CSC3101 - CF1

24 janvier 2024

Prénom Nom: _____

Lisez attentivement les instructions données ci-dessous, et vérifiez que vous avez bien toutes les pages (15 au total). Cet examen papier dure deux heures, et les documents sont autorisés. L'usage d'un appareil connecté (ordinateur, tablette, etc) est interdit. Vous pouvez répondre à chaque question dans l'espace dédié. Soyez sûr que votre réponse est lisible. Si vous avez besoin de davantage d'espace, vous pouvez utiliser le verso des feuilles ou demander du papier supplémentaire. Le cas échéant, merci de le préciser sur le recto. Cet examen est noté sur 20 points + 1 point de bonus. N'hésitez pas à sauter des questions si vous êtes bloqués. Vous pouvez répondre aux questions suivantes en utilisant les fonctions supposées implémentées aux questions précédentes (pensez à regarder les signatures).

Durant la période des fêtes de Noël, les ostréiculteurs du bassin d'Arcachon doivent envoyer très rapidement de grosses quantités d'huîtres à Paris. Cette année, par conscience écologique, ils décident d'utiliser le fret ferroviaire pour acheminer les précieuses denrées. Il y a cependant un problème : la ligne Bordeaux-Paris n'a pas un trafic suffisant pour tout acheminer assez rapidement ! Il va donc falloir emprunter d'autres routes afin d'optimiser le volume de marchandises transmises par heure. Par exemple, il devrait être possible d'utiliser les lignes Bordeaux-Poitiers-Paris, Bordeaux-Nantes-Paris, ou encore Bordeaux-Limoges-Paris.

Nous pouvons représenter notre problème sous forme d'un **réseau**. Un réseau est un **graphe orienté pondéré** (c'est-à-dire un graphe dans lequel les arêtes ont un poids) avec un nœud **source** (le nœud d'où partent les trains) et un nœud **puits** (le nœud sur lequel arrivent les trains). On suppose par la suite que tout graphe a un seul puits et une seules source. Les autres nœuds sont les gares intermédiaires possibles entre la source et le puits. Quant aux arêtes orientées, elles représentent les rails entre deux gares et le poids est le nombre de trains maximum par heure. Dans cet exercice, nous appellerons le poids d'une arête son **coût**. Dans notre problème, nous considérerons que ce coût est **un entier positif**. De plus, dans la suite du problème, nous supposerons qu'il n'y a pas d'arêtes anti-parallèles, c'est-à-dire que si un nœud A est relié à un nœud B, nous n'avons pas d'arête inverse entre B et A. Voici un exemple de représentation :

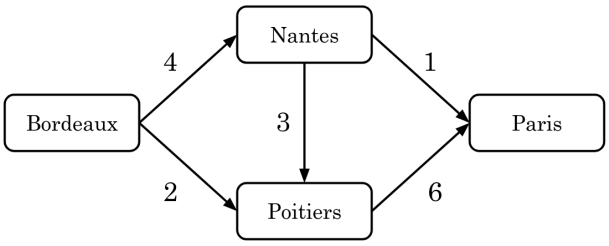

Figure 1: Exemple de représentation des connexions entre Bordeaux et Paris

Dans cet exemple, quatre trains par heure peuvent aller de Bordeaux à Nantes, un train par heure peut aller de Nantes à Paris, trois trains par heure peuvent aller de Nantes à Poitiers, deux trains par heures peuvent aller de Bordeaux à Poitiers, et six trains par heures peuvent aller de Poitiers à Paris.

Le but de cet examen est de trouver le nombre de trains maximum pouvant arriver à Paris par heure en régime continu (on ne considère pas le début de la livraison où les trains sont tous à Bordeaux) et de

donner le volume d'occupation des lignes. Par exemple, nous pourrions dire que la ligne Bordeaux-Nantes, qui a un volume d'occupation maximum de quatre trains par heure, ne verra circuler que trois trains en pratique.

L'ensemble des volumes d'occupation des arêtes est appelé un **flux**. Plus formellement, un flux est une fonction associant à chaque arête entre deux nœuds u et v un volume d'occupation $f(u, v)$ tel que $0 <= f(u, v) <= c(u, v)$ où $c(u, v)$ est le coût de l'arête entre u et v. Dit plus simplement, un flux peut se représenter comme un graphe où chaque arête doit avoir un coût plus faible que le coût original dans le réseau. Nous donnons ici un exemple de **flux non maximum** dans lequel deux trains circulent entre Bordeaux et Nantes, deux entre Nantes et Paris, un entre Bordeaux et Poitiers, et trois entre Poitiers et Paris.

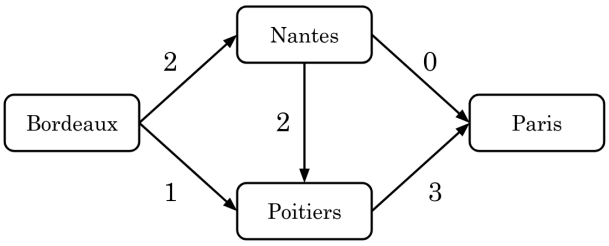

Figure 2: Exemple de flux entre Bordeaux et Paris

Pour un réseau donné avec un puits t, **la valeur d'un flux** f est la somme des flux entrants dans le puits t, i.e., $\Sigma_{u \text{ un nœud}} f(u, t)$. Dans l'exemple ci-dessus, la valeur du flux (le puits est Paris) est 3.

1 Préliminaires (3 points)

1. (1 point) Dans l'exemple donné dans la figure 1, combien de trains par heure au maximum peuvent arriver à Paris (c'est-à-dire, quel est le flux maximum) ? Quel est le volume d'occupation de **chaque ligne** permettant d'arriver à ce maximum en faisant circuler le minimum de trains ? Vous pouvez dessiner un diagramme comme dans la figure 2.

Solution: Le nombre maximum est 6. Il est atteint en faisant circuler 4 trains entre Bordeaux et Nantes, 2 entre Bordeaux et Poitiers, 3 entre Nantes et Poitiers, 1 entre Nantes et Paris, et 5 entre Poitiers et Paris.

2. (1 point) Dans une méthode *main* placée dans une classe *SNCF*, initialisez un tableau de chaînes de caractères nommé *cities* contenant les villes Bordeaux, Nantes, Poitiers et Paris.

Solution:

```
1 public class SNCF {
2     public static void main(String[] args) {
3         String[] cities = {"Bordeaux", "Nantes", "Poitiers", "Paris"};
4     }
5 }
```

3. (0.5 points) Dans cette méthode main, nous écrivons *System.out.println(cities);*. Notre programme compile. Est-ce que cette instruction va afficher le nom de toutes les villes dans *cities* dans le terminal ? Pourquoi ?

Solution: Nous n'affichons pas les villes, mais une référence sans signification particulière. Il n'y a pas de méthode *toString* pour un tableau. Pour afficher les villes, il faut faire *Arrays.toString*.

4. (0.5 points) Dans un graphe quelconque, la valeur du flux maximum est-elle bornée ? Si oui, par quelle valeur ?

Solution: Il s'agit du minimum entre la somme de tous les coûts entrants dans le puits et la somme de tous les coûts sortant de la source.

2 Graphe Orienté Pondéré (5 points)

Pour représenter un graphe orienté pondéré, nous allons utiliser une matrice d'adjacence. Une matrice d'adjacence A est une matrice carrée de taille de côté le nombre de nœuds dans le graphe, et telle que $A[i][j]$ représente le coût de l'arête allant du nœud i au nœud j. S'il n'y a pas de connexion entre i et j, nous considérons le coût comme étant égal à 0. Par exemple, un graphe avec deux nœuds tel que le nœud 0 est relié au nœud 1 avec une arête de coût 2 (mais 1 n'est pas relié à 0 car les arêtes sont orientées) serait représenté par :

$$\begin{bmatrix} 0 & 2 \\ 0 & 0 \end{bmatrix}$$

Comme la matrice d'adjacence a besoin d'utiliser des index, nous allons associer à chaque ville un identifiant unique. Bordeaux correspondra à 0, Nantes à 1, Poitiers à 2, et Paris à 3.

1. (0.5 points) Quelle structure de données vous semble la plus appropriée pour associer un nom de ville à un nombre ?

   Solution: Une table d'association (c.à.d., une Map/HashMap).

2. (0.5 points) À partir de maintenant, nous supposons que nous avons cette structure de données pour facilement passer d'un nom de ville à un index. Nous ne travaillerons plus qu'avec des index. Donnez la matrice d'adjacence de l'exemple de la Figure 1 où l'index des villes est tel que décrit plus haut.

   Solution: $\begin{bmatrix} 0 & 4 & 2 & 0 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 \end{bmatrix}$

3. (1 point) Créez une classe nommée *WeightedGraph*. Dans cette classe, nous avons un champ unique et privé *adjacencyMatrix* qui sera un tableau en deux dimensions d'*int*. Puis, ajoutez un constructeur public prenant en entier un paramètre *int nbNodes* représentant le nombre de nœuds dans notre graphe et qui initialise une matrice d'adjacence vide (i.e. remplie de 0).

Solution:

```
1  public class WeightedGraph {
2
3      private int[][] adjacencyMatrix;
4
5      public WeightedGraph(int nbNodes){
6          adjacencyMatrix = new int[nbNodes][nbNodes];
7      }
8  }
```

4. (0.5 points) Écrivez une méthode *public int getCost(int from, int to)* retournant le coût de l'arête allant de *from* à *to*.

Solution:

```
1  public int getCost(int from, int to){
2      return this.adjacencyMatrix[from][to];
3  }
```

5. (0.5 points) Écrivez une méthode *public boolean isEdge(int from, int to)* retournant s'il y a une arête entre *from* et *to*.

Solution:

```
1 public boolean isEdge(int from, int to){
2     return this.getCost(from, to) > 0;
3 }
```

6. (1 point) Comme un coût ne peut pas être négatif, nous allons lever une exception si l'utilisateur essaie de créer une arête de coût négatif. Créez une nouvelle exception *NegativeCostException*. Elle devra contenir un constructeur *NegativeCostException(int source, int target, int cost)* qui fera en sorte d'initialiser le message de l'exception à *A negative cost of **cost** between **source** and **target** was created.* (les variables sont en gras).

Solution:

```
1 public class NegativeCostException extends Exception {
2     public NegativeCostException(int source, int target, int cost){
3         super("A negative cost of " + cost +
4                 " between " + source + " and " + target + " was created.");
5     }
6 }
```

7. (1 point) Écrivez une fonction *public void addEdge(int from, int to, int cost)* levant l'exception précédente si le coût est négatif ou nul ou rempli la matrice d'adjacence pour avoir un coût *cost* entre *from* et *to*.

Solution:

```
1  public void addEdge(int from, int to, int cost) throws NegativeCostException {
2      if (cost <= 0){
3          throw new NegativeCostException(from, to, cost);
4      }
5      this.adjacencyMatrix[from][to] = cost;
6  }
```

3 Réseau (2 points)

1. (1.5 points) Comme mentionné plus haut, un réseau est un graphe orienté pondéré avec un nœud source et un nœud puits. Créez une classe *Network* qui étend *WeightedGraph*. Dans cette classe, placez deux champs privés *int source* et *int sink* représentant la source et le puits. Enfin, créez un constructeur *public Network(int source, int sink, int nbNodes)* initialisant la matrice d'adjacence, la source et le puits.

Solution:

```
1  public class Network extends WeightedGraph {
2
3      private int source;
4      private int sink;
5
6      public Network(int source, int sink, int nbNodes){
7          super(nbNodes);
8          this.source = source;
9          this.sink = sink;
10     }
11 }
```

2. (0.5 points) Écrivez les getters *public int getSource()* et *public int getSink()* retournant les valeurs des champs correspondants.

Solution:

```
1  public int getSource(){
2      return source;
3  }
4
5  public int getSink(){
6      return sink;
7  }
```

4 Flux (5 points)

Dans les questions précédentes, nous nous sommes efforcés de représenter nos données avec des structures idoines. À partir de maintenant, nous allons nous concentrer sur le calcul du flux maximum.

Nous pouvons encoder un flux par un graphe orienté pondéré où la valeur des arêtes correspond à la valeur du flux. C'est ce que nous avons fait dans la figure 2. Dès lors, nous pouvons créer une classe *Flow* héritant de *WeightedGraph* et ayant pour constructeur *public Flow(Network network, int nbNodes)*. Nous ne demandons pas d'écrire ce code d'initialisation. Notez simplement que *Flow* possède un champ privé *Network network* accédant au réseau sous-jacent au flux et qu'il a toujours accès à *adjacencyMatrix* de *WeightedGraph*.

1. (0.5 points) En réalité, si vous avez bien créé *adjacencyMatrix* comme expliqué plus haut à la question 2.3, nous n'avons pas accès à *adjacencyMatrix*. Pourquoi et comment rendre ce champ accessible dans *Flow* (et *Network* au passage).

Solution: L'erreur vient du fait que *adjacencyMatrix* est privé. Nous pouvons le passer *protected*.

2. (1 point) Écrivez une fonction *public int getFlowValue()* donnant la valeur du flux au puits. Comme illustré plus haut, la valeur de ce flux est la somme des coûts des arêtes entrantes dans le puits t, i.e., $\Sigma_{u \text{ un nœud}} f(u,t)$.

Solution:

```
1  public int getFlowValue(){
2      int value = 0;
3      for (int i = 0; i < this.adjacencyMatrix.length; i++){
4          value += this.getCost(i, network.getSink());
5      }
6      return value;
7
8  }
```

3. (0.5 points) Dans un flux f, la *valeur d'exploitation* d'une arête entre les nœuds u et v est définie par $f(u,v) - f(v,u)$. Notez que cette valeur est aussi définie quand l'arête de u à v n'existe pas dans le réseau original. Dans ce cas, elle représente de combien on pourrait réduire le coût. Écrivez une méthode *public int getExploitationValue(int from, int to)* retournant cette valeur.

Solution:

```
1  public int getExploitationValue(int from, int to){
2      return this.getCost(from, to) - this.getCost(to, from);
3  }
```

4. (0.5 points) Écrivez une méthode *public void addFlow(int from, int to, int value)* ajoutant *value* à *f(from, to)*.

Solution:

```
1  public void addFlow(int from, int to, int value){
2      this.adjacencyMatrix[from][to] += value;
3  }
```

5. (2 points) Pour qu'un flux soit valide, il doit vérifier les propriétés suivantes :

1. Pour toute arête entre u et v, $0 <= f(u,v) <= c(u,v)$ où $c(u,v)$ est le coût de l'arête entre u et v et $f(u,v)$ est le coût du flux entre u et v.
2. Pour tout nœud u différent de la source et du puits, la somme des coûts entrants du flux doit être égale à la somme des coûts sortants du flux, c'est-à-dire $\Sigma_{u \text{ un nœud}} f(u,n) = \Sigma_{u \text{ un nœud}} f(n,u)$.

Écrivez une méthode *public boolean isValidFlow()* vérifiant si le flux est valide.

Solution:

```
1  public boolean isValidFlow(){
2      for (int i = 0; i < this.adjacencyMatrix.length; i++){
3          if (i == network.getSink() || i == network.getSource()) continue;
4          int totalOut = 0;
5          int totalIn = 0;
6          for (int j = 0; j < this.adjacencyMatrix.length; j++){
7              if (this.adjacencyMatrix[i][j] < 0 ||
8                      this.adjacencyMatrix[i][j] > network.getCost(i, j)){
9                  return false;
10             }
11             totalOut += this.adjacencyMatrix[i][j];
12             totalIn += this.adjacencyMatrix[j][i];
13         }
14         if (totalIn != totalOut) return false;
15     }
16     return true;
17 }
```

6. (0.5 points) Nous définissons le coût résiduel d'une arête comme étant le coût de l'arête dans le réseau moins le coût d'exploitation de cette arête comme défini à la question précédente. Instinctivement, cette valeur nous donne les occupations restantes dans notre réseau. Par exemple, les coûts résiduels du réseau de la figure 1 avec le flux de la figure 2 sont illustrés à la Figure 3.

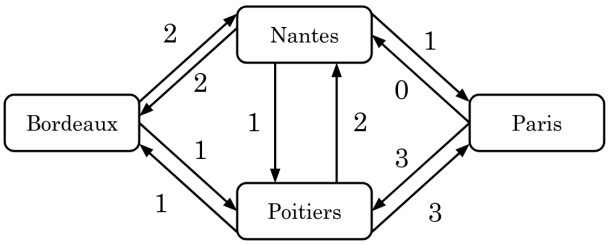

Figure 3: Graphe des coûts résiduels

Écrivez une méthode *public int getResidualCost(int from, int to, Flow flow)* que nous placerons dans *WeightedGraph* et retournant ce coût résiduel.

Solution:

```
1 public int getResidualCost(int from, int to, Flow flow){
2     return this.adjacencyMatrix[from][to] - flow.getExploitationValue(from, to);
3 }
```

5 L'algorithme de Ford-Fulkerson (6 points)

1. (1 point) Nous donnons l'algorithme de parcours de graphe suivant que nous plaçons dans la classe *Network*. Il permet de trouver un **chemin simple** dans un réseau :

```
1  public List<Integer> findSimplePath(Flow flow){
2      Set<Integer> visited = new HashSet<>();
3      Stack<Integer> toProcess = new Stack<>();
4      Map<Integer, Integer> previous = new HashMap<>();
5      toProcess.add(source);
6      visited.add(source);
7      while (!toProcess.isEmpty()){
8          int current = toProcess.pop();
9          if (current == sink) break;
10         for (int j = 0; j < this.adjacencyMatrix.length; j++){
11             if (!visited.contains(j) && getResidualCost(current, j, flow) > 0){
12                 toProcess.add(j);
13                 visited.add(j);
14                 previous.put(j, current);
15             }
16         }
17     }
18     if (!previous.containsKey(sink)) return null;
19     List<Integer> res = new ArrayList<>();
20     int current = sink;
21     res.add(sink);
22     while (current != source){
23         current = previous.get(current);
24         res.add(0, current);
25     }
26     return res;
27 }
```

Expliquez simplement ce que fait cet algorithme.

Solution: Cet algorithme cherche un chemin quelconque qui ne contient aucun nœud en double entre la source et le puits et le retourne. S'il n'y a pas de chemin, nous retournons null.

2. (1 point) Quelle est la complexité de cet algorithme en fonction du nombre de nœuds ? Pourquoi ?

Solution: Elle est quadratique en le nombre de nœuds N. En effet, la boucle *while* se répète au maximum N fois et elle contient une autre boucle se répétant N fois.

3. (1 point) La méthode *findSimplePath* retourne un chemin dans le graphe représenté par une suite de nœuds $a_0, a_1, ..., a_N$. Par exemple, il pourrait retourner le chemin Bordeaux-Nantes-Paris. Écrivez une méthode *public int getMinResCostPath(List<Integer> path, Flow flow)* qui sera placée dans *WeightedGraph* et retournant le coût résiduel (défini plus haut) minimal sur le chemin $c_r(a_i, a_{i+1})$, c'est-à-dire que pour tout $j \neq i$, $c_r(a_i, a_{i+1}) \leq c_r(a_j, a_{j+1})$. Par exemple, dans la figure 3, le coût résiduel minimal sur le chemin Bordeaux-Nantes-Paris est 1.

   Nous rappelons que nous pouvons utiliser la méthode *size()* sur une liste pour obtenir sa taille et *get(i)* pour obtenir la valeur à l'index i.

Solution:

```
1  int getMinResCostPath(List<Integer> path, Flow flow){
2      int mini = Integer.MAX_VALUE;
3      for (int i = 0; i < path.size() - 1; i++){
4          int edgeValue = this.getResidualCost(path.get(i), path.get(i + 1), flow);
5          if (edgeValue <= mini){
6              mini = edgeValue;
7          }
8      }
9      return mini;
10 }
```

4. (1.5 points) L'algorithme de Ford-Fulkerson permet de trouver le flux maximum dans un réseau. Il est donné par l'algorithme suivant :

   1. $f \leftarrow$ Flux vide
   2. Tant qu'il y a un chemin simple $P = a_0, a_1, ..., a_n$:
      (a) $minResCost \leftarrow$ Coût résiduel minimal sur le chemin P
      (b) Pour toute arête (a_i, a_{i+1}) sur le chemin P :
         i. Si (a_i, a_{i+1}) est une arête du réseau, alors $f(a_i, a_{i+1}) += minResCost$.
         ii. Sinon, $f(a_{i+1}, a_i) -= minResCost$

   Implémentez l'algorithme de Ford-Fulkerson dans une méthode *public Flow getMaxFlow()* placée dans *Network*.

Solution:

```
1  public Flow getMaxFlow(){
2      Flow flow = new Flow(this, this.adjacencyMatrix.length);
3      List<Integer> path = this.findSimplePath(flow);
4      while (path != null){
5          int min_value = this.getMinResCostPath(path, flow); // O(N)
6          for (int i = 0; i < path.size() - 1; i++){ // O(N)
7              // 1.
8              if (this.isEdge(path.get(i), path.get(i + 1))) {
9                  flow.addFlow(path.get(i), path.get(i + 1), min_value);
10             } else {
11                 flow.addFlow(path.get(i + 1), path.get(i), -min_value);
12             }
13         }
14         path = this.findSimplePath(flow); // O(N^2)
15     }
16     return flow;
17 }
```

5. (1 point) Quelle est la complexité de l'algorithme de Ford-Fulkerson tel qu'implémenté ici ? Pourquoi ? Vous pourrez l'exprimer en fonction du nombre de nœuds et de la valeur maximale du flux. Pour cela, il vous faudra montrer que la valeur du flux (au puits) augmente de 1 à chaque itération dans le pire des cas.

Solution: La valeur du flux augmente de 1 à chaque itération de la boucle *while*. En effet, la dernière arête du chemin est toujours une arête du réseau. Donc la valeur du flux augmente au moins de 1. On a donc une complexité de $\mathcal{O}(f_{max} * N^2)$, car *findSimplePath* est quadratique.

6. (0.5 points) Comment pourrait-on généraliser simplement cet algorithme dans le cas où nous avons plusieurs sources et plusieurs puits ?

Solution: Nous pourrions nous réduire à notre ancien problème en créant un nouvel unique nœud source relié aux anciennes sources avec ces coûts infinis. Idem pour les puits.