Travail Pratique

**Département de génie logiciel et des TI**

|  |  |
| --- | --- |
| Numéro du laboratoire | 4 |
| 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 | 10 Décembre 2010 |

**Table des matières**

[1.Introduction 1](#_Toc279480335)

[2.Choix de design 2](#_Toc279480336)

[3. Implémentation 3](#_Toc279480337)

[4. Analyse du projet 4](#_Toc279480338)

[5. Conclusion 5](#_Toc279480339)

[6. Questions supplémentaires 6](#_Toc279480340)

[7. Manuel de l’usager 7](#_Toc279480341)

[ANNEXE 1](#_Toc279480342)

# Introduction

La compagnie Squeeze-media désire pouvoir améliorer ses performances en stockant plus d’image dans leur base de données. Pour ce faire, l’utilisation d’une méthode de compression similaire à celle du JPEG a été proposée.

Le projet a pour but de compléter un cadre de développement déjà existant (Squeeze Light Media Codec) afin d’y ajouter les classes permettant la compression d’images au format ppm vers un format quasi-JPEG et vice-versa.

Pour ce faire, une classe ConvertColor se chargeant des conversion de modèle de couleur, une classe BlocManager s’occupant de fractionner l’image en bloc de 8x8, une classe DCTManager pour la gestion des coefficients produits par la DCT, une classe Quantification servant à réduire le nombre de bits représentant une valeur et une classe ZigZag pour changer l’ordre des coefficients 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 les réponses aux questions supplémentaires, le guide d’utilisateur ainsi que l’annexe.

# Choix de design

Pour réaliser le projet, il était nécessaire de respecter certaines contraintes. Ces éléments consistaient à ne pas utiliser de code JPEG disponibles sur le web, garder le code simple malgré un facteur de compression moins optimal, permettre à l’utilisateur de choisir le facteur de qualité pour chaque image à coder (Valeur entière allant de 1 à 100) et l’application doit également permettre le décodage des images.

Plusieurs possibilités ont été envisagées afin de répondre aux demandes.

1. Utiliser une méthode de séparation en bloc différente de celle proposée par le document de lab.

# 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.

# 6. Questions supplémentaires

# 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