Projet algorithmique : SocialNetworkGraphs
IATIC4

Enseignant : George Manoussakis

Réda Guendouz Marwane Bahraoui

2021

Table des matières

Première partie

Introduction

Chapitre 1

Présentation brève du projet

Ce rapport a pour objectif de présenter notre projet d'implémentation et de modélisation d'algorithmes effectué en c++ durant notre deuxième année d'école d'ingénieur en informatique.

Nous avons reçu un ensemble d'algorithmes relié à la théorie des graphes. L'objectif principal qui nous a été confié est d'implémenter ces algorithmes en c++, un langage informatique, afin de bien pouvoir les exécuter.

Notre but final est donc de modéliser des graphes selon tous les principes reliés à la théorie des graphes afin de pouvoir ensuite manipuler nos graphes modélisés dans les algorithmes donnés en sujet.

Chapitre 2

Description de la class graph

2.1 Structure de données

Choix de la structure Nous avons choisi de créer une classe "graph" correspondant à un graphe selon la définition relié à la théorie des graphes. Grâce à cela et aux modèles objets que propose le C++, nous pourrons à notre tour manipuler des graphes non orientés comme indiqué dans le sujet avec des sommets numérotés à partir de 0. Nous considérerons donc tout objet graph comme un graphe qui respecte la théorie des graphes et avons choisi de stocker notre graphe (G(V,E)) sous forme de liste d'adjacence.

Listing 2.1 – structure de la classe graph

```
class graph
{
private:
    map<int, vector<int>> adjancyList;
    map<int, vector<int>> cliquesMax;
public:
    // constructor
    // methods...
};
```

Champ privé adjancyList Le champ principal permettant la manipulation de graphes de notre classe est un dictionnaire en c++ que l'on a nommé "adjancyList". Ce champ est donc de type "map" et a pour clé un sommet du graphe et pour valeur les voisins du dit sommet. Il y a donc autant de clé que dans notre "map" que de sommets dans le graphe.
Nous avons choisi la structure d'un dictionnaire car c'est celle qui correspond le plus pour nous à la structure d'une liste d'adjacence en c. En effet, l'utilisation de celle-ci nous permet un gain de temps considérable comparé à notre premier choix qui fut d'utiliser les tableaux classiques en c.
La liste des voisins correspondant à la valeur de notre dictionnaire est de type "vector<int>". Ce type nous permet d'ajouter, de supprimer et de parcourir aisément notre liste.

Champ privé cliquesMax Le second champ de notre classe est du même type que le premier mais contient des données différentes. En effet, comme peut le supposer son nom, nous stockons

dans chaque graphe, le nombre de cliques maximales afin de pouvoir faciliter notre accès à cette donnée.

méthodes usuelles : constructeurs/accesseurs...

1. accesseurs :
   (a) get_adjancyList() : accesseur du champ adjancyList
   (b) get_cliquesMax() : accesseur du champ cliquesMax
2. modifieurs :
   (a) set_cliquesMax() : accesseur du champ cliquesMax
3. add_vertice(entier x) : ajoute x sommets au graphe
4. add_edge(entier i, entier j) : ajoute l'arête (i,j) au graphe
5. delete_vertice(entier k) : supprime le sommet k et les arêtes liés du graphe

Deuxième partie

Rapport algorithmique

Chapitre 3

Algorithmes

Nous aborderons dans cette partie le côté théorique des algorithmes données en sujet que nous avons implémentés. Cette vision permettra de mieux comprendre notre code que l'on présentera dans la prochaine partie.

3.1 Présentation des algorithmes

algorithme 1 : random-graph Selon le sujet, le premier algorithme à réaliser est celui d'un générateur de graphe selon deux paramètres : le nombre de sommets et une probabilité p. L'entête de l'algorithme est simple : ajouter chaque arête possible entre les sommets avec la probabilité p.

algorithme 2 : barabasi-albert L'algorithme des graphes de Barabàsi-Albert est un peu plus complexe. Déjà, il possède une condition : le graphe doit posséder au moins 3 sommets avec au moins 1 arête. Il induit la construction d'un graphe selon un paramètre donnée "m" et le nombre de degrés des sommets.

algorithmes partie 2 & 3 Ces algorithmes sont déjà présents et bien définis dans les annexes du sujet : "papier1 & papier2". Nous avons bien analysés ces annexes du sujet afin de bien comprendre ces algorithmes.

3.2 Random-graph

Voici comment nous avons pensé le 1er algorithme de la première partie du sujet en sachant que les graphes sont numérotés de 0 à *nombre de sommets du graphe*.

Algorithm 1 random-graph

paramètre d'entrée: probabilité p : type flottant
foreach $i \in$ *sommets du graphe* **do**
  foreach $j \in$ *sommets du graphe* $\neq i$ **do**
    if *aléat(0,100)* <= p **then**
      ajouter arête (i,j)
    end
  end
end

3.3 barabasi-albert

Voici comment nous avons pensé le 2nd algorithme en sachant que les graphes sont numérotés de 0 à *nombre de sommets du graphe*.

Conditions initiales :

1. le graphe doit contenir au moins

Algorithm 2 barabasi-albert

paramètre d'entrée: degré m : type entier
$i \leftarrow 0$
proba $\leftarrow 0$
ajout_sommet()

while $m > 0$ ***and*** $i <$ *nombre sommets graphe - 1* **do**
  proba $\leftarrow$ *nombre_voisins*(i)/*degr_total_graphe*()
  if *aleat(0,100)* <= *proba* **then**
    ajouter arête (dernier sommet du graphe,i)
    $m \leftarrow m - 1$
  end
  $i \leftarrow i + 1$
end

3.4 Bron-kerbosch

Cet algorithme récursif retourne les cliques maximales, avec 3 ensemble de sommets disjoints :

1. P
2. X
3. R

et Γ (v) : l'ensemble des sommets voisins de v.
Au fur et à mesure de notre analyse, nous avons compris qu'il fallait parcourir une copie de l'ensemble P à la ligne 4.En effet, sans cette information trouvée, certaines sorties ne donnaient pas toutes les cliques maximales attendues. Ainsi en comprenant l'algorithme de mieux en mieux nous avons pu le coder de manière optimisé en c++.

proc BronKerbosch(P, R, X)

1: **if** $P \cup X = \emptyset$ **then**
2: report R as a maximal clique
3: **end if**
4: **for each** vertex $v \in P$ **do**
5: BronKerbosch($P \cap \Gamma(v)$, $R \cup \{v\}$, $X \cap \Gamma(v)$)
6: $P \leftarrow P \setminus \{v\}$
7: $X \leftarrow X \cup \{v\}$
8: **end for**

3.5 Bron-kerbosch dégénérescence

Cet algorithme a le même objectif et les mêmes entrées/sorties que l'algorithme bron-kerbosch "classique". Il diffère avec ce dernier dans la méthode utilisée. En effet, il utilise un tri préalable afin d'optimiser la complexité en temps de l'algorithme.

proc BronKerboschDegeneracy(V, E)

1: **for** each vertex v_i in a degeneracy ordering v_0, v_1, v_2, ... of (V,E) **do**
2: $P \leftarrow \Gamma(v_i) \cap \{v_{i+1}, \ldots, v_{n-1}\}$
3: $X \leftarrow \Gamma(v_i) \cap \{v_0, \ldots, v_{i-1}\}$
4: BronKerboschPivot(P, $\{v_i\}$, X)
5: **end for**

Trouver l'ordre de dégénérescence Nous avons créer notre propre algorithme pour trouver l'ordre de dégénérescence. Nous avons réfléchi en binôme et nous avons remarquer ceci : Si l'on crée une liste et que l'on met les les sommets dans l'ordre croissant de leur degré, qu'on retire un à un les sommets (du sommet avec le plus petit degré au sommet avec le plus haut degré) et qu'on actualise l'ordre des sommets a chaque suppression d'un sommet de ces degrés, on obtient un ordre de dégénérescence.

3.6 Manoussakis algortihme 1

Cet algorithme est présenté dans le sujet en partie 3 et dans le papier2 en annexe. Il permet de trouver les cliques maximales en décomposant le graphe G donnée en entrée.

Algorithm 1:

```
Data: A graph G.
Result: All the maximal cliques of G.
Compute k the degeneracy of G and σ_G.
Compute the degenerate adjacency lists of G.
Initialize T an empty generalized suffix tree.
for j = 1 to n do
    Compute all maximal cliques of graph G_j.
    for every maximal clique K of graph G_j do
        Order the vertices of K following σ_G
        Search for K in T.
        if there is a match then
            Reject it.
        else
            Insert the proper suffixes of K in T.
            Output K.
```

3.7 Manoussakis algortihme 2

Cet algorithme est quasiment le même que le précédent mais possède une complexité différente.

Algorithm 2:

```
for j = 1 to n do
    Compute all maximal cliques of graph G_j.
    for every maximal clique K of graph G_j do
        for every vertex x ∈ K do
            if any of its neighbors in G with lower rank than v_j in σ_G is adjacent to all the vertices in K then
                reject K.
            else
                output K.
```

3.8 Complexité temps / Complexité espace

random-graph L'ajout d'arête se réalise autant de fois qu'il y a de sommets (soit n) puis autant de fois qu'il y a de sommets - 1 (soit n-1). Donc, nous avons une complexité en temps de O(n * (n-1)) soit de $O(n^2)$.
Cet algorithme a une complexité en espace quasi-nulle sans compter le graphe étudié étant donnée qu'elle ne stocke que 4 variables : i, j, p et aléat() (=> car retourne une variable aléatoire).

barabasi-albert Notre algorithme comporte un boucle avec deux conditions, la condition $m > 0$ est d'une complexité infinie car la variable m n'est modifié qu'à travers une condition aléatoire. Donc dans le pire des cas, $m > 0$ sera toujours vrai. Ensuite, la seconde partie de la condition possède une complexité en temps de O(n) car on parcours tous les sommets du graphe.
Cet algorithme possède une complexité en espace équivalente au premier algorithme.

Troisième partie

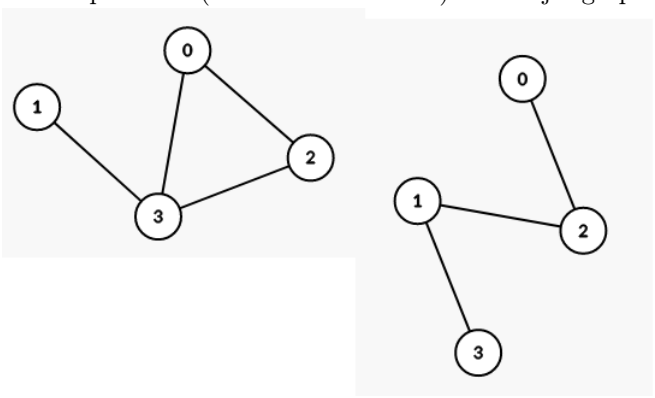
Tests des algorithmes

Chapitre 4

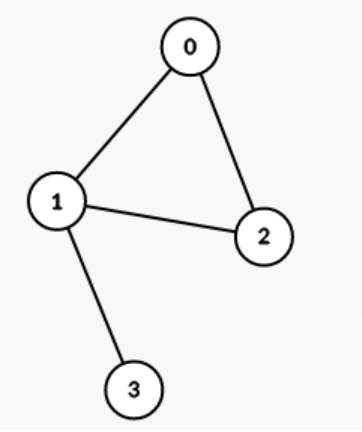
Exemples

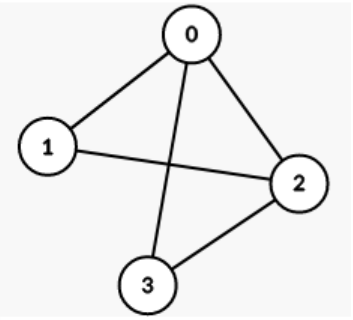
4.1 Partie 1 du sujet

algorithme : Random graph Selon notre algorithme codé en c++, nous pouvons avoir cela :
Entrée : 4 sommets, p = 60%
Sorties possibles (liste non-exhaustive) : un objet graph



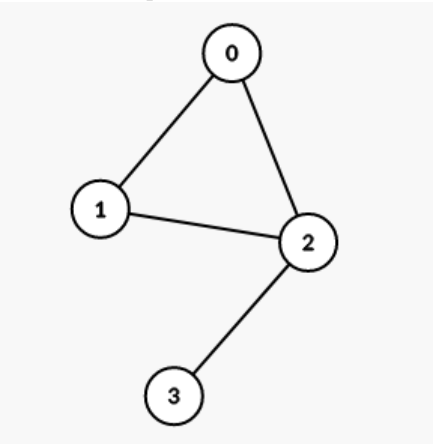
algorithme : Barabasì-Albert Selon notre algorithme codé en c++, nous pouvons avoir celas :
Entrée : graphe complet a 3 sommets, m = 2
Sorties possibles (liste non-exhaustive) :





4.2 Partie 2 du sujet

algorithme : Bron-Kerbosch Cet algorithme nous renvoie les cliques maximales sous forme de liste d'entiers correspondant aux sommets de ces cliques.
Entrée : Graphe suivant :

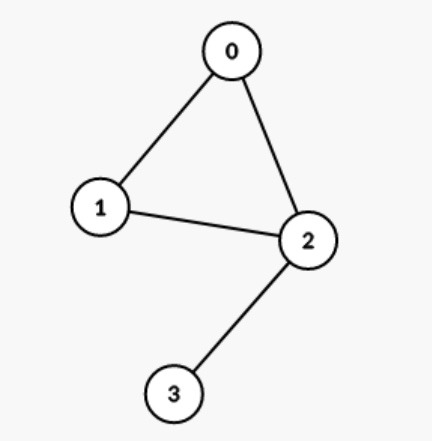


Sorties :

1. $[0, 1, 2]$
2. $[2, 3]$

4.3 Partie 3 du sujet

Manoussakis algorithme 1 Voici ce que nous sommes censés obtenir avec notre code :
Entrée Graphe suivant :



Sorties :

1. $[0, 1, 2]$
2. $[2, 3]$

Manoussakis algorithme 2 Les entrées/sorties de cet algorithme seront les mêmes que ceux de l'algorithme précédent.