DALENCOURT Alex
GOLDAK Ludovic

# Compte rendu TP1 Lettre qui saute

Pour ce TP tous les exercices ont étés traité à l'exception de la question qui traite de l'excentricité de l'exercice 5 (à la fin du document nous expliquons nos pistes de conception pour cette partie). L'implémentation des programmes de chaque exercices est réalisé dans la classe Exercices du package Utils

Détail des exercices :

## Exercice 1

La réalisation de la lettre qui saute est effectué à l'aide d'une double boucle qui permet de comparer tous les mots les uns avec les autres. Le calcul de la différence d'une lettre est effectué dans la classe StringUtils. Le calcul se déroule en deux partie :

- Parcours des chaînes avec un même curseur jusque la première différence ou la fin.

- Parcours des chaînes avec un même curseur jusque la deuxième différence ou la fin.

```java
public class StringUtils {

    public 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;
        }
        return i == a.length();
    }
```

Il est possible d'exécuter le programme via le jar exécutable Exercice1.jar :

**java -jar {chemin vers Exercice1.jar}**

DALENCOURT Alex
GOLDAK Ludovic

## Exercice 2

Le calcul de DFS est réalisé selon le principe de récursivité. A partir du mot x, si celui ci n'est pas déjà vu on parcours tous les successeurs en rappelant la méthode sur le successeur. Au moment du DFS le mot en cours de traitement est marqué comme parcouru.

```java
/**
 * Affiche la composante connexe contenant le mot x.
 *
 * @param g Le graph
 * @param x Le mot à visiter
 */
public static void DFS(Graphe g, int x) {
    if (!g.dejaVu(x)) {
        System.out.print(g.parcour(x) + " ");
        Successeur s = g.getSuccesseur(x);
        while (s != null) {
            DFS(g, s.getNoeud());
            s = s.getNextNoeuds();
        }
    }
}
```

Il est possible d'exécuter le programme via le jar exécutable Exercice2.jar :

**java -jar {chemin ver Exercice2.jar}**

## Exercice 3

La réalisation du parcours en profondeur pour trouver un chemin entre deux mots fonctionne comme pour DFS selon le système de récurrence. On dépile la liste des successeurs jusque arrivé au cas où le mot courant est égal au mot demandé. Enfin, si tous les mots ont été parcouru sans trouver le mot choisi c'est qu'il n'y a pas de chemin possible.

DALENCOURT Alex
GOLDAK Ludovic

```java
/**
 * Récupère la liste des mots entre deux mots avec le parcours en profondeur par récurence
 * @param g graphe
 * @param path chemin actuellement réalisé
 * @param actuel mot courrant
 * @param to mot final
 * @return chemin réalisé
 */
public static ArrayList<Integer> getChemin(Graphe g, ArrayList<Integer> path, int actuel, int to) {
    if (actuel == to) {
        path.add(actuel);
        return path;
    }
    g.parcour(actuel);
    Successeur next = g.getSuccesseur(actuel);
    while (next != null) {
        if (!g.dejaVu(next.getNoeud()) && getChemin(g, path, next.getNoeud(), to) != null) {
            path.add(actuel);
            return path;
        }
        next = next.getNextNoeuds();
    }
    return null;
}
```

Il est possible d'exécuter le programme via le jar Exercice3.jar :

**java -jar {chemin vers Exercice3.jar} {mot de démarrage} {mot recherché de destination}**

Les mots se basent sur le dico 4 lettres.

## Exercice 4

Le parcours en largeur est réalisé par une double boucle tant que qui se termine soit lorsque la pile des mots à exécuté est non vide où que le mot ciblé est atteint.

DALENCOURT Alex
GOLDAK Ludovic

```java
/**
 * Implémentation de la recherche de chemin via le parcours en largeur
 * @param g graphe
 * @param from mot de départ
 * @param to mot d'arrivé
 * @return la liste des étapes
 */
public static List<Integer> BFSIteratif(Graphe g, int from, int to) {
    g.resetParcour();
    List<Integer> aList = new ArrayList<>();
    aList.add(from);
    g.parcour(from);
    while (aList.size() > 0) {
        int s = aList.remove( i: 0);
        Successeur next = g.getSuccesseur(s);
        while (next != null) {
            if (!g.dejaVu(next.getNoeud())) {
                aList.add(next.getNoeud());
                g.setPere(next.getNoeud(), s);
                g.parcour(next.getNoeud());
                if (next.getNoeud() == to) return aList;
            }
            next = next.getNextNoeuds();
        }
    }
    return null;
}
```

Il est possible d'exécuter le programme via l'archive Exercice4.jar :

**java -jar {chemin vers Exercice4.jar} {mot de démarrage} {mot recherché de destination}**

Les mots se basent sur le dico 4 lettres.

## Exercice 5

Pour réaliser la succession entre deux mot en fonction du nombre de suppressions possibles et le nombre de différences possibles on utilise la méthode de la classe StringUtils : recSupDiff.

La recherche des successions possible est effectuée par récurrence qui test tous les chemins possibles. La récurrence s'arrête si une contrainte n'est plus respecté ou si la chaîne d'origine à entièrement été analysée sans problèmes. Chaque cycle de récurrence créer 2 ou 3 (un match entre deux caractères est possible) récurrences : cas de différence, cas de suppression et si possible cas de match..

DALENCOURT Alex
GOLDAK Ludovic

```
// Test de finission des parcours
if(sup_actuel > sup || diff_actuel > diff){
    // si trop de suppressions ou trops de différence chemin refusé
    return false;
} else if(objectif.length() <= objectif_idx){
    // si l'objectif a été atteint le chemin est accepté
    return true;
} else if(origine.length() <= origine_idx){
    // sinon si a est arrivé à la fin on regarde si il est possible de f
    return sup_actuel + (objectif.length() - objectif_idx) <= sup;
}

// si les parcours ne sont pas fini
if(origine.charAt(origine_idx) == objectif.charAt(objectif_idx)){
    // si un match et réalisé on déplace les deux curseurs en a et en b
    // il possible que le match soit un faut match il faut donc recherch
    match = recSupDiff(origine,objectif,sup,diff, origine_idx: origine_idx
}
// sinon on test les différents chemins possibles :
// - différence entre deux caractères
// - suppression sur le mot a
swap = recSupDiff(origine,objectif,sup,diff, origine_idx: origine_idx + 1,
sup_a = recSupDiff(origine,objectif,sup,diff, origine_idx: origine_idx + 1,

return  swap || sup_a || match;
```

Il est possible d'exécuter la recherche du plus court chemin avec un dictionnaire chargé selon la méthode sup / diff via l'archive Exercice5.jar :

**java -jar {chemin vers Exercice5.jar} {chemin vers le dictionnaire à charger} {nombre de suppressions possibles} {nombre de différences possibles} {mot d'origine} {mot objectif}**

Nous avons pas traité la question excentricité faute de temps mais nous avons réfléchi à la manière de l'implémenter. Nous avons relever deux conceptions possible afin de trouver le plus long cas de plus court chemin pour un mot d'origine :

- La première conception consiste à rechercher pour tous les mots le plus court chemin entre l'origine et ces mots. Enfin, à chaque itération, si le plus court chemin est plus grand que dernier plus court chemin trouvé ce chemin est sauvegardé à la place du dernier.

- La seconde conception consiste à rechercher tous les mots pouvant être accéder par le mot d'origine (en dépilant tous les successeurs) puis en recherchant le plus court chemin pour chaque mot trouvé. On sauvegardera à chaque itération le chemin le plus long étant un plus court chemin.

Cette deuxième implémentation sera préférable à la première car elle permettra de ne pas réaliser des recherches de plus court chemins pour les cas où ils n'existent pas. La première implémentation

sera préférable lorsque nous aurons une configuration dans les pires cas. Si on prend le cas où tous les mots peuvent être atteint à partir du mot d'origine tous les chemins devrons être cherché (dépend n) toutefois dans le deuxième cas en plus chercher tous les chemins il faudra rechercher tous les mots atteignable (qui se trouve être la totalité des mots du dictionnaire).