Université d'Ottawa
Faculté de génie

École d'ingénierie et de
technologie de l'information

uOttawa
L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information
Technology and Engineering

Exercice de programmation 1 (15%) CSI2510

Date de remise : 13 Novembre, 23:59

Politique de retard: ***1min-24hres de retard -30% pénalité; devoirs non acceptés après 24hres.***

Le problème des mariages stables

Description du Problème

Pour ce devoir, on vous demande de programmer une solution au problème dit des mariages stables. Dans notre cas, le problème consiste à associer des étudiants avec des employeurs. Afin de simplifier ce problème nous allons considérer le cas ou n employeurs désirent embaucher n étudiants; chaque employeur embauchant un seul étudiant. Chaque employeur doit donc classer les étudiants en ordre de préférence (de 1 à n) et les étudiants font de même. La solution que nous recherchons associe chaque employeur avec un étudiant de façon que tous sont aussi satisfaits quepossible.

Vous aurez donc une liste d'employeurs (numéroté de 0 à n-1), par exemple:

```
0. Thales
1. Canada Post
2. Cisco
```

Et une liste d'étudiants, par exemple:

```
0. Olivia
1. Jackson
2. Sophia
```

Ces deux listes auront toujours la même taille.

Employeurs et étudiants auront établis leur préférence, par exemple :

- Préférence des employeurs :

```
0. Thales:        0.Olivia, 1.Jackson, 2.Sophia
1. Canada Post:   2. Sophia, 1.Jackson, 0.Olivia
2. Cisco:         0.Olivia, 2.Sophia, 1.Jackson
```

- Préférences des étudiants:

```
0. Olivia:   0.Thales, 1.Canada Post, 2.Cisco
1. Jackson:  0.Thales, 1.Canada Post, 2.Cisco
2. Sophia:   2.Cisco, 0.Thales, 1.Canada Post
```

Ces préférences vous seront données dans un fichier texte spécialement formatté.

Une solution stable à ce problème pourrait être comme suit:

```
Thales - Olivia
Canada Post - Jackson
Cisco - Sophia
```

Pourquoi cette solution est-elle stable? Un mariage est stable s'il n'existe aucun autre mariage pour lequel l'échange d'un membre avec un membre de l'autre mariage rendrait tous les membres plus Satisfaits. Par exemple, dans la solution ci-dessus Canada Post aurait préféré embaucher Olivia au lieu de Jackson mais Olivia est appariée à Thales qu'elle préfère à Canada Post. Jackson lui aurait préféré travailler pour Thales mais Thales préfère Olivia. En conséquence, bien que ni Canada Post ou Jackson ont obtenu leur premier choix, le mariage est tout de même stable car il n'est pas possible de faire un échange qui améliorerait la situation. Le mariage stable est parfait lorsque toutes les parties ont été appariées.

Algorithme à réaliser

Le problème des mariages stables peut être résolu à l'aide de l'algorithme de Gale-Shapley. Un pseudo-code réalisant cet algorithme vous est donné ci-bas. L'entrée de cet algorithme comprend une liste d'employeurs, une liste d'étudiants et une matrice de préférences de dimension `nxn`.

Algorithme Gale-Shapley

Entrée:

- Une liste de `n` employeurs, indexés de `0` à `n-1`
- Une liste de `n` étudiants, indexés de `0` à `n-1`
- Une matrice de `nxn` dans laquelle chaque élément est une paire (*employeur_ranking*, *student_ranking*); (chaque préférence étant dans l'intervalle [1,n]). Une rangée de cette matrice correspond à l'employeur ayant l'index correspondant. Une colonne correspond à un étudiant. Ainsi si la paire située dans la rangée `r` et la colonne `c` contient la paire `i,j` alors ceci veut dire que l'employeur `r` à classé l'étudiant `c` à la position `i` et l'étudiant `c` a classé l'employeur `r` à la position `j`.

Initialisation:

- Créer la pile `Sue` contenant les employeurs non appariés
- Empiler tous les employeurs dans cette pile en débutant par l'employeur `0`

- Créer deux tableaux de dimension `n`, `students` et `employers`, utilizes afin de représenter l'appariement; (si l'étudiant `s` a été apparié avec l'employeur `e` alors `students[s]=e` et `employers[e]=s`; la valeur `-1` désigne un employeur ou un étudiant non apparié
- Initialiser les éléments de ces tableaux à `-1`

- Créer une matrice 2D `A` de dimension `nxn` avec `A[s][e]` étant le rang donné par l'étudiant `s` à l'employeur `e`
- Initialiser chaque entrée de cette matrice avec les *student rankings*.

- Créer `n` file à priorité avec `PQ[e]` étant la file de l'employeur `e`
- `For each student s`
  - `For each employer e`
    - `PQ[e].insert(employer_ranking,s)`

Procédure:

- `while ( !Sue.empty() )`
  - `e= Sue.pop()  // e is looking for a student`
  - `s= PQ[e].removeMin()  // most preferred student of e`
  - `e'= students[s]`
  - `if (students[s] == -1) // student is unmatched`
    - `students[s]= e`
    - `employers[e]= s  // match (e,s)`
  - `else if (A[s][e] < A[s][e']) // s prefers e to employer e'`
    - `students[s]= e`
    - `employers[e]= s  // Replace the match`
    - `employers[e']= -1 // now unmatched`
    - `Sue.push(e')`
  - `else s rejects offer from e`
    - `Sue.push(e)`
- `return the set of stable matches`

Voir l'annexe en fin de ce document pour un déroulement complet de l'algorithme sur l'exemple donné plus haut.

Input file format

Un fichier texte vous sera donné en entrée. Le premier nombre de ce fichier sera le nombre d'employeurs et d'étudiants. Puis la liste des employeurs et des étudiants sera donnée. Finalement, la matrice de préférences sera fournie sous forme de pairs (*employer ranking, student ranking*). Chaque rangée correspond à un employeur et chaque colonne correspond à un étudiant. Ainsi pour l'exemple donné plus haut, ce fichier serait :

```
3
Thales
Canada Post
Cisco
Olivia
Jackson
Sophia
1,1 2,1 3,2
3,2 2,2 1,3
1,3 3,3 2,1
```

Output file format

Votre programme doit simplement produire un fichier texte contenant la solution. Si le fichier d'entrée se nomme ABC.txt alors le fichier de sortie doit être nommé `matches_ABC.txt`. Ce fichier contient la liste des paires `employers - students` données dans l'ordre des employeurs listé dans le fichier d'entrée.

```
Match 0: Thales - Olivia
Match 1: Canada Post - Jackson
Match 2: Cisco - Sophia
```

Le fichier de sortie doit avoir le format montré dans lequel les paires sont listés de `0` à `n-1` (paire `i` étant l'appariement de l'employeur `i`). Les noms des employeurs et des étudiants doivent être séparés par un tiret (`-`). Si vous ne vous conformez pas à ce format, votre solution ne sera pas évaluée.

Exigences

- Vou devez écrire un programme réssolvant le problème des mariages stables en suivant l'algorithme Gale-Shapley tel que décrit ci-haut. Vous devez utiliser les structures de données et les procédures décrites. Vous ne pouvez pas proposer votre propre variante de l'algorithme.
- Vous devez créer une classe appelée `GaleShapley` contenant les structures de données `Sue, PQ, A, students, employers`. Pour la réalisation de ces TADs, vous pouvez utilizer les classes Java standards, ou les implémentations du livre de Goodrich ou celles données dans les notes de cours ou encore vos propres réalisations. Mais les méthodes `pop` et `push` doivent avoir une complexité de O(1) et la file à priority doit avoir une méthode `insert` et `removeMin` chacune ayant une complexité de O(log n).
- Votre classe doit avoir une méthode `initialize(filename)` qui effectue la lecture du fichier d'entrée et qui effectue toutes les étapes d'initialisation.
- Votre classe doit avoir une méthode `execute()` qui execute l'algorithme de Gale-Shapley tel que décrit.
- Votre classe doit avoir une méthode `save(filename)` qui sauvegarde la solution trouvée dans le format prescrit.
- La méthode `main` de votre classe demande simplement à l'utilisateur le nom du fichier d'entrée et appelle ensuite les méthodes `initialize`, `execute` et `save`.
- Tous vos fichier Java doivent avoir un entête qui inclue votre nom et numéro d'étudiant.
- Votre code Java doit être commenté. Inclure tous vos fichiers dans un fichier zip appelé projectCSI2510_XXX.zip ou XXX et votre numéro d'étudiant. Inclure tous les fichiers requis afin de compiler votre programme.
- Inclure aussi dans le fichier zip les fichiers solutions à tous les fichiers d'entrée qui vous sont données

Grille de correction

Exactitude de la solution:	15%
Qualité de la programmation:	15%
Les méthodes `Initialize` and `save`:	10%
La méthode `execute`:	20%
TADs utilisés:	10%
Questions supplémentaires (Partie II):	30% (à être publiées plus tard)

Annexe: déroulement de l'algorithme

Voici le déroulement étape par étape du cas simple donné en page 1 et 2.

```
Sue= [ 0  1  2
Students= {-1, -1, -1}
Employers= {-1, -1, -1}
A= {1,  2,  3}
   {1,  2,  3}
   {2,  3,  1}
PQ[0]= {(1,0),(2,1),(3,2)}
PQ[1]= {(3,0),(2,1),(1,2)}
PQ[2]= {(1,0),(3,1),(2,2)}
```

Step 1:

```
2 <- Sue.pop()
Sue= [ 0  1
0 <- PQ[2].removeMin()
PQ[2]= {(3,1),(2,2)}
-1 <- e'=Students[0]
Students[0]= 2
Employers[2]= 0
Match: Cisco-Olivia
```

Step 2:

```
1 <- Sue.pop()
Sue= [ 0
2 <- PQ[1].removeMin()
PQ[1]= {(3,0),(2,1)}
-1 <- e'=Students[2]
Students[2]= 1
Employers[1]= 2
Match: CanadaPost-Sophia
```

Step 3:

```
0 <- Sue.pop()
```

```
Sue= [
0 <- PQ[0].removeMin()
PQ[0]= {(2,1),(3,2)}
2 <- e'=Students[0]
(1<-A[0][0]) < (3<-A[0][2])
Students[0]= 0
Employers[0]= 0
Employers[2]= -1
Sue= [ 2
Match: Thales-Olivia
```

Step 4:

```
2 <- Sue.pop()
Sue= [
2 <- PQ[2].removeMin()
PQ[2]= {(3,1)}
1 <- e'=Students[2]
(1<-A[2][2]) < (3<-A[2][1])
Students[2]= 2
Employers[2]= 2
Employers[1]= -1
Sue= [ 1
Match: Cisco-Sophia
```

Step 5:

```
1 <- Sue.pop()
Sue= [
1 <- PQ[1].removeMin()
PQ[1]= {(3,0)}
-1 <- e'=Students[1]
Students[1]= 1
Employers[1]= 1
Match: CanadaPost-Jackson
```

Sue is empty!