Compléments d'informatique

Projet 1 : percolation

18 octobre 2018

Dans ce projet, on se propose d'étudier le phénomène de percolation, qui trouve des applications dans de nombreux domaines, allant de la science des matériaux à l'économie. En terme de programmation, ce projet vous permettra d'exercer les compétences suivantes : programmation modulaire, types de données abstraits, pointeurs de fonctions, et récursivité.

La section 1 décrit d'abord le modèle simplifié étudié et les questions auxquelles le programme à développer devra répondre. La section 2 décrit ensuite la structure du code proposée et les différentes fonctions qu'il vous faudra implémenter. Ce projet est à réaliser individuellement ou par groupe de deux étudiants maximum. Les modalités de soumission du projet via la plateforme sont décrites dans la section 3.

1 Théorie de la percolation

Soit une grille de taille $M \times N$ composée d'un ensemble de $M \times N$ cellules carrées disposées en M lignes et N colonnes. Chaque cellule est soit ouverte, soit fermée. Deux exemples de grille 20×20 sont représentées à la figure 1. Un cellule ouverte sera dite remplie si elle peut être connectée à une cellule ouverte sur la rangée supérieure de la grille via une chaîne de cellules ouvertes adjacentes les unes aux autres (c'est-à-dire se touchant à droite, à gauche, en dessous ou au dessus). On dira que le système constitué par cette grille *percole* s'il y a au moins une cellule remplie dans la rangée inférieure de la grille. Autrement dit, le système percole dès qu'il y a une chemin passant uniquement par des cellules ouvertes entre une cellule ouverte de la rangée supérieure et une cellule ouverte de la rangée inférieure. La figure 2 montre les grilles de la figure 2 dans lesquelles les cellules remplies ont été marquées en bleu. La grille de gauche ne percole pas alors que celle de droite percole. Ce système simple permet de modéliser différents phénomènes, tels que l'écoulement d'un liquide dans un matériau poreux (les cellules ouvertes représentant le vide dans le matériau par lesquelles le liquide peut s'écouler), la propagation d'un feu de forêt (les cellules vides représentant les arbres de la forêt par lesquels le feu peut se propager) ou encore la conductivé d'un

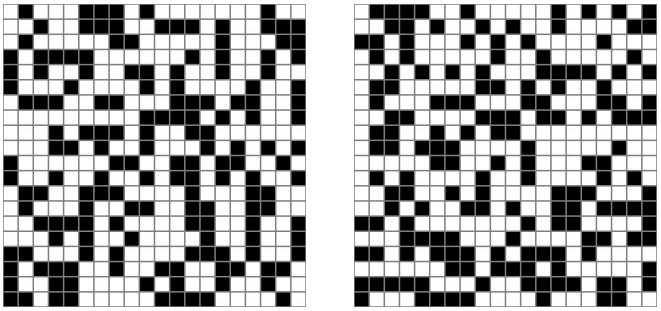

FIGURE 1 – Deux grilles 20×20 de densité $d = 0,6$. En blanc, les cellules ouverts, en noir, les cellules fermées.

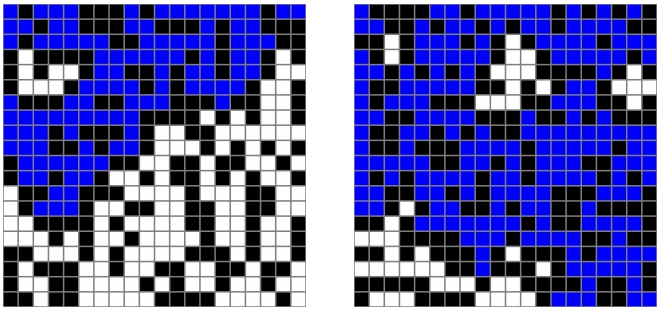

FIGURE 2 – Deux grilles 20×20 de densité $d = 0,6$. Les cellules bleus représentent les cases remplies. Seule la grille de gauche percole de part l'existence d'un passage de haut en bas.

matériau composite fait d'isolant et de métal (les cellules ouvertes représentant le matériau métallique par lequel le courant peut passer).

Une question qui intéresse les chercheurs est l'influence de la densité des cellules ouvertes sur la possibilité que le système percole, lorsqu'on suppose que la position des cellules ouvertes pour une densité donnée est purement aléatoire. Malgré la simplicité du système, il n'est pas possible de dériver une équation mettant en relation ces deux grandeurs et l'étude du phénomène doit passer obligatoirement par des simulations numériques sur ordinateur. Les scientifiques qui ont étudié la question ont observé qu'il existait un seuil critique en terme de densité (correspondant à une transition de phase) en dessous duquel la probabilité que le système percole est proche de 0 et au dessus duquel la probabilité que le système percole est proche de 1. On se propose dans le cadre de ce projet de vérifier ce résultat et d'estimer ce seuil critique de densité.

Évidemment, le fait qu'un système percole ne dépend pas que de la densité des cellules ouvertes mais également de la configuration précise de ces dernières qu'on supposera purement aléatoire. Pour estimer la probabilité qu'un système percole pour une valeur de densité $d \in [0, 1]$ donnée, on procédera de la manière suivante :

1. En partant d'une grille ne contenant que des cellules fermées, on choisit aléatoirement dMN cellules (sans remplacement) parmi les MN cellules initiales et on les ouvre.
2. On détermine si cette grille percole.
3. On répète les deux étapes précédentes un certain nombre T de fois et on calcule la proportion, notée $p_{M,N}(d)$, des T grilles qui ont percolé.

Cette proportion est une estimation de la probabilité réelle de percolation pour un système de taille $M \times N$ et de densité d. La théorie des probabilités et statistiques nous dit que cette estimation sera d'autant meilleure que la valeur de T sera grande, au prix évidemment de temps de calcul plus élevés. Dans ce projet, on fixera typiquement T à des valeurs au moins égales à 1000.

Dans le cadre du projet, on vous demande d'implémenter cette procédure d'estimation de $p_{M,N}(d)$ et d'ensuite tracer l'évolution de $p_{M,N}(d)$ en fonction de d afin de vérifier l'hypothèse de l'existence d'un seuil critique. Ensuite, on vous demande de produire une estimation de ce seuil critique qui sera déterminé comme la valeur de d pour laquelle la probabilité $p_{M,N}(d)$ est égale à 0.5. Il sera intéressant également de regarder l'impact du nombres de simulations T et des paramètres de taille, M et N, sur la courbe ainsi que sur le seuil critique.

2 Implémentation

Votre implémentation demandera la création de deux modules :
— Un module constitué des fichiers `grid.c` et `grid.h` implémentant un structure de type grille et les fonctions de manipulation associées
— Un module constitué des fichiers `function.c` et `function.h` implémentant les fonctions de tracé de courbe et de calcul de racine.

Le fichier `percolation.c` contiendra la fonction de calcul de la probabilité $p_{M,N}(d)$ et la fonction `main` qui effectuera le tracé de la courbe et le calcul de la racine. Le contenu attendu des différents fichiers est décrit ci-dessous.

Fichiers `grid.h` et `grid.c`

Ce module doit définir une structure abstraite de type `Grid` qui servira à contenir une grille. Les fonctions de manipulation de cette grille seront les suivantes :

`Grid *gridCreate(int n, int m, float d)` : crée une grille de taille $n \times m$ dont (exactement) $\lceil dnm \rceil$ cellules prises au hasard seront ouvertes et toutes les autres fermées. `d` est une valeur réelle comprise dans $[0, 1]$.

`void gridDestroy(Grid *g)` : libère la mémoire prise par la grille en argument.

`int gridGetWidth(Grid *g)` : renvoie la largeur de la grille.

`int gridGetHeight(Grid *g)` : renvoie la hauteur de la grille.

`int gridIsCellOpen(Grid *g, int i, int j)` : renvoie 1 si la cellule à la position (i, j) dans la grille est *ouverte*, 0 sinon.

`int gridIsCellClosed(Grid *g, int i, int j)` : renvoie 1 si la cellule à la position (i, j) dans la grille est *fermée*, 0 sinon.

`int gridIsCellFull(Grid *g, int i, int j)` : renvoie 1 si la cellule à la position (i, j) dans la grille est *remplie*, 0 sinon. Cette fonction ne renverra le résultat correct qu'une fois que la fonction `gridHasPercolated` aura été appelée sur la grille pour calculer les cellules remplies. Sinon, elle renverra faux pour toutes les cellules.

`int gridHasPercolated(Grid *g)` : renvoie 1 si la grille percole, 0 sinon. Une fois cette fonction appelée sur la grille, les cellules remplies auront été marquées comme tel et on pourra faire appel à la fonction `gridIsCellFull`.

`void gridPrint(Grid *g)` : affiche la grille sur la sortie standard sous un format texte. Une cellule fermée sera représentée par le caractère `#`, un case ouverte non remplie par une espace et une case ouverte remplie (lorsque la fonction `gridHasPercolated` aura été appelé) par le caractère `O`. On représentera des "murs" sur les bords gauche et droit par le caractère `|`. Voir un exemple à la figure 3.

`void gridPrintPPM(Grid *g, char *fileName)` : génére un fichier ppm représentant la grille graphiquement comme illustré sur les figures 1 et 2. Vous pouvez choisir la représentation graphique que vous souhaitez.

Les trois fonctions suivantes ne sont pas utiles directement pour le projet mais doivent être ajoutées pour nous permettre de corriger votre code. Elles ne seront appliquées que sur une grille obtenue en sortie de la fonction `gridCreate`.

`void gridSetOpen(Grid *g, int i, int j)` : marque la cellule (i, j) comme ouverte.
`void gridSetClosed(Grid *g, int i, int j)` : marque la cellule (i, j) comme fermée.
`void gridSetFull(Grid *g, int i, int j)` : marque la cellule (i, j) comme remplie.

Vous devez choisir vous-même la structure de données utilisées pour stocker une grille. Les deux fonctions les plus délicates à implémenter sont `gridCreate` et `gridHasPercolated`. Pour la première, il vous faudra en effet générer aléatoirement et sans remplacement une liste

```
|o##ooo##oo|
|oooo## #oo|
|ooooo# ##o|
|#o##ooo##o|
|ooooo## ##|
|##o##  ## |
|#oo# #    |
|#oooo# #  |
|#oooo# # #|
|oo#ooo#  #|
```

FIGURE 3 – Une représentation en texte d'une grille 10×10 qui percole

de *dnm* cellules à ouvrir[1]. Pour la seconde, le test de percolation devra passer d'abord par une marquage des cellules remplies. Nous vous suggérons de faire ce marquage en utilisant une fonction récursive.

Fichiers `function.h` et `function.c`

Ce fichier définira deux fonctions :

`void functionPlot(char *fileName, double (* f)(double), double xMin, double xMax, double minErrorX, double maxErrorX, double minErrorY)` :
qui enregistre dans un fichier une liste de paires $(x, \texttt{f}(x))$ permettant de tracer ensuite la courbe entre les points `xMin` et `xMax` avec le programme de votre choix.

`double functionGetVal(double (* f)(double), double fValue, double xMin, double xMax, double tolerance)` : renvoyant une valeur $x \in [\texttt{xMin}, \texttt{xMax}]$ tel que $\texttt{f}(x) = \texttt{fValue}$.

La fonction `functionPlot` devra générer un fichier nommé `fileName` qui contiendra sur chaque ligne une valeur de x et la valeur de $\texttt{f}(x)$ correspondante séparées par un simple espace. La fonction devra se charger de sélectionner les valeurs de x (qui devront inclure les bornes de l'intervalle `xMin` et `xMax`) auxquelles la fonction sera évaluée. Lorsque la fonction `f` connaît des changements brusque de valeurs sur l'intervalle considéré (ce qui sera le cas de la fonction étudiée dans ce projet), échantillonner des valeurs de x uniformémement n'est pas une bonne idée : si le pas d'échantillonnage est trop grand, les discontinuités ne seront pas visibles et si le pas d'échantillonnage est trop élévé, il faudra faire trop d'évaluation de la fonction même dans les régions où elle varie peu. La fonction `functionPlot` devra adapter automatiquement le pas d'échantillonnage de manière à placer plus de points dans les régions où la fonction connaît de grandes variations. Pour ce faire, les points $x \in [\texttt{xMin}, \texttt{xMax}]$ auxquels la fonction sera évaluée seront déterminés récursivement de la manière suivante

(en plus des points `xMax` et `xMin`) :

1. Si $|\texttt{xMax} - \texttt{xMin}| < \texttt{minErrorX}$, on n'ajoute pas de point supplémentaire (l'intervalle est suffisamment serré).
2. Sinon,
   (a) On échantillonne un nouveau point $\texttt{xMid} = \frac{\texttt{xMin}+\texttt{xMax}}{2}$ au milieu de l'intervalle.
   (b) Si $|\texttt{xMax} - \texttt{xMin}| \geq \texttt{maxErrorX}$ ou $|\texttt{f}(\texttt{xMid}) - \frac{\texttt{f(xMin)}+\texttt{f(xMax)}}{2}| > \texttt{minErrorY}$, on détermine récursivement de nouveaux points dans les intervalles $[\texttt{xMin}, \texttt{xMid}]$ et $[\texttt{xMid}, \texttt{xMax}]$.

La condition $|\texttt{xMax} - \texttt{xMin}| \geq \texttt{maxErrorX}$ permet d'assurer que la fonction sera évaluée en un minimum de point dans tous les cas. La condition $|\texttt{f}(\texttt{xMid}) - \frac{\texttt{f(xMin)}+\texttt{f(xMax)}}{2}| < \texttt{minErrorY}$ arrête l'échantillonnage lorsque le point $(\texttt{xMid}, \texttt{f}(\texttt{xMid}))$ est proche de la droite reliant $(\texttt{xMin}, \texttt{f}(\texttt{xMin}))$ et $(\texttt{xMax}, \texttt{f}(\texttt{xMax})$.

Pour implémenter la fonction `functionGetVal`, l'idée est d'appliquer la méthode de la bissection à la fonction $\texttt{f}(x) - \texttt{fVal}$. Pour rappel, la méthode de la bissection, qui suppose que $(\texttt{f}(\texttt{xMin}) - \texttt{fVal})(\texttt{f}(\texttt{xMax}) - \texttt{fVal}) < 0$, calcule la valeur $\texttt{f}(\texttt{xMid})$ au point $\texttt{xMid} = \frac{\texttt{xMin}+\texttt{xMax}}{2}$, remplace `xMax` par `xMid` si $\texttt{f}(\texttt{xMin}) * \texttt{f}(\texttt{xMid}) \leq 0$, `xMin` par `xMid` sinon, et ré-itére cette opération tant que $\texttt{xMax} - \texttt{xMin} > \texttt{epsilon}$. La valeur de la racine est alors $\frac{\texttt{xMin}+\texttt{xMax}}{2}$.

Fichiers `percolation.c`

Ce fichier contiendra la fonction `main`. Cette fonction devra d'abord récupérer trois arguments en ligne de commande [1] : la hauteur de la grille M, la largeur de la grille N, et le nombre T de grilles aléatoires à générer pour estimer $p_{M,N}(d)$. En utilisant la fonction `functionPlot`, elle générera ensuite un fichier appelé[2] `percolation-plot-M-N-T.txt` (où `M`, `N` et `T` seront remplacés par leurs valeurs) contenant les valeurs estimées de $p_{M,N}(d)$ pour des valeurs de d allant de 0.0 à 1.0 et, en utilisant `functionGetVal`, elle affichera sur la sortie standard la valeur de d tel que $p_{N,M}(d) = 0.5$. A vous de déterminer des valeurs appropriées des paramètres des fonctions `functionPlot` et `functionGetVal`.

En plus de la fonction `main`, ce fichier devra définir la fonction à passer en argument aux fonctions `functionPlot` et `functionGetVal`. Cette fonction calculera une estimation de $p_{M,N}(d)$ comme expliqué dans la section 1. Une difficulté potentielle est que la fonction `f` passée en argument par pointeur à `functionPlot` et `functionGetVal` ne doit dépendre que d'une seule variable, alors que la fonction qui estimera $p_{M,N}(d)$ devra dépendre en plus de d des valeurs de M, N, et T. La solution que nous vous préconisons est de passer ces valeurs à la fonction par le biais de variables globales dont les valeurs seront fixées dans la fonction `main` avant d'appeler `functionPlot` et `functionGetVal`.

1. Les instructions permettant de récupérer les arguments de la ligne de commande sont données dans le fichier `percolation.c` fourni avec l'énoncé.

3 Soumission

Le projet doit être soumis via la plateforme de soumission sous la forme d'une archive au format `zip` ou `tar.gz` contenant les fichiers suivants :

- `grid.h` et `grid.c`
- `function.h` et `function.c`
- `percolation.c`
- Un graphe (dans un fichier `graphe.pdf`) montrant les courbes de $P_{N,N}(d)$ générées par votre programme pour des valeurs de $N \in \{10, 25, 50, 100\}$.
- Le fichier `questions.txt` complété avec vos réponses aux différentes questions.

Vos fichiers seront compilés et testés sur la plateforme de soumission en utilisant le fichier `Makefile` fourni via la commande `make all` ou de manière équivalente en utilisant la commande suivante :

```
gcc -o percolation percolation.c grid.c function.c --std=c99 -lm
```

Votre programme devra alors s'utiliser comme suite :

```
./percolation 50 50 1000
```

En outre, nous utiliserons les flags de compilation habituels (`-pedantic -Wall -Wextra -Wmissing-prototypes`), qui ne devront déclencher aucun avertissement lors de la compilation, sous peine d'affecter négativement la cote. Toutes les soumissions seront soumises à un programme de détection de plagiat. En cas de plagiat avéré, l'étudiant (ou le groupe) se verra affecter une cote nulle à l'ensemble des projets.

Bon travail !

Notes

[1] La fonction `int rand()` de `stdlib.h` qui génère un entier aléatoirement entre 0 et `RAND_MAX` vous sera utile. Pour obtenir un entier aléatoire entre 0 et $k-1$, vous pouvez utiliser l'expression `rand()%k`.

[2] Si `n`, `m` et `t` sont des variables contenant les valeurs de N, M, et T, un chaîne de caractères contenant le nom du fichier peut être générée en utilisant la fonction `snprintf` de `<stdlib.h>`, par exemple :

```
char fileName[100];
snprintf(fileName, sizeof(fileName), "percolation-plot-%d-%d-%d.txt", m, n, t);
```