Travail Pratique

**Département de génie logiciel et des TI**

|  |  |
| --- | --- |
| Numéro du laboratoire | 3 |
| Étudiant(s) | Benoit, Marie-Ève  Desmarais, Gabriel |
| Code(s) Permanent(s) | BENM22568707  DESG24078908 |
| Cours | GTI310 |
| Session | Automne 2010 |
| Groupe | 01 |
| Professeur | Stéphane Coulombe |
| Chargé de laboratoire | Jean-François Franche |
| Date | 02 Novembre 2010 |

**Table des matières**

[1. Introduction 1](#_Toc275597477)

[2. Choix de design 2](#_Toc275597478)

[3. Implémentation 3](#_Toc275597479)

[4. Analyse du projet 4](#_Toc275597480)

[5. Conclusion 5](#_Toc275597481)

[6. Réponse aux questions supplémentaires 6](#_Toc275597482)

[7. Manuel de l’usager 7](#_Toc275597483)

[ANNEXE 1](#_Toc275597484)

# Introduction

Le musée Labyrinthe-En-Ville désire pourvoir identifier tous les parcours possibles de ses salles.

La présente itération a pour but de compléter une application déjà existante (Unreal Networks Solver) afin d’y ajouter les classes permettant la lecture du fichier d’information (informations du graphe), le calcul des parcours et, finalement, d’écrire la solution dans un fichier texte.

Pour ce faire, une classe Data pour conserver les informations des nœuds et des arches, une classe concreteParser pour prendre les informations provenant d’un fichier texte, une classe ConcreteSolver qui résout la problématique, une classe PathList pour conserver les différentes solutions et une classe ConcreteWriter pour l’écriture du fichier de solutions devront être implémentées.

Le présent document contient une section Choix de Design qui expose les différentes décisions possibles inhérentes à la conception, une section Implémentation contenant les détails de l’implémentation et l’analyse complète du projet est exposée dans la section Analyse du Projet. Un retour sommaire sur les notions abordées se trouve dans la section Conclusion, il s’en suit le guide d’utilisateur ainsi que l’annexe.

# Choix de design

Pour réaliser le projet, il était nécessaire de respecter certains éléments de design. Ces éléments consistaient à devoir parcourir tous les nœuds (salles) du graphe, de respecter le fait que certains arches (couloirs) étaient à sens unique, de pouvoir commencer le parcours à partir de n’importe quel nœud et de pouvoir réutiliser l’application pour d’autres problématiques (Ne pas la limiter au musée). L’application devait également générer un fichier texte contenant tous les parcours solutionnant la problématique.

Plusieurs possibilités ont été envisagées afin de répondre aux demandes.

1. Faire une liste de listes pour les chemins possibles ou faire des tableaux statiques.

Une des principales problématiques rencontrées durant le développement de l’application était le choix d’utiliser une liste de liste pour conserver les différents parcours solutionnant les problématiques ou bien des tableaux statiques.

La première solution permet plus de flexibilité car elle ne nécessite pas de savoir combien d’éléments la solution va contenir ni le nombre de solution qu’il y aura. La deuxième nécessite de limiter le nombre maximal de solution et d’éléments par solution.

1. Utiliser la récursivité ou faire plusieurs boucles imbriquées pour solutionner le graphe.

Un autre dilemme s’est présenté lorsqu’est venu la nécessité de résoudre la problématique, soit trouver tous les parcours possibles passant par tous les nœuds.

La première solution considérée fut de faire plusieurs boucles imbriquées pour parcourir le tableau plusieurs fois et, ainsi, s’assurer d’avoir vérifié toutes les possibilités. Cela entraînerait un code lourd et difficile à déchiffrer qui nécessiterait de parcourir à nouveau toutes les valeurs jusqu’à ce qu’aucun changement n’ai été fait. La deuxième solution consiste à utiliser une fonction récursive qui fait appel à elle-même quand il est nécessaire de vérifier si un nœud à déjà été parcouru. Cette solution est plus lisible en terme de code, plus complexe aussi car elle nécessite des notions de récursivité pour la comprendre et évite de parcourir plusieurs fois les données déjà traitées.

# 3. Implémentation

Pour réaliser l’application solutionnant la problématique, plusieurs classes ont été ajoutées/implémentées à l’application Unreal Networks Solver.

Premièrement, une classe Data a été ajoutée afin de conserver les informations obtenues par la lecture du fichier texte d’information. Augmentant ainsi la cohésion en évitant de faire plusieurs lectures du fichier d’informations durant le solutionnement de la problématique. Cette classe contient 4 méthodes, getNbSommet(), getValInf(), getDepart() et getLinks().

Ensuite la classe ConcreteParser héritant de Parser a été implémentée. Puisque qu’obtenir le nombre de lignes contenues par un fichier sans en faire la lecture préalablement n’est pas possible, il a été décidé de faire une première boucle de lecture du fichier pour obtenir cette information. Ce faisant, cela permet de vérifier si la dernière ligne du fichier comprenant le symbole $ signifiant la fin du fichier et d’ainsi vérifier la validité du fichier, évitant ainsi de le parcourir entièrement pour rien. De plus, cela permet de créer un tableau ayant une taille correspondant au nombre exacte d’éléments du fichier et de faciliter l’obtention et l’enregistrement des informations du fichier.

Par après, une classe PathList a été créée pour conserver une liste de liste de solution. Cette façon de faire étant plus flexible et permettant l’ajout dynamique de solution à une solution évite de connaître préalablement le nombre de solution que contiendra le fichier final de solution et d’avoir une lite de chemin. Ainsi, si l’on veut les parcourir plus tard ou ajouter des manipulations sur ces listes de chemins, leur accès en est facilité et ne nécessite pas de changement majeur de la classe PathList. Cette classe contient 3 méthodes, addPath(List<Integer> path), getPath(int index) et pathCount().

Vien ensuite l’implémentation de la classe ConcreteSolver dont le but est de solutionner la problématique à partir des informations emmagasinées dans la classe Data. On commence par vérifier si le nœud est fait et si c’est celui de départ, pour ensuite appeler la fonction récursive recursion(input, input.getDepart(), new LinkedList<Intefer>), qui s’appelera autant de fois que nécessaire, c'est-à-dire tant que tous les nœuds n’ont pas été parcourus et qu’on n’est pas revenu au nœud de départ. Quand un chemin est trouvé, il est ajouté la liste de solution (list.addPath(path)).

Finalement, la classe concreteWriter a été complétée et écrits les solutions obtenus en parcourant la liste de solution (et chaque éléments de la solution lue) dans un fichier texte de solution.

# 4. Analyse du projet

*On ne calcul pas les coûts car ce n’est pas nécessaire pour résoudre la problématique ni pour la réutilisation de l’application. Dans le parser on récolte les coûts et on les gardes mémoires (dans links) dans la classe data, mais suivant l’énoncé et les contraintes du laboratoire on a pas besoin de les utiliser pour le coût total de la solution, mais dans l’Éventualité ou les r`gles/pobjectifs changerait, elle serait déjà acquise. Seul l’algo de solution serait à modifier.*

Nous avons utilisé deux projets complètement séparés pour les deux applications à développer, mais la création d’un seul projet avec des sources partagées aurait permis de mieux réutiliser le code déjà écrit lors de la première application et d’éviter le doublon de code pour certaines fonctions. Cette réutilisation de code nous aurait donné plus de difficulté pour écrire un code s’appliquant à toutes les situations encourues dans les deux programmes à la place de palier seulement aux situations survenant dans un programme spécifique, mais la réutilisation de ces classes partagées aurait alors pu être possible pour un autre projet gérant des fichiers audio tels que ceux-ci.

Nous n’avons pas utilisé de classe gérant le type de fichier audio utilisé et cette méthode reste la meilleure à notre avis pour atteindre les objectifs fixés par l’énoncé du projet. Dans le cas où plusieurs de types de fichiers audio doivent être supportés, le transfert du code servant à décoder des fichiers WAV devra être envoyé dans une classe tierce, nous permettant ainsi d’accomplir ces nouveaux objectifs.

Il n’y a aucune fonction lisant l’en-tête du fichier audio, cette fonction, si elle doit exister, devrait être placée dans la classe gérant le type du fichier audio utilisé. Nous n’avons pas trouvé nécessaire de créer cette fonction tant que nous utilisions le même type de fichier audio ou que nous n’avions pas de classe spécifique à ce type.

La fonction validant les en-têtes a par contre été faite dans la deuxième application, puisqu’elle doit valider l’en-tête de plusieurs fichiers audio. Cette fonction aurait facilement pu être retrouvée dans la première application, mais les objectifs de celle-ci ne nécessitaient pas de la faire. Dans le cas où notre application prendrait de l’ampleur de quelque manière que ce soit, soit en incorporant la deuxième application ou en ajoutant plus de types de fichiers audio possibles pour la lecture, la création de cette méthode serait un excellent ajout.

# 5. Conclusion

Lors de l’écriture de la première application, nous avons eu plusieurs problèmes de compréhension liés au transfert des valeurs en 16 bits vers des valeurs en 8 bits.

En plus de ces quelques problèmes, la gestion de la lecture et la manipulation des fichiers par morceaux sans perdre d’informations c’est avéré un intéressant défi à relever. En effet, ces morceaux servaient à réduire la mémoire utilisée par les applications, mais sectionnaient la lecture et la manipulation des fichiers.

Malgré ces divers problèmes, les objectifs énoncés à travers le projet ont été accomplis en entier. Les différentes complications et problématiques rencontrées ont donc pu être réglées, mais auront nécessité plusieurs explications et recherches.

# 7. Manuel de l’usager

Lors des tests effectués, nous avons utilisé les configurations fournies par Eclipse pour gérer les arguments envoyés aux applications. Ces arguments étaient les suivants :

"Session 3\\GTI310\\AudioResampler-TP2-P1\\App1Test1Stereo16bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P1\\output.wav"

pour l’application 1 et les suivants pour l’application 2 :

" Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Original\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod1\_AmpBase\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod2\_30SamplesDelay\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod3\_Echo1SecMono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod4\_WhiteNoise1\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod5\_WhiteNoise2\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod6\_WhiteNoise3\_Mono8bits.wav" "Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod7\_WhiteNoise4\_Mono8bits.wav" " Session 3\\GTI310\\AudioResampler-TP2-P2\\App2\_Mod8\_PinkNoise\_Mono8bits.wav"

La méthode qui doit être utilisée en dehors de l’interface de développement est la suivante :

java <programme1> <ficher d’entrée> <fichier de sortie>

java <programme2> <ficher de référence> <fichiers à analyser>

Cette méthode permettra à un utilisateur de lancer les deux applications avec les arguments nécessaires.

# ANNEXE

**Classe ConcreteSolver**

**package** gti310.tp3.solver;

**import** java.util.LinkedList;

**import** java.util.List;

**import** gti310.tp3.Data;

**import** gti310.tp3.PathList;

**public** **class** ConcreteSolver **implements** Solver<Data,PathList> {

PathList list = **new** PathList();

**private** **boolean**[] nodeDone = **null**;

**private** **int**[][] links;

**int** start = -1;

/\*\*

\* c1+c2+c3+c4(n+1)+c5(n)+N^N+c7

\* (c1+c2+c3+c4+c7)+(c4+c5)n+n^n

\* k1+k2n+n^n

\* n^n

\*

\* L'appel de la fonction recursion à la ligne 6 vaut N^N, ce qui donne, à la fonction solve, la valeur de :

\* O(N^N)

\*

\* **@param** input

\*/

@Override

**public** PathList solve(Data input)

{

// **TODO** Auto-generated method stub

links = input.getLinks();

start = input.getDepart();

nodeDone = **new** **boolean**[input.getNbSommet()];

**for** (**int** i=0;i<nodeDone.length;i++)

nodeDone[i]=**false**;

recursion(input, input.getDepart(), **new** LinkedList<Integer>()); //O(N^N)

**return** list;

}

/\*\*

\* À cause de la récursion de la ligne 5, la valeur de la fonction est de :

\* O(N^N)

\*

\* **@param** data

\* **@param** node

\* **@param** path

\*/

**private** **void** recursion(Data data, **int** node, List<Integer> path)

{

nodeDone[node-1] = **true**;

path.add(node);

**for** (**int** i=0;i<links.length;i++)

{

**if** (links[i][0]==node && !nodeDone[links[i][1]-1])

recursion(data, links[i][1], path);

**if** (links[i][0]==node && links[i][1]==start)

{

**boolean** winner=**true**;

**for** (**int** j=0;j<nodeDone.length;j++)

**if** (!nodeDone[j])

winner=**false**;

**if** (winner)

{

path.add(start);

list.addPath(path);

path.remove(**new** Integer(start));

}

}

}

path.remove(**new** Integer(node));

nodeDone[node-1] = **false**;

}

}