CC1 d'Informatique
INF 431, Épreuve de 2H
Dominique Rossin

1 Introduction

Ce sujet propose d'étudier le graphe du Web à savoir le graphe orienté formé par les pages Web et où les arcs sont les liens reliant les pages les unes aux autres. Nous allons mettre en oeuvre différents algorithmes permettent de calculer des caractéristiques de ce graphe comme son diamètre ou encore son coefficient de clustering. Ces paramètres ainsi que la forme du graphe en terme de composantes fortement connexes sont autant d'éléments qui permettent à la fois de développer des algorithmes pour engendrer aléatoirement des graphes similaires au graphe du Web et d'autre part de proposer des algorithmes efficaces de parcours de ce graphe. En effet, la taille gigantesque du graphe nécessite la mise en oeuvre d'algorithmes spécifiques. Rappelons que début 2009, le nombre de sites Web atteignait 240 millions alors qu'il n'était que de 183 millions un an auparavant. Nous allons donc dans une première partie proposer une structure de données permettant de représenter le graphe du Web puis nous donnerons des algorithmes permettant de calculer certaines de ses caractéristiques.

Dans ce problème, nous aurons besoin d'utiliser les structures de données Java suivantes:

```
class LinkedList<E> {
  // Renvoie true si la liste
  // contient l'élément elt
  boolean contains(E elt);
  // Ajoute l'élément elt en
  // tête ou en queue de liste
  void addFirst(E elt);
  void addLast(E elt);
  // Supprime et renvoie
  // l'élément en tête/queue
  E removeFirst();
  E removeLast();
  // Renvoie l'élément
  // en tête ou en queue
  E getFirst();
  E getLast();
  // Taille de la liste
  int size();
}
```

```
class HashMap<K,V> {
  // Renvoie true si l element est
  // present en tant que cle / valeur
  boolean containsKey(K elt);
  boolean containsValue(V elt);
  // Ajoute l'élément elt associé
  // à la clé cle.
  void put(K cle, V elt);
  // Renvoie l'élément de clé cle.
  V get(K cle);
  // Liste des clés
  Set<K> keySet();
  // Liste des valeurs
  Collection<V> values()
}
```

Dans tout le problème, le paramètre `K` de la définition de HashMap sera `String`. La classe `String` définit convenablement la méthode `equals` bien connue ainsi que la méthode `hashCode()` que l'on pourra considérer comme efficace.

Afin de faciliter l'écriture des parcours de graphe, on suppose donnée une classe `Marquage` contenant les méthodes suivantes:

```
class Marquage {
  // Marque un élément
  void marque(String elt);
  // Supprime une marque
  void suppMarque(String elt);
  // elt est il marque ?
  boolean estMarque(String elt);
  // Renvoie l ensemble des elements marques
  Collection<String> marques();
```

```
}
```

Notez que:

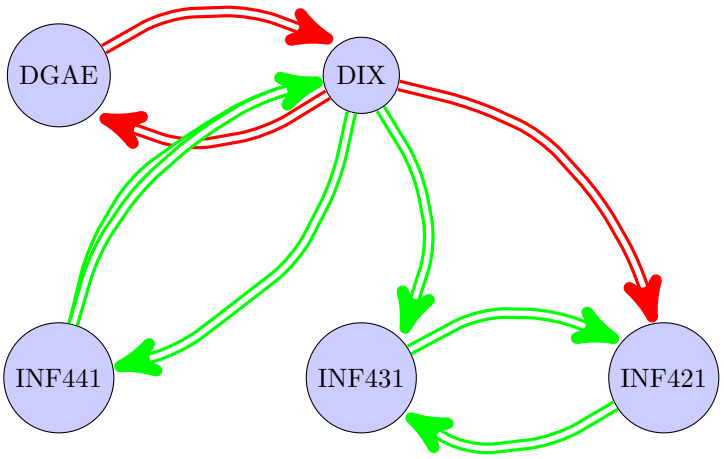
- `suppMarque` ne fait rien si l'élément passé en argument n'était pas marqué.
- `marque` ne fait rien si l'élément passé en argument était déjà marqué.
- `new Marquage()` permet de créer un nouvel ensemble de marquage initialement vide.

On supposera que toutes les méthodes de la classe `Marquage` sont réalisées en temps constant.

2 Construction du graphe orienté

Dans cette partie nous allons étudier une structure de données permettant de représenter le graphe orienté du Web. Une page web en sera un sommet et les arcs sont les liens vers les pages adjacentes. Par exemple la page web de la Direction de l'enseignement (DGAE) contient un lien vers la page du département d'informatique (DIX). Cette dernière procure des liens vers les pages web des cours INF431, INF441, INF421 et vers la page de la DGAE. La page de 441 pointe vers celle du DIX. La page de 431 vers celle de 421 et celle de 421 vers 431.

Question 1. Complétez le graphe afin de représenter l'ensemble des pages décrites précédemment.



En Java nous représenterons une page web par un objet de classe `PageWeb` contenant:

- l'URL de la page (adresse que vous tapez dans votre navigateur web, par exemple http://www.polytechnique.fr). Pour simplifier, on représentera cette adresse par une chaîne de caractères (*String*).
- un entier représentant le nombre de pages pointant vers elle (c'est le degré entrant du sommet)
- la liste des PageWeb pointées par la page (la liste des liens). On pourra pour cela utiliser la classe LinkedList de Java.

Question 2. Écrire en Java la définition de la classe PageWeb ainsi qu'un constructeur prenant l'adresse de la page en paramètre et initialisant une page web isolée (aucun lien vers cette page ou depuis cette page vers une autre page).

```
public class PageWeb {
  // question 2
  private final String url;
  private int degreEntrant;
  private final LinkedList<PageWeb> liens;
  public PageWeb(String s) {
    url = s;
    degreEntrant = 0;
    liens = new LinkedList<PageWeb>();
  }
```

Le graphe du Web est l'ensemble des pages Web et on utilisera pour cela la structure Java HashMap pour stocker l'association entre l'adresse des pages et les pages elles-mêmes représentées par la classe PageWeb précédement écrite. On supposera dans tout le problème que chaque page a sa propre adresse et réciproquement qu'à chaque adresse ne correspond qu'une page. De plus, le graphe ne possède pas d'arcs multiples c'est-à-dire qu'une page contient 0 ou 1 lien vers les autres pages.

Question 3. Écrire la classe GrapheWeb permettant de représenter un graphe comme décrit précédement. On munira cette classe d'un constructeur initialisant le graphe au graphe vide. Notez qu'il est inutile de définir une classe `Arc` comme dans le polycopié, la liste chaînée présente dans la classe `PageWeb` suffisant pour connaître ses voisins. Ainsi la classe `GrapheWeb` ne contient que l'ensemble des `PageWeb` indexées par leurs adresses.

```
public class GrapheWeb {
  // question 3
  private final HashMap<String, PageWeb> index;
  public GrapheWeb() {
    index = new HashMap<String, PageWeb>();
  }
}
```

Question 4. Donnez le code des méthodes suivantes (on lancera une erreur dans le cas où un des paramètres est incohérent):

- void ajouterPage(*String s*) dans la classe GrapheWeb. Cette méthode créera une nouvelle *PageWeb* d'adresse *s*.
- void ajouterLienSortant(*PageWeb destination*) dans la classe PageWeb qui ajoute un lien de la page courante vers la page `destination`.
- void ajouterLien(*String origine*, *String destination*) qui ajoute un lien (orienté) entre la page d'adresse origine et celle d'adresse destination.

```
  // question 4 Classe GrapheWeb
  public void ajouterPage(String s) {
    if (index.containsKey(s))
      throw new RuntimeException("la page " + s + " est deja présente");
    index.put(s, new PageWeb(s));
  }
  public void ajouterLien(String origine, String destination) {
```

```
  if (!index.containsKey(origine))
    throw new RuntimeException("la page origine " + origine
        + " n'est pas dans le graphe");
  if (!index.containsKey(destination))
    throw new RuntimeException("la page destination " + destination
        + " n'est pas dans le graphe");
  index.get(origine).ajouterLienSortant(index.get(destination));
}

// question 4 classe PageWeb
public void ajouterLienSortant(PageWeb destination) {
  if (liens.contains(destination)) // attention test en O(n)
    throw new RuntimeException("la page " + destination.url
        + " est deja presente dans les liens de " + url);
  liens.addLast(destination);
  destination.degreEntrant++;
}
```

3 Caractéristiques du graphe du Web

Nous allons donner dans cette partie des algorithmes permettant de calculer certaines caractéristiques des graphes comme le coefficient de clustering puis dans une section ultérieure nous nous servirons des résultats obtenus sur la forme particulière du graphe pour donner quelques exemples d'algorithmes approchés rapides.

Distribution de degrés La première caractéristique observée du graphe du web en a été sa distribution de degrés qui suit une loi de puissance c'est à dire que le nombre de pages N de degré sortant -resp. entrant- d est proportionnel à $d^{-\beta}$ où β est un coefficient positif. Notons que la valeur de ce paramètre n'est pas le même pour les degrés entrants et sortants (environ 2.15 pour les degrés entrants et 2.75 pour les degrés sortants). Afin de calculer ces coefficients, il faut connaître dans un premier temps la distribution des degrés.

Question 5. Écrire une méthode int [] distributionEntrante() qui calcule et renvoie un tableau tab contenant dans la case i le nombre de pages ayant degré entrant i.

```
// questions 5, 6 classe PageWeb (facultatif)
public int degreEntrant() {
  return degreEntrant;
}
public int degreSortant() {
  return liens.size();
}

// question 5 classe GrapheWeb
public int[] distributionEntrante() {
  int max = 0;
  for (PageWeb p : index.values())
    if (max < p.degreEntrant())
      max = p.degreEntrant();
  int[] t = new int[max + 1];
  for (PageWeb p : index.values())
```

```
    t[p.degreEntrant()]++;
  return t;
}
```

Coefficient de connexion (clustering) Ce coefficient code dans quelle mesure un sommet a ses voisins connectés entre eux. Le graphe des pages web diffère d'un graphe aléatoire en ce que deux pages pointées par une même page ont une probabilité assez forte d'être connectées entre elles. Mathématiquement parlant, le coefficient de clustering local C_i pour un sommet i est la proportion de paires ordonnées $(j,k), j \neq k$ de voisins du sommet i (il existe des arcs entre i et j et i et k) connectés entre eux (il existe un arc de j vers k). En d'autres termes on pose $C_i = \frac{E_i}{d_i(d_i-1)}$ où E_i est le nombre d'arcs reliant des sommets j et k tel que les arcs $i \to j$ et $i \to k$ existent. d_i est le degré sortant du sommet i.

Question 6. Écrire en Java dans la classe PageWeb une méthode double clustering() qui calcule et renvoie le coefficient de clustering local de cette page. Quelle est la complexité de votre méthode en fonction des degrés des sommets ?

```
// question 6
public double clustering() {
  Marquage<String> pages = new Marquage<String>();
  for (PageWeb j : liens)
    pages.marque(j.url);
  int c = 0;
  for (PageWeb j : liens)
    // O(d_i)
    for (PageWeb k : j.liens)
      // O(d_j) => \sum d_j
      if (pages.estMarque(k.url)) // O(1)
        ++c;
  if (degreSortant() < 2)
    return 0;
  else
    return (double) c / degreSortant() / (degreSortant() - 1);
}
```

On marque les voisins de la page puis on parcourt chaque voisin j et on compte le nombre de voisins de j qui sont aussi voisins de i c'est à dire qui sont marqués.

La complexité est donc en $d_i + \sum_{j \ voisin \ de \ i} d_j$

Diamètre du graphe Dans cette partie, nous allons considérer un utilisateur qui surfe sur le Web. Une question naturelle est de savoir si partant d'une page donnée, l'utilisateur peut atteindre une autre page donnée simplement en cliquant sur les différents liens. Notez que d'une page p on peut atteindre p par 0 clics.

Question 7. Écrire une méthode boolean peutAtteindre(PageWeb p) qui par un parcours en profondeur cherche si la page p est accessible depuis la page courante en suivant des liens de page en page.

```
// question 7
public boolean peutAtteindre(PageWeb p, Marquage<String> marques) {
  if (this == p)
    return true;
  if (marques.estMarque(p.url))
```

```
    return false;
  marques.marque(p.url);
  for (PageWeb pp : liens) {
    if (pp.peutAtteindre(p, marques))
      return true;
  }
  return false;
}
public boolean peutAtteindre(PageWeb p) {
  return peutAtteindre(p, new Marquage<String>());
}
```

On se propose maintenant de calculer le nombre de racines du Web, c'est-à-dire le nombre minimal de pages à partir desquelles tout le graphe est accessible.

Question 8. Soit R un ensemble de sommets (pages) tels que

- pour tout couple (v, v') $(v \neq v')$ de sommets de R , v' n'est pas accessible à partir de v.
- l'ensemble des sommets du graphe est accessible depuis R.

Montrez qu'il n'existe pas d'ensemble ayant les mêmes propriétés que R qui soit de cardinal strictement plus petit.

> Prenons deux ensembles R et R' vérifiant les conditions ci-dessus et tel que $|R| < |R'|$. Comme par définition, l'ensemble des sommets est accessible depuis R et R' ceux de R' sont accessibles à partir de ceux de R. Donc par le lemme des tiroirs et des chaussettes (pigeonhole) il existe deux sommets distincts x et y dans R' et z dans R tels que $z \to x$ et $z \to y$. De même z est accessible par les éléments de R' donc il existe t dans R' tel que $t \to z$. Par transitivité on a donc $t \to x$ et $t \to y$ mais t, x, y sont dans R' et comme x et y sont distincts R' n'est pas indépendant et contredit donc l'hypothèse de départ.

D'après la question précédente, il nous suffit donc de calculer un ensemble indépendant de racines couvrant tout le graphe. Pour cela nous allons utiliser l'algorithme suivant où $G = (S, A)$ est le graphe du Web avec S son ensemble de sommets (pages Web) et A son ensemble d'arcs (liens) :

- Soient $R \leftarrow \varnothing$ (ensemble des racines), $P \leftarrow \varnothing$ (ensemble des pages accessibles depuis R).
- Tant que $P \neq S$, faire
  - Soient $p \in S \setminus P$ et Acc_p l'ensemble des pages accessibles depuis p.
  - $R \leftarrow (R \setminus \{R \bigcap Acc_p\}) \bigcup \{p\}$
  - $P \leftarrow P \bigcup Acc_p$
- Retourner $|R|$

Question 9. Prouvez que l'algorithme précédent termine et calcule bien le nombre de racines du graphe du Web.

À chaque étape de la boucle Tant que on ajoute à l'ensemble P lensemble Acc_p qui contient au moins l'élément p (qui n'appartient pas à P). Donc P grossit strictement à chaque étape. Or $|P|$ est borné par le nombre de sommets donc l'algorithme termine. Pour montrer que R est bien un ensemble de racines, il suffit par la question précédente de montrer :

- que l'ensemble produit est indépendant. La ligne $R \leftarrow (R \setminus \{R \bigcap Acc_p\}) \bigcup \{p\}$ assure cette indépendance tout au long du processus. En effet, on remarque que à chaque étape l'élément p choisi n'est pas atteignable par l'ensemble R courant car p est pris dans $S \setminus P$. Par contre on peut effectivement atteindre des éléments de R en partant de p mais l'opération $R \leftarrow (R \setminus \{R \bigcap Acc_p\}) \bigcup \{p\}$ élimine ces éléments de l'ensemble racine.
- que l'ensemble permet d'atteindre tous les sommets du graphe. Attention, cela n'est pas immédiat, il faut vérifier que P est bien l'ensemble des sommets accessibles. En effet, la ligne $R \leftarrow (R \setminus \{R \bigcap Acc_p\}) \bigcup \{p\}$ retire des éléments de R et donc pourrait modifier les pages accessibles mais si le sommet x est retiré lors de cette opération alors cela veut dire que $p \rightarrow x$ et donc les pages qui étaient accessibles par x le sont aussi par p qui est rajouté à l'ensemble.

Question 10. Écrire une méthode int nombreRacines() qui calcule le nombre de racines du graphe du Web selon l'algorithme précédent.

Remarquons qu'il s'agit essentiellement de faire une DFS.

```
// question 10 dans GrapheWeb
public int nombreRacines() {
  Marquage<String> R = new Marquage<String>();
  Marquage<String> P = new Marquage<String>();
  for (String p : index.keySet())
    if (!P.estMarque(p)) {
      PageWeb.dfsMarques(index.get(p), R, P);
      R.marque(p);
    }
  return R.marques().size(); // ou R.cardinal();
}

// question 10 dans PageWeb
public static void dfsMarques(PageWeb p, Marquage<String> R,
    Marquage<String> P) {
  P.marque(p.url);
  for (PageWeb pp : p.liens) {
    R.suppMarque(pp.url);
    if (!P.estMarque(pp.url))
      dfsMarques(pp, R, P);
  }
}
```

On supposera dans le reste de la partie 3 que le graphe est fortement connexe c'est-à-dire que l'on peut aller de chaque sommet à n'importe quel autre en suivant des arcs. Nous verrons dans une question ultérieure que cette hypothèse n'est pas complètement absurde. Il est maintenant naturel de souhaiter connaître le nombre minimum de clics pour passer d'une page donnée à une autre page. Il s'agit donc de calculer le diamètre du graphe orienté.

Question 11. Écrire une fonction static int distance(PageWeb origine, PageWeb destination) qui calcule et renvoie la distance minimale (en nombre d'arcs) entre la page origine et la page destination.

Et maintenant c'est une BFS.

```
// question 11
public static int distance(PageWeb origine, PageWeb destination) {
  if (origine.url.equals(destination.url))
    return 0;
  HashMap<String, Integer> distance = new HashMap<String, Integer>();
  LinkedList<PageWeb> file = new LinkedList<PageWeb>();
  distance.put(origine.url, 0);
  file.addLast(origine);
  while (!file.isEmpty()) {
    PageWeb p = file.removeFirst();
    int d = distance.get(p.url) + 1; // rem. get ne peut pas retourner null
    for (PageWeb pp : p.liens) {
      if (pp.url.equals(destination.url))
        return d;
      if (!distance.containsKey(pp.url)) { // on utilise distance comme
        // marqueur
        distance.put(pp.url, d);
        file.addLast(pp);
      }
    }
  }
  return -1; // ne peut pas arriver si le graphe est fortement connexe
}
```

Question 12. Écrire une méthode int distanceMax() dans la classe PageWeb qui calcule et renvoie le maximum des distances minimales entre la page courante et les autres pages web. Quelle est la complexité de votre fonction ?

```
// question 12
public void distancesMax(HashMap<String, Integer> distances) {
  LinkedList<PageWeb> file = new LinkedList<PageWeb>();
  distances.put(this.url, 0);
  file.addLast(this);
  while (!file.isEmpty()) {
    PageWeb p = file.removeFirst();
    int d = distances.get(p.url) + 1; // rem. get ne peut pas retourner null
    for (PageWeb pp : p.liens) {
      if (!distances.containsKey(pp.url)) { // on utilise distances comme
        // marqueur
        distances.put(pp.url, d);
        file.addLast(pp);
      }
    }
  }
}
public int distanceMax() {
  HashMap<String, Integer> distances = new HashMap<String, Integer>();
  distancesMax(distances);
```

```
  int max = 0;
  for (int i : distances.values())
    // variante: for (Integer i:distance.values())
    if (max < i)
      max = i;
  return max;
}
```

La complexité est $\mathcal{O}(m)$ le nombre d'arêtes du graphe.

Question 13. Écrire finalement une méthode int diametre() qui calcule le diamètre du graphe du Web. Quelle est la complexité de votre fonction ?

```
// question 13
public int diametre() {
  int diametre = 0;
  for (PageWeb p : index.values()) {
    int d = p.distanceMax();
    if (diametre < d)
      diametre = d;
  }
  return diametre;
}
```

Complexité en $\mathcal{O}(nm)$, n est le nombre de sommets et m les arêtes.

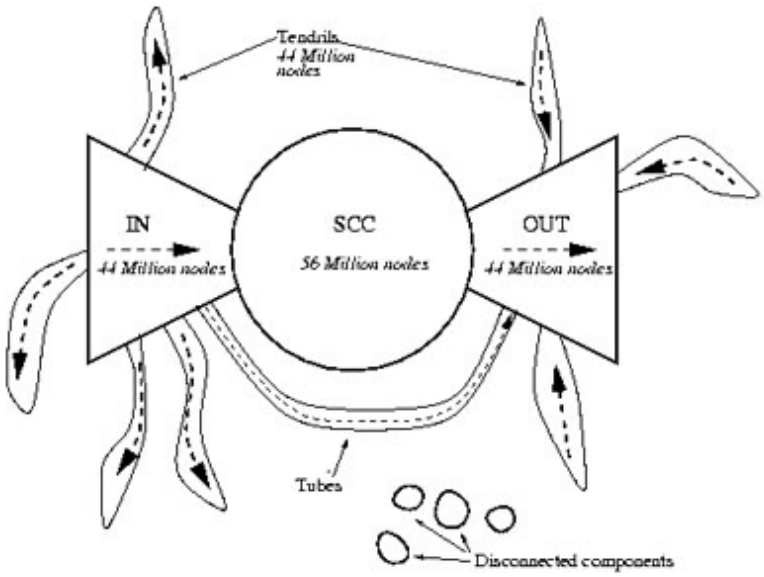
Question 14. Écrire de même une fonction double diametreMoyen() qui calcule et renvoie le diamètre moyen du graphe c'est à dire la moyenne des distances minimales entre les couples de sommets. Quelle en est la complexité ?

```
// question 14
public double diametreMoyen() {
  int somme = 0;
  for (PageWeb p : index.values()) {
    HashMap<String, Integer> distances = new HashMap<String, Integer>();
    p.distancesMax(distances);
    for (int i : distances.values())
      // variante: for (Integer i:distance.values())
      somme+= i;
   }
  return (double)somme/index.size()/index.size();
}
```

Complexité en $\mathcal{O}(nm)$, n est le nombre de sommets et m les arêtes.

4 Connexité du graphe du Web

Nous avons supposé précédement que le graphe du web était fortement connexe. En réalité, on peut *montrer* expérimentalement que le graphe a la forme suivante dite du noeud papillon:



Question 15. Sans écrire le code, décrire un algorithme permettant de calculer cette structure (identifier les sommets de SCC, IN, ...) sachant que SCC est la plus grosse composante fortement connexe. Votre algorithme devra être capable de distinguer les petites composantes non connectées, la composante connexe géante (SCC), les sommets faisant partie de IN, c'est-à-dire ceux à partir desquels on peut atteindre la composante SCC et ceux de OUT, c'est à dire ceux qu'on peut atteindre à partir de SCC mais d'où on ne peut atteindre SCC, les tubes et les tentrils. Les tubes contiennent les sommets appartenant à des chemins reliant IN à OUT mais ne passant pas par SCC. Les tentrils de IN (resp. OUT) contiennent les sommets que l'on peut atteindre à partir de IN (resp. d'où l'on peut atteindre OUT) mais d'où on ne peut atteindre ni SCC ni OUT. (resp. ni IN ni SCC)

L'algorithme est plutot simple.

On calcule les composantes connexes pour trouver les composantes deconnectées.

On utilise l'algorithme vu en cours pour calculer les composantes fortement connexes. La plus grosse contient les éléments de SCC. (Complexité $\mathcal{O}(m)$). Puis en partant des éléments de SCC, on fait des DFS en marquant au fur et à mesure les sommets. Les sommets visités sont alors dans OUT. (Complexité $\mathcal{O}(m)$).

Pour chaque sommet r non affecté, on fait une DFS jusqu a rencontré un sommet déjà dans une composante. Si on aboutit à un sommet x de SCC ou de IN alors on met tous les sommets sur le chemin entre r et x dans IN, si on aboutit à un sommet de OUT alors on met les sommets dans TENTRIL, autrement on les met dans IN.

Question 16. Quel est la complexité de l'algorithme précédent ?