Rapport de TP d'Algorithmique

Loris Croce

23 mars 2018

Table des matières

Résumé

Pour ce TP le langage **Java** a été utilisé. Pour la compilation et l'éxecution, le logiciel **Ant** a été utilisé en utilisant des fichier de *build*. Pour compiler le projet (Ant doit être installé au préalable) il faut se placer dans le répertoire du TP correspondant et taper la commande : `ant`, pour l'exécuter il faudra entrer : `ant run`.

Chapitre 1

Gestion de partitions

1.1 TDA Gestion de partition

Le TDA *Gestion de partition* est un TDA qui qui définit une partition p composée d'éléments e [1], lesquels sont regroupés en son sein par classes. Il dispose de deux opérations :

— `p.classe(e)` : retourne la classe de l'élément e.

— `p.fusion(c1, c2)` : fusionne les classes c_1 et c_2 de la partition p.

Pour implémenter ce TDA une *interface* `GestionPartition` qui définit la structure du TDA a été créée. Deux solutions ont été implémentées : Le *Tableau de partition* et le *Tableau de pères*.

1.1.1 Tableau de partition

Comme les éléments gérés ici sont des entiers, il est possible de les représenter sous la forme d'indices d'un tableau où les cases contiendront la classe dans laquelle chaque élément est stocké. La classe associée à cette implémentation est `TableauPartition`.

Les complexités associées à cette implémentation sont :

— `p.classe(e)` : $\mathcal{O}(1)$, ici implémentée par `getClasse(int e)`.

— `p.fusion(c1, c2)` : $\mathcal{O}(n)$, ici implémentée par `fusion(int c1, int c2)`.

1.1.2 Tableau de pères

Cette implémentation est associée à celle par *forêts* en la traduisant sous forme de tableau où comme pour le tableau de père les indices représentent les éléments mais les valeurs des cases renseignent le *père* de l'élément [2]. La classe associée à cette implémentation est `TableauPerePartition`.

1. Ici, par souci de simplicité les éléments seront des entiers
2. si cet élément est racine on considèrera qu'il est son propre père.

Les complexités associées à cette implémentation sont :

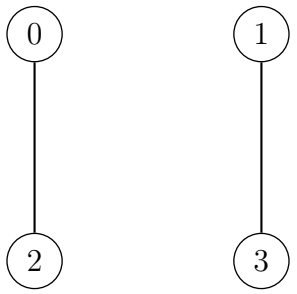
— `p.classe(e)` : $\mathcal{O}(h)$, ici implémentée par `getClasse(int e)`.

— `p.fusion(c1, c2)` : $\mathcal{O}(1)$, ici implémentée par `fusion(int c1, int c2)`.

1.2 Composantes connexes

Pour résoudre le problème des composantes connexes il a été nécessaire de créer une classe `Graphe` contenant les arêtes sous la forme d'une matrice d'entiers et les sommets sous la forme d'un tableau d'entiers. Il a également fallu surcharger le constructeur de `TableauPartition` pour qu'il prenne en paramètre un type `Graphe` et effectue les fusions nécessaires pour que les sommets reliés appartiennent à la même classe, montrant ainsi les composantes connexes du graphe.

Exemple Un graphe G de la forme :



Donnera un tableau de la forme :

Valeurs	0	1	2	3
Classes	0	1	0	1

1.3 Arbres couvrants minimaux

Pour le problème des arbres recouvrant de poids minimaux on définit une classe `Arbre`, une méthode `getOrdoredAretes` dans `Graphe`. Dans `TableauPartition` on a une méthode `GrapheToArbre` qui, comme son nom l'indique, prend en paramètre un graphe et retourne l'arbre de poids minimum en passant par le tableau de partition.

Chapitre 2

Programmation dynamique *Distance d'édition*

Ici, on va chercher à implémenter l'algorithme de **distance d'édition** qui consiste à rechercher le nombre d'opérations –*insertion*, *suppression* ou *substitution* de caractères– nécessaires pour passer d'un mot donné à un autre. On définit donc une classe `Comparaison` qui contiendra toutes les méthodes associées.

2.1 Méthode récursive

Pour cette méthode on créée donc une méthode `Recursif` avec pour paramètre deux chaines de caractères A et B. On définit deux cas d'arrêts :

— A est nul.

— B est nul.

Sinon on rappelle l'algorithme sur des sous parties de A et B en incrémentant la distance si les deux caractères sélectionnés sont différents.

2.2 Avec mémoïzation

La méthode récursive avec *mémoïzation* reprend le même principe en ajoutant un tableau de distances entre sous-chaînes où les valeurs sont ici initialisées à 999[1] et un *niveau* qui s'incrémentera si l'on repère une différence de caractère dans l'appel récursif. Elle est ici implémentée par `RecursifMemo`.

1. Pour simuler un $+\infty$.

2.3 Méthode dynamique

Pour l'algorithme de programmation dynamique les chaines de caractères placées en paramètres sont remplacées par des *tableaux de caractères* (comme en C), pour effectuer la transformation à l'appel de la fonction on appelle la fonction `toCharArray()` du type `String` de Java. La fonction `Dynamique` utilise une matrice à deux dimensions qui stocke les résultats intermédiaires pour calculer les suivants et la retourne.

Exemple avec les mots *tourte* et *tartre* :

	ϵ	**t**	**a**	**r**	**t**	**r**	**e**
ϵ	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
o	2	1	1	2	3	4	5
u	3	2	2	2	3	4	5
r	4	3	3	2	3	3	4
t	5	4	4	3	2	3	4
e	6	5	5	4	3	3	3

2.4 Affichage des opérations

Pour l'affichage (en mode console) on dispose de plusieurs méthodes. Tout d'abord `tableauToString` prend en paramètre une matrice 2×2 d'entiers fournies par `Dynamique` et `RecursifMemo` et l'affiche. Les fonctions `chemin` et `cheminRecMemo` prennent en paramètre les matrices retournées par les fonctions respectives et des chaines sous forme de tableau d'entier pour afficher à l'écran les opérations effectuées pour passer d'un mot à un autre.

2.5 Comparaison des résultats

Après études des temps d'exécution on peut observer que en raison de sa complexité exponentielle l'algorithme *récursif* "naïf" présente de loin le temps le plus élevé car on effectue énormément de calculs inutiles. Le récursif avec *mémoïzation* augmente significativement la rapidité mais l'algorithme de *programmation dynamique* reste de loin le plus optimisé. Cependant pour l'affichage des opérations `cheminRecMemo` semble plus performant que que `chemin`.