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](#_Toc276807952)

[2. Choix de design 2](#_Toc276807953)

[3. Implémentation 3](#_Toc276807954)

[4. Analyse du projet 4](#_Toc276807955)

[5. Conclusion 5](#_Toc276807956)

[7. Manuel de l’usager 6](#_Toc276807957)

[ANNEXE 1](#_Toc276807958)

# Introduction

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

La présente itération a pour but de compléter un cadre de développement déjà existant (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 les solutions possibles dans un fichier texte.

Pour ce faire, une classe Data pour conserver les informations des nœuds et des arcs, 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 arcs (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 résolvant 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 résolvant 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 résoudre le graphe.

Un autre dilemme s’est présenté lorsqu’est venue 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 s’il y a un nouveau chemin possible à partir du nœud en cours. 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.

1. Utiliser une seule boucle de lecture du fichier texte ou en faire deux.

Il a été considéré de faire une seule boucle de lecture qui éviterait plusieurs lectures inutiles, mais nécessite de savoir à l’avance le nombre d’éléments contenus dans le fichier. L’autre solution était de faire deux boucles. La première permettrait d’évaluer le nombre de lignes du fichier, de vérifier partiellement s’il est valide et la deuxième de traiter l’information.

# 3. Implémentation

Pour réaliser l’application résolvant la problématique, plusieurs classes ont été ajoutées ou implémentées au cadre de développement 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 calcul de la solution à la problématique. Cette classe contient 4 méthodes, getNbSommet(), getValInf(), getDepart() et getLinks().

Ensuite la classe ConcreteParser, héritant de Parser, a été implémentée. Puisqu’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 exact 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 solutions. Cette façon de faire étant plus flexible et permettant l’ajout dynamique de solutions évite de devoir connaître préalablement le nombre de solution que contiendra le fichier final. 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().

Vient ensuite l’implémentation de la classe ConcreteSolver dont le but est de résoudre 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(Data data, int node, List<Integer> path) qui s’appellera 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 ou qu’il n’y a pas de chemin possible à partir de là où nous sommes rendus. Quand un chemin est trouvé, il est ajouté la liste de solutions (list.addPath(path)).

Finalement, la classe concreteWriter a été complétée et elle écrit les solutions obtenues en parcourant la liste de solutions ainsi que chaque élément de la solution lue, dans un fichier texte de solution.

# 4. Analyse du projet

Dans le fichier contenant l’information sur le graphe à parcourir, on retrouve les coûts de chaque arc. Cette information n’est pas utilisée dans la présente application puisqu’elle n’est pas nécessaire à la complétion de la tâche ou pour répondre aux contraintes, car les solutions obtenues sont indépendantes des coûts des arcs. Cependant, lors de la lecture du fichier d’information, le « parser » récolte les coûts de chaque arc et les gardes en mémoire dans la variable links de la classe Data. Ainsi, si les objectifs de l’application venaient à changer, l’information serait déjà conservée par l’application et seule la classe se chargeant de résoudre la problématique serait à modifier.

*.*

La lecture du fichier a amené un questionnement sur le nombre de lectures nécessaire pour le parcourir et gérer ses informations. Il a été décidé de faire une double lecture, la première pour obtenir le nombre exact de lignes dans le fichier texte et la deuxième pour emmagasiner seulement l’information pertinente.

L’application fait plusieurs vérifications afin de s’assurer de la validité du fichier d’information, commençant par vérifier que le signe $ se trouve à la dernière ligne, pour ensuite vérifier que les 2 premières lignes contiennent une seule valeur valide. S’il n’y a pas de valeur pour le nœud de départ, l’application utilise le nœud ayant la valeur la plus basse. En cas d’erreur, une méthode error() est appelée et affiche un message d’erreur dans la console pour ensuite terminer l’application.

Pour ce qui est de résoudre le graphe, un algorithme récursif a été utilisé afin de pouvoir trouver les solutions sans avoir à parcourir les nœuds plusieurs fois inutilement. Cette approche permet d’être plus dynamique face au problème, l’algorithme se terminant une fois que tous les nœuds ont été parcourus et que l’on est revenu au nœud de départ.

Une liste de liste de solutions est utilisée pour permettre une écriture, une lecture et une manipulation simple des solutions obtenues. En effet, chaque élément de la liste de solutions est en fait une liste d’éléments (nœuds) constituant un parcours valide.

# 5. Conclusion

Le principal défi rencontré durant cette itération fut l’implémentation de l’algorithme trouvant la solution. En effet, l’utilisation d’une méthode récursive est complexe, car elle est moins intuitive à comprendre et complique le calcul de la notation O. Cela a permis des acquis utiles et pertinents en récursivité.

De plus, la lecture et la conservation de l’information du graphe se sont avérées plus complexes que prévu, car il fallait trouver une façon optimale et flexible de conserver l’information sans limiter la quantité ou la qualité des résultats finaux. Ce fut un casse-tête intéressant et stimulant.

Finalement, malgré ces divers problèmes, les objectifs et contraintes du projet ont été respectés et accomplis en entier. Les différentes complications et problématiques rencontrées ont donc pu être réglées, mais auront nécessité quelques recherches et un bon travail de conception.

# 7. Manuel de l’usager

L’environnement utilisé lors des tests était Windows 7 avec Java version 6 update 22.

Lors des tests effectués, les configurations fournies par Eclipse ont été utilisées pour gérer les arguments envoyés à l’application. Ces arguments étaient les suivants :

Pour la situation du musée :

"Session 3\\GTI310\\UnrealNetworksSolver-TP3\\Musee.txt"

"Session 3\\GTI310\\ UnrealNetworksSolver-TP3\\Sol-Musee.txt"

Pour la situation de Vendeur :

"Session 3\\GTI310\\UnrealNetworksSolver-TP3\\Vendeur.txt"

"Session 3\\GTI310\\ UnrealNetworksSolver-TP3\\Sol-Vendeur.txt"

Pour la situation de Grosse-Neige :

"Session 3\\GTI310\\UnrealNetworksSolver-TP3\\Grosse-Neige.txt"

"Session 3\\GTI310\\ UnrealNetworksSolver-TP3\\Sol-Grosse-Neige.txt"

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

java <programme> <fichier d’entrée> <fichier de sortie>

Cette méthode permettra à un utilisateur de lancer l’application avec les arguments nécessaires.

# ANNEXE

**Classe ConcreteSolver**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Laboratoire #3 : Résolution de graphe

Cours : GTI310

Session : Automne 2010

Groupe : 01

Projet : Laboratoire #3

Étudiant(e)(s) : Gabriel Desmarais

Marie-Ève Benoit

Code(s) perm. : DESG24078908

BENM22568707

Chargée de lab. : Jean-François Franche

Nom du fichier : ConcreteSolver.java

Date crée : 2010-11-09

Date dern. modif. 2010-11-09

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

**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**;

}

}