du travail dans le groupe

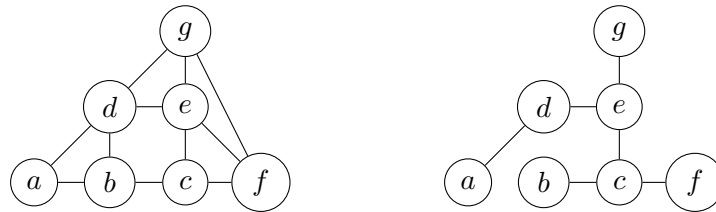
Vous pouvez bien entendu incorporer les graphiques dans le rapport.

2 Arbres couvrants

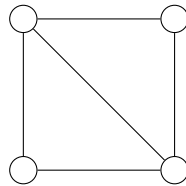
Note : L'énoncé est très long car il est composé de beaucoup d'exemples, mais son contenu est assez simple, à l'exception des questions Q4 et Q6, qui sont plus difficiles que les autres. A titre d'exemple, le professeur a réalisé les questions Q2 et Q5 en moins de 30 lignes chacune.

On rappelle qu'un arbre couvrant d'un graphe G (non orienté) est un ensemble d'arêtes du graphe qui forment un arbre (il n'y a pas de cycle) et qui relient tous les sommets entre eux.

Voici un exemple d'un graphe G_0 , et d'un arbre couvrant de G_0 .



Q 1) Déterminer par la méthode de votre choix tous les arbres couvrants du graphe G_1 suivant (Aide : il y en a 8) :



Le but de l'exercice est d'écrire un algorithme qui choisit un arbre couvrant au hasard uniformément parmi tous les arbres couvrants du graphe. Dans l'exemple du graphe G_1 , il y a 8 arbres couvrants, donc chacun d'entre eux aura une chance sur 8 d'être choisi.

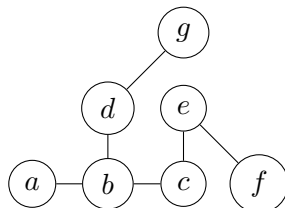
On va proposer trois algorithmes. Le premier est simple à mettre en oeuvre mais ne donne pas le bon résultat. Les deux autres donnent le bon résultat mais sont plus lents que le premier algorithme.

2.1 Algo 1 - Kruskal

Le premier algorithme fonctionne de la manière suivante :

- On mélange toutes les arêtes
- On les ajoute ensuite une par une sans créer de cycle.

Reprenons l'exemple du graphe G_0 . Il y a 11 arêtes : $ab, ad, bc, bd, ce, cf, de, dg, ef, eg, fg$. Si on les mélange, on obtient (par exemple) $ce, ef, ab, dg, bd, cf, bc, eg, cf, de, fg, ad$. Si on les ajoute sans créer de cycle, on obtient : ce, ef, ab, dg, bd, bc , c'est à dire :



Q 2) Implémenter l'algorithme de Kruskal.

Q 3) Tester l'algorithme un million de fois sur le graphe G_1 , et vérifier expérimentalement que les 8 arbres couvrants n'ont pas tous la même probabilité d'apparaître.

Q 4) Prouver rigoureusement que les 8 arbres couvrants n'ont pas tous la même probabilité d'apparaître.

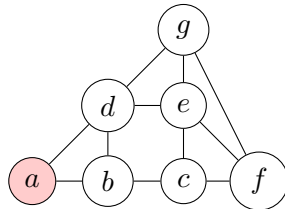
2.2 Algo 2 - Aldous-Broder

L'algorithme d'Aldous-Broder est un algorithme un peu plus lent mais qui donne bien la bonne distribution de probabilités : tous les arbres couvrants auront la même probabilité d'apparition.

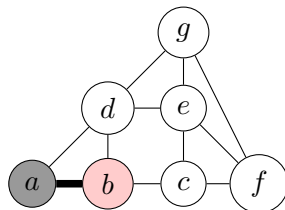
Il est basé sur la notion de marche aléatoire. Une marche aléatoire sur un graphe est obtenu de la façon suivante : On commence sur un sommet donné (aléatoire ou fixé), et à chaque étape, on se déplace aléatoirement sur l'un des voisins du sommet (tous les voisins ayant la même probabilité d'être choisi) et on recommence. Une marche aléatoire peut passer plusieurs fois par le même sommet : on peut même prendre une arête, et reprendre la même arête immédiatement dans l'autre sens !.

L'algorithme d'Aldous Broder fonctionne de la manière suivante : On part d'un sommet quelconque, et on effectue une marche aléatoire. Dès qu'on rencontre un nouveau sommet, on marque l'arête par laquelle on est passé. On s'arrête quand tous les sommets sont visités, et les arêtes marquées forment l'arbre couvrant.

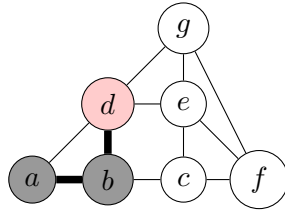
Essayons sur l'exemple du graphe G_0 , en partant du sommet a . On note en rouge le sommet sur lequel on est actuellement et en gris les sommets visités. Au départ, on est donc dans la configuration suivante :



On a donc une chance sur deux d'aller en d et une chance sur deux d'aller en b . Supposons qu'on décide d'aller en b . Le sommet b n'était pas visité, donc on marque l'arête ab , et on arrive en b :



Du sommet b , on a une chance sur 3 d'aller en a , une chance sur 3 d'aller en d , et une chance sur 3 d'aller en c . Supposons qu'on va en d , on obtient alors :

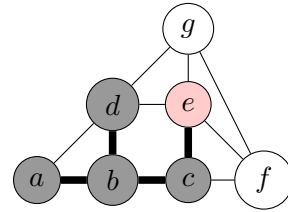
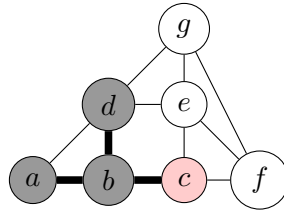
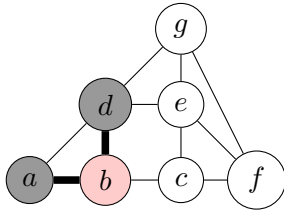


Et on continue ainsi, ce qui peut donner, par exemple, la suite d'opérations suivante :

1. on retourne en b

2. on va en c

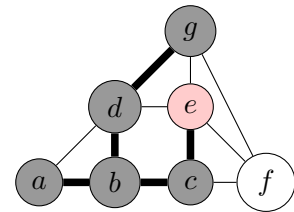
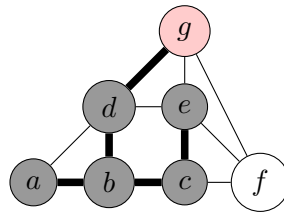
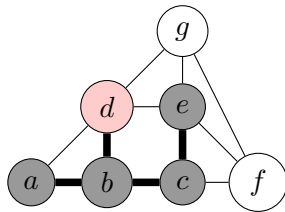
3. on va en e



4. on va en d (déjà visité)

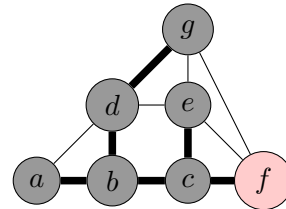
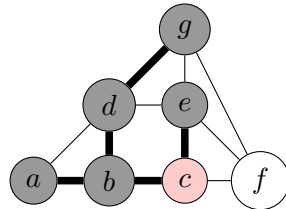
5. on va en g

6. on va en e (déjà visité)



7. on va en c (déjà visité)

8. on va en f



Q 5) Implémenter l'algorithme d'Aldous et Broder. Tester l'algorithme un million de fois sur le graphe G_1 , et vérifier expérimentalement que les 8 arbres couvrants ont tous la même probabilité d'apparaître.

2.3 Algorithme de Wilson

Note : cette partie est plus difficile que les précédentes.

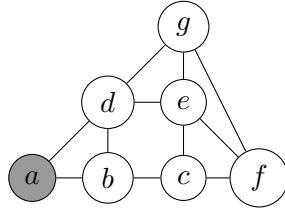
Le troisième algorithme, dû à Wilson, donne également le bon résultat, et est plus rapide que l'algorithme précédent.

Son fonctionnement est le suivant :

- Au départ, on visite un sommet (par exemple le sommet 0).
- A un moment donné de l'algorithme, on a donc des sommets visités et des sommets non visités.

- On choisit un sommet non visité (on le choisit comme on veut, aléatoirement ou non, ce n'est pas important)
- On effectue une marche aléatoire jusqu'à tomber sur un sommet déjà visité
- On élimine les boucles de la marche
- On ajoute les sommets qui restent aux sommets visités, et les arêtes qui restent dans l'arbre couvrant

Reprenons l'exemple du graphe G_0 . On suppose le sommet a déjà visité.



On prend un sommet non visité quelconque, par exemple g , et on lance une marche aléatoire jusqu'à tomber sur un sommet visité (donc ici, il faut tomber sur a).

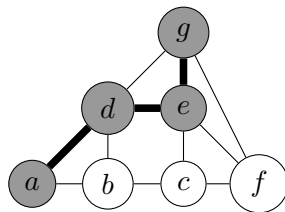
Supposons que la marche aléatoire donne le chemin suivant :

$$g - e - d - b - c - e - d - a$$

On élimine les boucles du chemin. Pour cela, on regarde le premier sommet qui apparaît au moins deux fois. S'il n'y en a aucun c'est terminé. Sinon, on enlève tout jusqu'à la dernière occurrence de ce sommet et on recommence. Dans notre exemple, les sommets e et d apparaissent deux fois : le premier dans le chemin, c'est e , on enlève donc tout ce qu'il y a entre la première occurrence de e et la dernière occurrence de e et on obtient :

$$g - e - d - a$$

On ajoute ensuite tous les sommets aux sommets visités, et toutes les arêtes à l'arbre pour obtenir :



Et on recommence : on prend un sommet non visité (par exemple c et on lance une marche aléatoire jusqu'à tomber sur un sommet visité, etc.

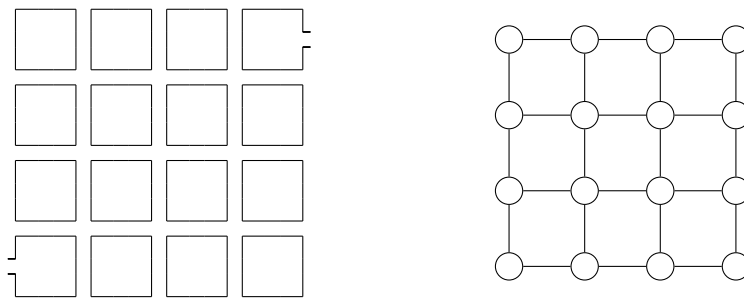
Q 6) Implémenter l'algorithme de Wilson. Tester l'algorithme un million de fois sur le graphe G_1 , et vérifier expérimentalement que les 8 arbres couvrants ont tous la même probabilité d'apparaître.

2.4 Application ludique : les labyrinthes

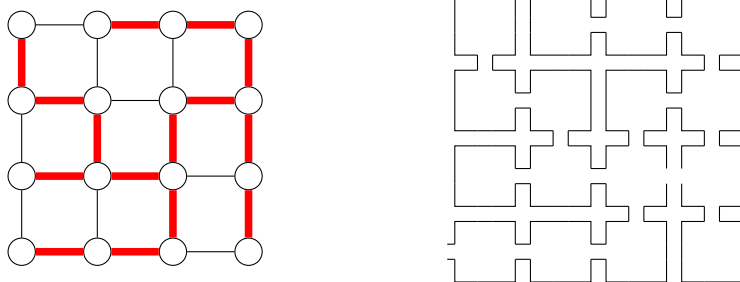
Outre les applications sérieuses des arbres couvrants en réseaux, ils permettent également de créer des labyrinthes. Partant d'un ensemble de cases, toutes isolées par des murs, on crée un labyrinthe de la façon suivante :

- On crée un graphe où les sommets sont les cases, et les arêtes les murs
- On tire un arbre couvrant du graphe
- On met un passage là où il y a une arête de l'arbre couvrant, et un mur sinon.

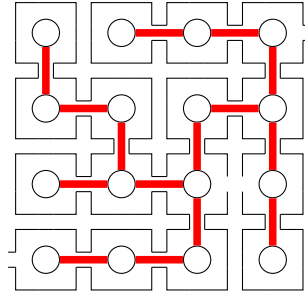
Prenons l'exemple d'un labyrinthe 4×4 . Voici le labyrinthe au début, alors qu'aucune arête n'a été choisie pour l'arbre couvrant. Il y a donc des murs partout. On a mis arbitrairement l'entrée en bas à gauche et la sortie en haut à droite, mais cela ne se voit pas dans le graphe :



On trouve ensuite un arbre couvrant aléatoire du graphe, et on reporte dans le labyrinthe. Par exemple :



Pour bien comprendre, on peut superposer les deux images :



Q 7) Ecrire un algorithme qui crée un labyrinthe 20x20 obtenu avec la méthode de Kruskal et un labyrinthe 20x20 obtenu avec la méthode de Aldous-Broder (ou Wilson).

Q 8) Comparer les deux méthodes. Pour cela, on tirera au hasard 1000 labyrinthes (ou plus) suivant les deux méthodes et on comparera :

- Le nombre moyen de culs de sac
- La distance moyenne de l'entrée à la sortie

2.5 Programmation

Le projet doit être implémenté dans un langage de programmation usuel : Java, C, Rust, Go, Python. Il est possible de choisir un autre langage, en informant d'abord le professeur pour voir s'il est d'accord.

Des classes de bases sont fournies en Java. Il est conseillé de partir sur une implémentation similaire dans les autres langages.

Classes fournies en Java

Détaillons les différentes classes :

- La classe **Edge** représente une arête non orientée. Elle dispose d'une méthode **other** qui renvoie le sommet à l'autre extrémité de l'arête. Le champ **used** permet de savoir si l'arête est utilisée dans l'arbre couvrant.
- La classe **Graphe** représente un graphe.
 - On peut créer un graphe à N sommets en appelant le constructeur.
 - On peut ajouter arête après arête au graphe avec la méthode **addEdge**.
 - La méthode **adj** renvoie la liste des arêtes reliées à un sommet.
 - La méthode **edges** renvoie toutes les arêtes.
 - La méthode **writeFile** permet d'écrire un fichier dot contenant le résultat
 - La méthode statique **example** renvoie un petit graphe de Test (le graphe G_1)
 - La méthode statique **Grid** renvoie une grille de $n \times n$.
- La classe **Display** sert à débbugger et permet d'afficher le graphe dans une fenêtre. On peut éventuellement afficher plusieurs fenêtres d'un coup. Un exemple d'utilisation est donné dans le fichier **Test**

- La classe `Test` donne un exemple d'utilisation. La méthode `printLaby` permet également de créer un fichier `.tex` contenant un labyrinthe. On peut ensuite convertir le labyrinthe en `.pdf` en tapant `pdflatex toto.tex` dans un terminal.

Aides

- Pour implémenter Kruskal, il peut être bon de programmer une structure Union-Find. L'implémentation dans la Wikipedia française est suffisante, en remplaçant par simplicité la référence vers le noeud père par un tableau `parent`.
- La méthode `Collections.shuffle` peut être très utile. A noter que toutes les méthodes de la classe `Graphe` renvoient des copies des `(Array)`listes, donc vous pouvez appeler `Collections.shuffle` sur le résultat sans remords.