

Les arbres couvrants dans les graphes possèdent de très nombreuses applications, et pas seulement en informatique ou en mathématiques, voir par exemple [la page wikipédia anglophone](#) qui liste plusieurs applications pour le calcul d'un arbre couvrant de poids minimum.

Moins célèbre, le problème de générer aléatoirement un arbre couvrant est pourtant aussi très utilisés, notamment par plusieurs algorithmes d'optimisation. Il s'agit, étant donné un graphe, de construire un arbre couvrant choisi aléatoirement parmi tous les arbres couvrants du graphe. Nous allons programmer plusieurs algorithmes, comparer leurs performances et la qualité des arbres construits.

1 Rendu

Ce devoir est constitué de plusieurs algorithmes indépendants. **Vous ne devez pas tous les coder** pour avoir une bonne note, essayez simplement d'en écrire un maximum avec le temps dont vous disposez en TP. De plus, vous n'êtes pas limité par les algorithmes que ce sujet vous présente : au contraire nous vous encourageons à trouver votre propre solution et à la comparer aux algorithmes du sujet.

À l'issue des séances de TP qui lui sont dédiées, et à la date prescrite, vous devrez rendre sur Ametice, sous forme d'un unique fichier archive d'extension zip, gz, ou tar.gz (**exclusivement**), contenant l'ensemble de vos sources (dans un sous-répertoire nommé `src`) et un bref rapport (dans un simple fichier texte qu'on appellera `README.txt`) décrivant comment utiliser votre projet, vos résultats, et une comparaison des algorithmes que vous avez codés. Un fichier `Makefile`, contenant les cibles usuelles `compile`, `exec`, `test`, `exec`, sera aussi inclu ; la notation sera meilleure aussi en fonction de la possibilité d'apprécier votre travail par le `Makefile`.

Si votre projet contient du code source que vous n'avez pas écrit vous-même, **vous indiquerez précisément sa nature et sa provenance** (internet, les amis, ...). Tout projet contenant du code non-correctement attribué à ses véritables auteurs s'expose à recevoir une note nulle. Vous êtes priés de ne pas fournir votre code à d'autres étudiants, sous peine de vous exposer aussi à recevoir une note nulle.

2 Présentation du problème

Rappelons qu'un arbre couvrant d'un graphe non-orienté connexe $G = (V, E)$ est donné par un sous-ensemble d'arêtes $F \subset E$ acyclique (F n'induit pas de cycle), connexe (toute paire de sommets de G est reliée par un chemin dans F), et avec $|F| = |V| - 1$.

Étant donné $c : E \rightarrow \mathbb{R}$, le poids d'un arbre F est défini par $\sum_{e \in F} c(e)$. L'[algorithme de Prim](#) et l'[algorithme de Kruskal](#) permettent de trouver l'arbre couvrant de poids minimum.

Dans le cadre de ce devoir, nous voulons construire un arbre arbitraire. Notons \mathcal{T} l'ensemble de tous les arbres couvrants de G . Notre algorithme idéal serait de construire un arbre choisi aléatoirement, de telle sorte que pour tout arbre $T \in \mathcal{T}$, on ait :

$$\Pr[T \text{ est l'arbre construit}] = \frac{1}{|\mathcal{T}|}$$

et que le tirage aléatoire soit indépendant à chaque utilisation de notre algorithme. Autrement dit, notre algorithme serait un *générateur aléatoire uniforme* d'arbres couvrants : tous les arbres ont la même probabilité d'être choisis à chaque appel de l'algorithme.

Une façon naïve de faire serait d'énumérer tous les arbres couvrants du graphe, c'est-à-dire de construire une liste d'arbres $T_1, T_2, \dots, T_{|\mathcal{T}|}$ telle que

$$\mathcal{T} = \{T_1, T_2, \dots, T_{|\mathcal{T}|}\},$$

puis de choisir un entier i entre 1 et $|\mathcal{T}|$ aléatoirement uniformément, et de retourner T_i . Le problème est que $|\mathcal{T}|$ est bien trop grand : pour un graphe complet à n sommet, $|\mathcal{T}| = n^{n-2}$, une quantité super-exponentielle par rapport à n . L'énumération ne peut donc pas être construite. Une alternative serait de construire T_i seulement à partir de i , mais cela semble compliqué.

Nous allons voir deux algorithmes permettant d'obtenir un arbre choisi uniformément. Pour chacun, prouver qu'ils ont la propriété d'uniformité est loin d'être immédiat (mais pas hors de portée des plus mathématiciens d'entre vous). Nous nous contenterons donc de les implémenter en faisant confiance en leurs concepteurs.

Ces deux algorithmes sont des réponses idéales à notre problème d'un point de vue mathématique, mais leur complexité asymptotique laisse à désirer. Par ailleurs, il n'est pas toujours vrai que l'application exige la propriété d'uniformité. On peut vouloir des arbres aléatoires avec un grand diamètre, ou au contraire un petit diamètre, avec beaucoup, ou peu de nœuds internes, privilégiant certaines arêtes plutôt que d'autres... Il est donc aussi pertinent de regarder des générateurs non-uniformes, surtout s'ils sont en plus efficaces. Nous présenterons quelques exemples. Mais libre à vous d'inventer d'autres méthodes et de les comparer à celles listées dans le sujet.

3 Quelques algorithmes

Certains de ces algorithmes peuvent être les mêmes, présentés différemment. De quoi vous faire un peu réfléchir !

3.1 Arbres couvrant de poids minimum aléatoire

Voici l'algorithme techniquement le plus simple pour obtenir un arbre couvrant qui semble vraiment très aléatoire.

1. Attribuer à chaque arête un poids réel choisi dans $[0, 1]$ uniformément aléatoirement, indépendamment pour chaque arête.
2. Retourner un arbre couvrant de poids minimum (pour ce choix de poids).

Pour la deuxième étape, utiliser l'algorithme de Prim ou celui de Kruskal par exemple.

3.2 Parcours aléatoire

On effectue un parcours du graphe en partant d'un sommet aléatoire. Lors de l'extraction d'un arc de la frontière, on choisit l'arc aléatoirement parmi ceux de la frontière, plutôt que de choisir le premier entré (parcours en largeur) ou le dernier entré (parcours en profondeur). On garde l'arbre défini par les arcs prédécesseurs (ce sont les arcs qui ont permis d'atteindre un sommet pour la première fois).

Il est aussi possible de faire un parcours en largeur (ou en profondeur) aléatoire : au moment d'ajouter les arcs sortants d'un sommet dans la frontière, les ajouter dans un ordre aléatoire. Cela suffit pour varier les arbres générés. Pour le parcours en largeur, on peut même mélanger tous les arcs sortants des sommets qui sont à la même distance de la racine.

3.3 Insertion aléatoire d'arêtes

Partant d'un ensemble F d'arêtes vide, on tire aléatoirement une arête e du graphe. Si $\{e\} \cup F$ ne contient pas de cycle, on ajoute e à F . On tire ainsi des arêtes jusqu'à avoir $|F| = |V| - 1$, F est alors un arbre couvrant.

Toute la difficulté vient de détecter si $\{e\} \cup F$ contient un cycle. C'est le même problème (donc la même solution) que l'on retrouve dans l'algorithme de Kruskal.

3.4 Algorithme d'Aldous-Broder

On effectue une *marche aléatoire* dans le graphe. Qu'est-ce donc ? On utilise une variable `sommetActuel` contenant initialement un sommet quelconque du graphe. Puis, à chaque étape, on met à jour `sommetActuel` avec un de ses voisins, choisi aléatoirement uniformément. On répète jusqu'à être passé au moins une fois par chaque sommet (chaque sommet a été affecté au moins une fois dans `sommetActuel`). Il faut imaginer un bonhomme un peu ivre, se déplaçant en suivant les arêtes du graphe, et prenant à chaque sommet une direction aléatoire.

Pour chaque sommet u différent du sommet initial de la marche, on sélectionne l'arête qui a permis d'entrer dans u pour la première fois au cours de la marche. On obtient ainsi $|V| - 1$ arêtes, qui forment un arbre couvrant. Le choix ainsi fait est uniforme.

3.5 Par contraction d'arêtes

Choisir une arête aléatoirement e qui n'est pas une boucle, et contracter cette arête, c'est-à-dire considérer le graphe $G_e = (V \setminus \{\text{dst}_G(e)\}, E)$ avec :

$$\begin{aligned}\text{src}_{G_e}(e') &= \begin{cases} \text{src}_G(e') & \text{si } \text{src}_G(e') \neq \text{dst}_G(e) \\ \text{src}_G(e) & \text{sinon} \end{cases} \\ \text{dst}_{G_e}(e') &= \begin{cases} \text{dst}_G(e') & \text{si } \text{dst}_G(e') \neq \text{dst}_G(e) \\ \text{src}_G(e) & \text{sinon} \end{cases}\end{aligned}$$

(Rappelons que nous notons par $\text{src}_G(e)$ et $\text{dst}_G(e)$ les deux extrémités d'une arête e dans le graphe G).

Par induction, trouver un arbre couvrant T_e de G_e . Alors $T_e \cup \{e\}$ est un arbre couvrant de G .

Attention, l'opération de contraction est (parfois) pénible à implémenter (selon la façon dont le graphe est codé), notamment parce qu'elle peut créer des boucles ou des arêtes parallèles dans le graphe. Il faut donc faire attention à écrire des algorithmes supportant les boucles et arcs parallèles.

3.6 Algorithme de Wilson

Choisir un sommet initial v (de préférence un sommet de degré maximum). On construit l'arbre par étape, au début il ne contient que le sommet v et aucune arête, et on ajoute à chaque étape un chemin depuis un sommet qui n'est pas dans l'arbre vers un sommet qui est dans l'arbre.

Chaque étape consiste en :

1. choisir un sommet u qui n'est pas dans l'arbre,
2. faire une marche aléatoire depuis ce sommet, jusqu'à atteindre un sommet w de l'arbre,
3. supprimer les cycles de cette marche pour obtenir un chemin de u à w ,
4. ajouter ce chemin (et ses sommets) à l'arbre.

La suppression des cycles paraît être le travail le plus difficile, mais il y a une technique très simple. Lorsque la marche aléatoire quitte un sommet x pour aller au sommet adjacent y , en utilisant l'arête e (d'extrémités x et y), marquer le sommet x avec e . Une fois que la marche atteint l'arbre, on repart de u en suivant les marques : ceci donne le chemin sans cycle cherché.

Comme l'algorithme d'Aldous-Broder, l'algorithme de Wilson construit un arbre couvrant choisi aléatoirement uniformément.

3.7 Par flips successifs d'un arbre

On part d'un arbre quelconque T . On répète ensuite de nombreuses fois l'opération suivante (appelée flip) :

1. choisir une arête e qui n'est pas dans l'arbre T ,
2. choisir une arête e' dans l'unique circuit de $T \cup \{e\}$,

3. mettre à jour $T \leftarrow (T \setminus \{e\}) \cup \{e'\}$.

Idéalement, il faudrait choisir la paire (e, e') uniformément, mais cela prendrait trop de temps à calculer pour que l'algorithme soit efficace. Une méthode plus simple est de fixer une racine r à l'arbre T , et d'orienter toutes les arêtes de T vers la racine. Alors choisir e parmi les arêtes incidentes à r , ainsi $e = ru$, et prendre e' l'arc sortant de u dans T . Ensuite, déplacer la racine à u . On peut facilement encoder l'arbre comme un tableau associant à chaque sommet l'arc vers son père dans l'arbre, sauf la racine à laquelle on associe `null`. L'opération flip est alors très facile à coder.

Plus le nombre de flips effectués est grand, plus la distribution obtenue sera proche d'être uniforme. Théoriquement il faudrait $O(|V|^3)$ flips dans le pire des graphes (mais possiblement moins dans les graphes que nous avons : faites des tests et comparez avec les algorithmes d'Aldous-Broder ou de Wilson).

3.8 Par suppression de cycles

Choisir un sommet racine r . Pour tous les autres sommets, choisir une arête incidente. L'ensemble des arêtes choisies F vérifie bien $|F| = |V| - 1$, mais il peut contenir des cycles (de longueur 2 ou plus) et donc ne pas être un arbre. Dans ce cas, trouver un cycle, et retirer pour chaque sommet de ce cycle une nouvelle arête incidente.

La difficulté est de détecter les cycles. Une façon simple est de choisir un sommet et de suivre la suite d'arêtes associées : soit on arrive à r et tous les sommets parcourus ne sont pas dans des cycles, soit on passe deux fois par un même sommet et c'est qu'il y a un cycle depuis ce sommet.

Dès qu'il n'y a plus de cycle, on retourne l'arbre obtenu.

3.9 Par suppression de sommet

On choisit un sommet v , et une arête incidente e à ce sommet. On supprime le sommet v , on trouve un arbre aléatoire F_v (dans le graphe sans v), et on retourne $F_v \cup \{e\}$.

Il y a un problème si supprimer le sommet v déconnecte le graphe, il faut donc interdire de choisir un tel sommet, qu'on appelle sommet d'articulation. Il est possible de détecter les sommets d'articulations avec un parcours en profondeur, mais ce n'est pas trivial.

4 Travail

Implémentez une classe des graphes. Suivez le prototype fourni, mais ajoutez les fonctions qui vous semblent nécessaires par la suite.

Implémentez plusieurs de ces algorithmes. Au minimum, il faut en écrire un basé sur un parcours de graphe, et soit l'algorithme d'Aldous-Broder soit l'algorithme de Wilson. Nous vous fournissons un algorithme de parcours en profondeur (non randomisé) qui pourra peut-être vous aider, et qui peut vous servir de comparaison au début.

Si vous le souhaitez, définissez des classes de graphes sur lesquels tester vos algorithmes. Nous en fournissons déjà quatre : les grilles, les graphes complets, les graphes aléatoires suivant le modèle d'Erdős-Rényi, et les *sucettes* (*lollipops*).

Comparez vos implémentations, en utilisant les outils que nous vous mettons à disposition. Ceux-ci comprennent :

- une classe `Labyrinth` qui permet d'afficher l'arbre couvrant d'une grille.
- une classe `RootedTree`, qui reconstruit l'arbre à partir d'une liste d'arêtes, sous une forme plus exploitable. On peut alors calculer plusieurs valeurs sur l'arbre.

La classe `Main` contient déjà tout le code nécessaire pour utiliser ces deux classes, mais n'hésitez pas à regarder ce qu'il y a dans chacune et y effectuer les améliorations qui vous semblent propices.

Les valeurs calculables par `RootedTree` sont les suivantes (notons que l'arbre est automatiquement enraciné en son centre) :

- La distribution de degrés, c'est-à-dire le nombre de sommets ayant degré d , pour d entre 0 et une borne d_{\max} donnée en argument.
- Le diamètre de l'arbre, c'est-à-dire la longueur du plus court chemin entre les deux sommets les plus éloignés.
- Le centre de l'arbre, le sommet qui minimise la distance au sommet le plus éloigné de lui.
- Le rayon, la plus grande distance entre le centre et un sommet.
- Le centroïde : si on le retire, les morceaux obtenus sont chacun plus petit que la moitié de l'arbre. On l'appelle aussi centre de gravité.
- L'excentricité moyenne : la distance moyenne d'un sommet au centre.
- L'indice de Wiener : la somme des distances entre chaque paire de sommet.
- La hauteur du sous-arbre dont la racine est un sommet donné en argument.
- La profondeur d'un sommet : sa distance à la racine.

Pour comparer les algorithmes, il suffit de calculer la moyenne de ces indices prise sur plusieurs arbres générés avec le même algorithme. Si les distributions de probabilités de deux algorithmes sont différentes, le plus souvent cela se traduira par des valeurs très différentes pour plusieurs de ces paramètres.

Il faut ici noter un phénomène peu intuitif : les arbres tirés uniformément aléatoirement vont tous avoir un diamètre (une excentricité, un indice de Wiener...) très proche les uns des autres. C'est simplement parce que presque tous les arbres couvrants ont effectivement un tel diamètre : les arbres avec un grand ou un petit diamètre sont en tout petit nombre comparé à eux. La probabilité de les tirer aléatoirement est donc négligeable. Ainsi, les arbres couvrants tirés uniformément vont tous se ressembler assez fortement. On peut faire une analogie avec les lancers de dés : en lançant un seul dé, on obtient les valeurs entre 1 et 6 avec la même probabilité. En lançant 2 dés, on obtient plus souvent 7 et rarement 2 ou 12. En lançant 1000 dés, on n'obtiendra quasiment jamais 1000 ou 6000, mais presque tout le temps une valeur très proche de 3500 (faites en l'expérience).

Bon travail !