AeA

TD/TP 1 -- Jeu de la lettre qui saute

But

Le but de ce TD/TP est de se familiariser avec la notion de graphe et d'implémenter des algorithmes de parcours classiques dans le cadre d'un jeu ludique (qui peut par ailleurs et avec un peu d'imagination servir dans le cadre d'autres applications tel que un correcteur d'orthographe).

Description du jeu

Le but du jeu est de passer d'un mot à un autre par une suite de mots tels qu'un mot et le suivant ne diffèrent que d'une lettre. Par exemple, on peut passer de lion à peur par la suite de mots :

lion --> pion --> paon --> pain --> paix --> poix --> poux --> pour --> peur

Nous allons modéliser ce jeu par un graphe où deux mots sont reliés s'il ne différent que d'une lettre. Une suite de mots telle que ci-dessus est donc un chemin dans le graphe. L'ensemble des mots auxquels peut mener un mot constitue une composante connexe.

On appellera successeurs d'un mot les mots qui ne différent que d'une lettre de ce mot. Par exemple, les successeurs de lion sont pion et lien, ceux de lien sont rien, bien, sien, lieu, lier, lied, lion, tien et mien.

Exercice 1 : Modélisation et initialisation du jeu

Graphe par liste d'adjacence

Nous allons définir un graphe par un tableau `mot` qui contient l'ensemble des mots de départ et un tableau `listeSucc` de listes d'entiers qui contiendra les listes de successeurs. Un mot est repéré par son indice `i` dans le tableau `mot`. `listeSucc[i]` contiendra la liste des indices des mots reliés à `mot[i]`. Pour faciliter l'écriture du programme, il est utile de stocker le nombre de mots dans une variable `nb`.

Écrire une classe `Liste` pour manier des listes d'entiers, puis une classe `Graphe` avec un constructeur `Graphe(String[] lesMots)` qui initialise les variables ci-dessus avec des listes de successeurs vides.

Création des listes de successeurs

- Écrire une méthode `static void ajouterArete (Graphe g, int s, int d)` qui rajoute `s` à la liste des successeurs de `d` et `d` à celle de `s`, les mots d'indices `s` et `d` étant supposés différer d'une lettre.

- Écrire une méthode `static void lettreQuiSaute (Graphe g)` qui initialise les listes de successeurs selon la règle du jeu de la lettre qui saute. On pourra utiliser la fonction suivante qui renvoie vraie quand deux chaînes de même longueur diffèrent d'une lettre :

```
static boolean diffUneLettre (String a,
String b) {
    // a et b supposees de meme
longueur
    int i=0 ;
    while (i<a.length () && a.charAt
(i) == b.charAt (i))
      ++i ;
    if (i == a.length ()) return false
;
    ++i ;
    while (i<a.length () && a.charAt
(i) == b.charAt (i))
      ++i ;
    if (i == a.length ()) return true ;
    return false ;
  }
```

- Vous pouvez par exemple tester vos méthodes avec la fonction main suivante :

```
public static void main (String[] args)
{
    String[] dico3court = {
       "gag", "gai", "gaz", "gel",
"gks", "gin",
       "gnu", "glu", "gui", "guy",
"gre", "gue",
       "ace", "acm", "agi", "ait",
"aie", "ail",
       "air", "and", "alu", "ami",
"arc", "are",
       "art", "apr", "avr", "sur",
"mat", "mur" } ;
    Graphe g = new Graphe (dico3court)
;
    lettreQuiSaute (g) ;
    //afficher (g) ;
  }
```

Un Exemple de jeu de donnée

- La classe `Dicos.java` contient les tableaux des mots de 3, 4 et 5 lettres (`Dicos.dico3`, `Dicos.dico4`, `Dicos.dico5`). Il peut y avoir des doublons, mais on n'essaiera pas de les traiter (plusieurs sommets du graphe risquent donc de correspondre au même mot s'il apparaît plusieurs fois dans le dictionnaire).

Exercice 2 : Calcul des composantes connexes

Notre but est maintenant de calculer les composantes connexes grâce à un parcours en profondeur.

Parcours en profondeur : préparation

Pour faire un parcours en profondeur, nous avons besoin de stocker pour chaque mot d'indice `i` s'il a déjà été rencontré. On utilisera pour cela un tableau de boolean de nom `dejaVu`.

- Rajouter les variables nécessaires dans la classe `Graphe`, et modifier le constructeur en conséquence.

Parcours en profondeur : en partant d'un noeud

- Écrire une fonction `static void DFS(Graphe g, int x)` qui parcourt en profondeur le graphe `g` à partir du mot d'indice `x`. Pour mémoire, cela veut dire qu'on marque `x` comme déjà vu et que l'on inspecte un à un les mots de sa liste de successeurs. Si un successeur d'indice `y` n'est pas marqué comme déjà vu, on parcourt récursivement ce mot, on dit alors qu'il a été vu par le mot d'indice `x`. Pendant toute la durée de l'appel à `DFS(g,x)`, on dit que l'on est en train de visiter le mot d'indice `x`.

- Modifier la fonction de sorte que tous les mots vus durant la visite du mot d'indice `x` soit affichés.

Parcours en profondeur : visiter tous les noeuds

- Écrire une fonction `static void visit (Graphe g)` qui initialise `dejaVu`, puis qui parcourt en profondeur tout le graphe. Pour mémoire, cela veut dire qu'on prend un mot non encore vu, qu'on parcourt en profondeur à partir de ce mot et qu'on recommence jusqu'à ce que tous les mots aient été vus.

- Faire en sorte que les composantes connexes soient affichées. Avec les mots de dico3court, on doit trouver 9 composantes :

```
1:   gag gaz gai gui gue gre are art
arc ait air avr apr ail aie ace acm guy
2:   gel
3:   gks
4:   gin
5:   gnu glu alu
6:   agi ami
7:   and
8:   sur mur
9:   mat
```

- Trouver la composante connexe de lion et peur parmi les mots de 4 lettres qui sont listés dans le tableau `Dicos.dico4` de `Dicos.java`.

Exercice 3 : Calcul de chemins

Le but est maintenant de calculer un chemin d'un mot à un autre s'il en existe un.

Arborescence

Lors d'un parcours en profondeur, si lors de la visite d'un mot d'indice `x`, un successeur d'indice `y` est vu par `x`, on peut définir `x` comme le père de `y`. On obtient alors une arborescence. Rajouter dans la classe `Graphe` une variable `pere` qui permettra de stocker pour le mot d'indice `i`, l'indice `pere[i]` de son père dans l'arborescence.

- Modifier le constructeur et les deux fonctions de parcours en profondeur pour intégrer le calcul de l'arborescence.

Retrouver le chemin du parcours

- Modifier le programme pour qu'il prenne deux mots en arguments et affiche une suite de mots menant de l'un à l'autre si c'est possible. Écrire pour cela une fonction `static void chemin (Graphe g, String from, String to)` qui affiche un chemin de `from` à `to` s'il en existe un. On pourra utiliser la fonction suivante qui retrouve l'indice d'un mot `m` dans un tableau de mots `tabMots` :

```
static int indice (String m, String[]
tabMots) {
    for (int i=0 ; i<tabMots.length ;
++i)
        if (m.equals (tabMots[i]))
return i ;
    throw new Error (m + " n'est pas
dans le tableau.") ;
  }
```

- Trouver de cette manière un chemin de lion à peur.

Exercice 4 : Calcul de plus courts chemins

Parcours en largeur (BFS)

- Écrire une fonction `static void BFSIteratif (Graphe g, int x)` qui fait un parcours en largeur à partir du mot d'indice `t`. Dans un parcours en largeur, on visite d'abord tous les successeurs et ensuite les successeurs des successeurs non encore visités et ainsi de suite. Les premiers successeurs rencontrés seront visités en premier, il est donc utile d'utiliser une file "first in first out".

- Trouver de cette manière un chemin de lion à peur.

Plus court chemin

- Modifier la fonction `chemin` de l'exercice 3 pour qu'elle trouve un plus court

chemin plutôt qu'un chemin quelconque.

Exercice 5 : Généralisation

Graphe orienté généralisé

- On vous donne maintenanint les deux dictionnaire suivant : dico.court.txt et dico.long.txt. Chaque fichier contient un mot par ligne. Le graphe que l'on considère maintenant est orienté et défini par deux paramètres `sup` et `dif`. Il y a un arc d'un mot u vers un mot v quand v peut être obtenu à partir de u en supprimant au plus `sup` lettres et en changeant au plus `dif`.
- Modifier votre classe graphe pour calculer le graphe des mots (on testera par exemple avec `sup=1` et `dif`=1).

Excentricité

- L'excentricité d'un mot u est la longueur maximale d'un plus court chemin de u vers un autre mot.
- Trouver un mot d'excentricité maximale et découvrir ainsi un plus court chemin le plus long possible du graphe.