Département d'informatique et recherche opérationnelle

UNIVERSITÉ DE MONTRÉAL

Projet IFT3051

Une collection d'algorithmes de recherche opérationnelle en code source libre

Responsables	Adresses électroniques
Yann-Gael Gueheneuc	guehene@iro.umontreal.ca
Jean-Yves Potvin	potvin@iro.umontreal.ca

Membres du groupe	Adresses électroniques
Anouar BEN DHAOU	bendhaoa@iro.montreal.ca
Boubkre EL ALLANI	elallanb@iro.umontreal.ca
Khalid KANDOULI	kandoulk@iro.umontreal.ca
Bouchaib RIDA	ridabouc@iro.umontreal.ca

2 MAI 2005

TABLE DES MATIERES :

I. Contexte

La recherche opérationnelle est un domaine important en informatique et elle est très utilisée dans de nombreux domaines. Le besoin de logiciels facilitant l'exploitation des solutions offertes par la recherche opérationnelle s'avère donc très important.

À ce jour et à notre connaissance, il n'existe pas d'outil open source permettant de manipuler aisément des graphes et d'appliquer des algorithmes de recherche opérationnelle tout en réalisant deux objectifs : servir l'utilisateur normal voulant exploiter les services offerts par l'outil et préparant le terrain pour le programmeur qui veut ajouter de nouveaux algorithmes.

II. Introduction

Le but de ce projet est de mettre en place un outil ouvert (open source), qui permet l'implantation d'une collection d'algorithmes de recherche opérationnelle tout en préparant les bases nécessaires facilitant l'ajout, dans le future, de nouveaux algorithmes. Ceci est possible grâce au noyau de cet outil qui est en fait un meta-modele pour décrire les graphes et les méthodes de base pour les manipuler.

L'apport des patrons de conception tels que Decorator, Visitor, et Abstract Factory est primordial pour l'accomplissement de cette tâche.

Concernant le coté recherche opérationnelle, nous implémentons, en accord avec le superviseur du projet la liste des algorithmes suivants : BFS, DFS, Bellman, Dijkstra, Floyd, Kruskal, Postier Chinois, Prim.

III. Environnement de travail

Pour le développement de ce projet nous utilisons l'environnement de développement pour Java fourni avec la plate-forme Éclipse.

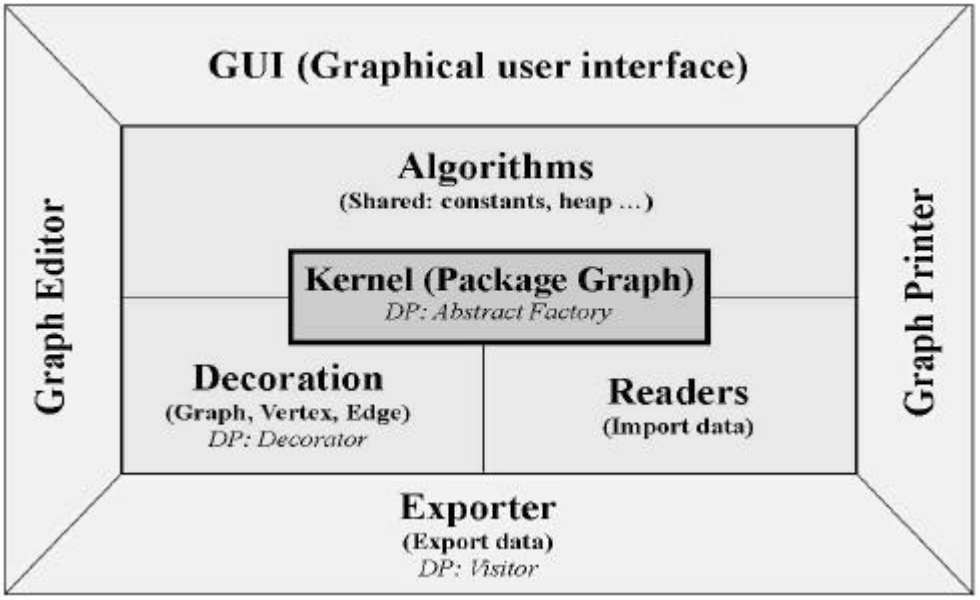
Ce logiciel est compatible avec la majorité des plateformes Linux, Windows.

Afin de mieux gérer le travail en équipe nous utilisons CVS qui est un système de travail collaboratif incorporé à Éclipse. Il permet à chacun des membres de l'équipe de

travailler sur une copie du projet tout en gérant les mises à jour et en offrant une sécurité grâce au backup qui permet de retracer l'évolution des versions du projet.
Enfin, pour mieux gérer les tests nous utilisons le framework JUnit. Il permet d'organiser les tests en créant une classe de test par classe à tester. Ceci nous permet de mieux contrôler le nombre de tests unitaires qui croît rapidement au cours du projet.

IV. Spécification technique du projet

1. Architecture du projet



NB : La mise à jour des couches externes n'affecte pas les couches internes.

Couche kernel

Ce module contient l'ensemble des classes représentant le noyau du logiciel. Il est utilisé comme base pour décrire les graphes et les méthodes pour implémenter les algorithmes. Le patron de conception utilisé est **Abstarct Factory**.

Couche algorithmique

- **Readers:** Ce module contient les classes responsables de l'importation des graphes de façon polymorphe.
- **Décoration:** Ce module est responsable de la décoration des trois classes Graph, Vertex et Edge propres à chaque algorithme. Le patron de conception utilisé est **Decorator**.
- **Algorithmes:** Ce module contient l'implémentation proprement dite des algorithmes.

Couche utilitaires:

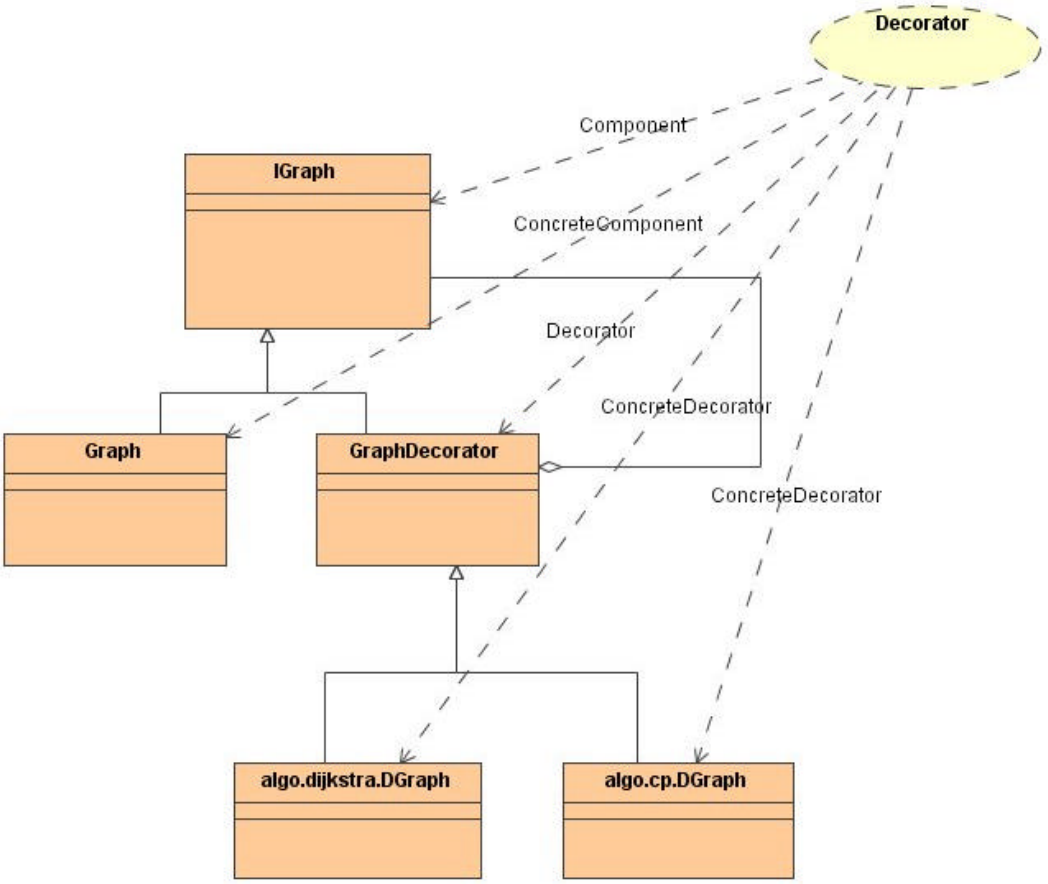
- **Exporter:** Ce module est responsable de l'exportation des graphes sous forme de fichier texte. Le patron de conception utilisé est **Visitor**.
- **GUI:** Représente l'interface graphique du logiciel.
- **Graph Editor:** Ce module permet d'éditer graphiquement le graphe de façon polymorphe.
- **Graph Printer:** Ce module est responsable de l'impression des graphes.

2. Patrons de conception

2.1 Decorator:

- Objectif : Ce patron de conception peut être assimilé à un emballage, il permet l'ajout dynamique de responsabilités à un objet. En effet l'héritage peut être utilisé dans ce cas, cependant ce concept est limité car il ne définit que des responsabilités statiques et manque donc énormément de souplesse.

- Diagramme de classe :

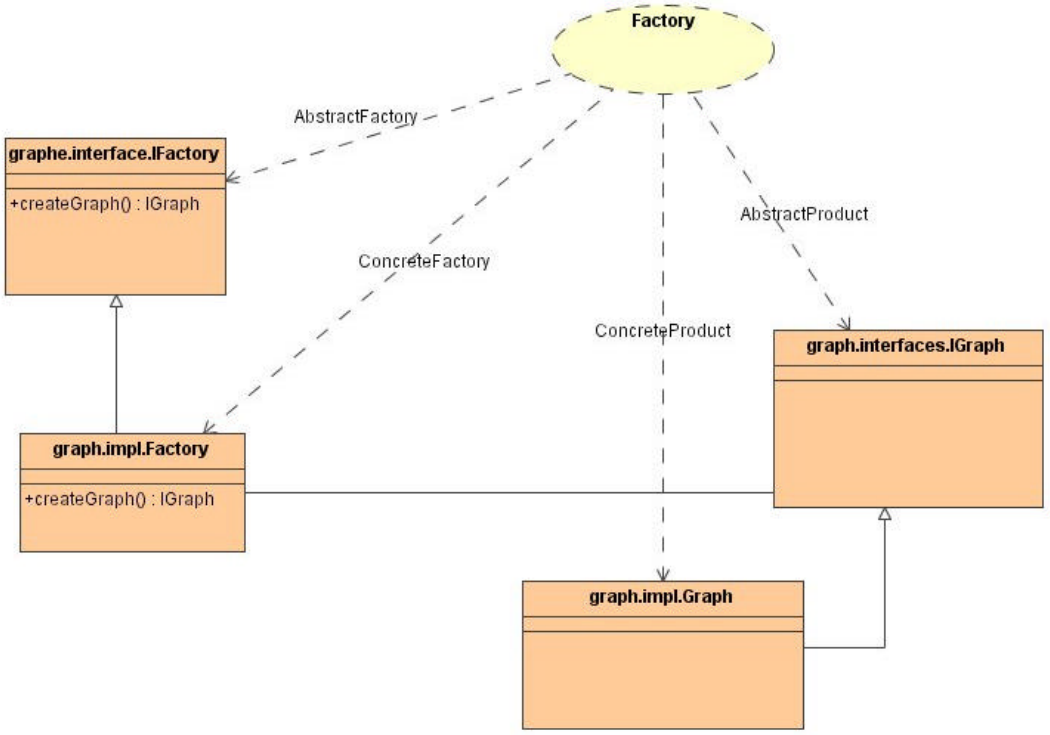


- Utilisation dans ORAlgo : ce patron est utilisé pour ajouter des attributs supplémentaires au Graph, Vertex et Edge de base.

2.2 Abstract Factory :

- Objectif : Cacher la nature des objets créés par des interfaces tout en maintenant leurs cohérences.

- Diagramme de classe :



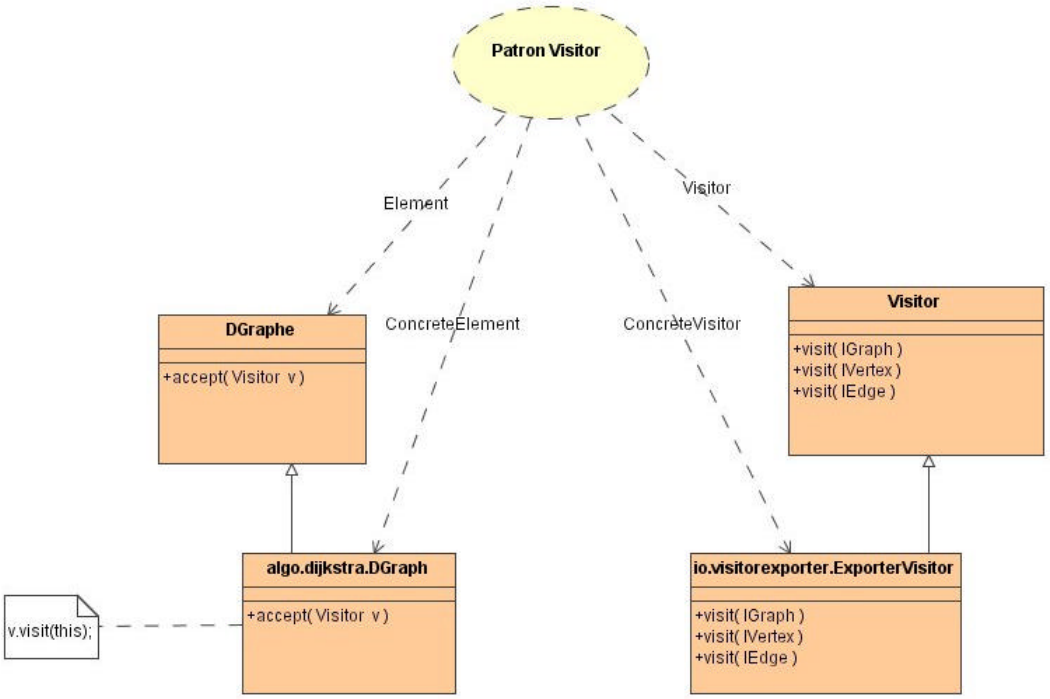
- Utilisation dans ORAlgo **:** ce patron permet de produire les objets du noyau à savoir Graph, Vertex et Edge d'une manière abstraite lors de l'importation.

2.3 Visitor :

- Objectif : Il présente une manière de séparer un algorithme de la structure d'un objet, ainsi que de spécifier comment les itérations se déroulent au sein de la structure de l'objet.

  Un jeu de classes forme la structure d'un objet. Chacune de ces classes dispose d'une méthode *accept* qui prend un objet visiteur en argument. Le visiteur est une interface ayant différentes méthodes *visit* pour chacune des classes. La méthode *accept* d'une classe appelle ainsi la méthode *visit* qui correspond à sa classe.

- Diagramme de classe :



- Utilisation dans ORAlgo **:** ce patron est utilisé dans le module d'exportation afin de récupérer la description textuelle des graphes en visitant les sommets et les arcs.

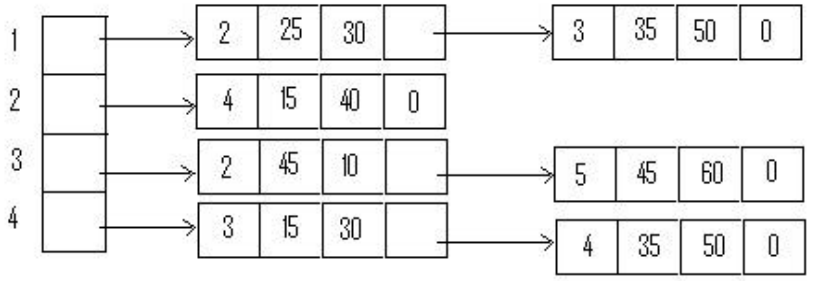
3. Structure de données utilisée

Pour le choix de la structure de données, deux implémentations des graphes sont couramment utilisées : la matrice d'adjacence et la liste d'adjacence.

Ce choix se fait en fonction des opérations que l'on veut faire, et de la densité du graphe, c'est-à-dire du rapport entre le nombre d'arcs et le nombre de sommets.

Dans le cadre de ce projet on a adopté comme solution la liste d'adjacence, vu la nature des graphes qui ne sont pas en général fortement connexe.

Donc dans notre cas, le graphe est un vecteur de sommets, chaque sommet contient une liste d'arcs sortants et chaque arc contient le noeud représentant le sommet de destination.



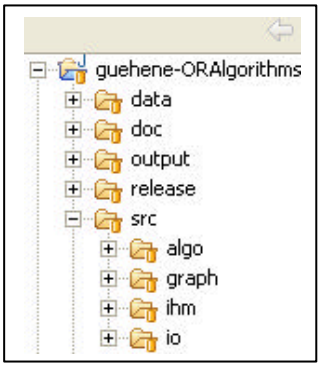
Exemple de liste d'adjacence

4. Organisation du projet

4.1 Structure du fichier d'importation/exportation

Exemple de fichier d'importation	Description
Réseau routier	Titre du graphe
##4	Nombre de sommets (doit commencer par '##')
Montréal	Identificateur de sommet
Québec	Idem
Ottawa	Idem
Toronto	Idem
##5	Nombre d'arcs (doit commencer par '##')
route1 Montréal Toronto 10	Identificateur arc, source, destination, poids (décoration)
route2 Montréal Québec 20	Idem
route3 Québec Toronto 30	Idem
route4 Québec Ottawa 40	Idem
route5 Ottawa Toronto 50	Idem

4.2 Structure du projet



- **data :** Ce répertoire contient les fichiers plats pour l'importation et l'exportation des graphes.

- **doc :** contient la documentation technique du projet (java doc)

- **release :** contient les versions opérationnelles et stables.

- **src :** contient le code source du projet.
  - **graph** : constitue le vrai noyau du projet, c'est un ensemble de classes et interfaces qui représente la structure interne des graphes.

  - **algo** : représente la collection des algorithmes implémentés tels que : BFS, DFS, Dijkstra etc…

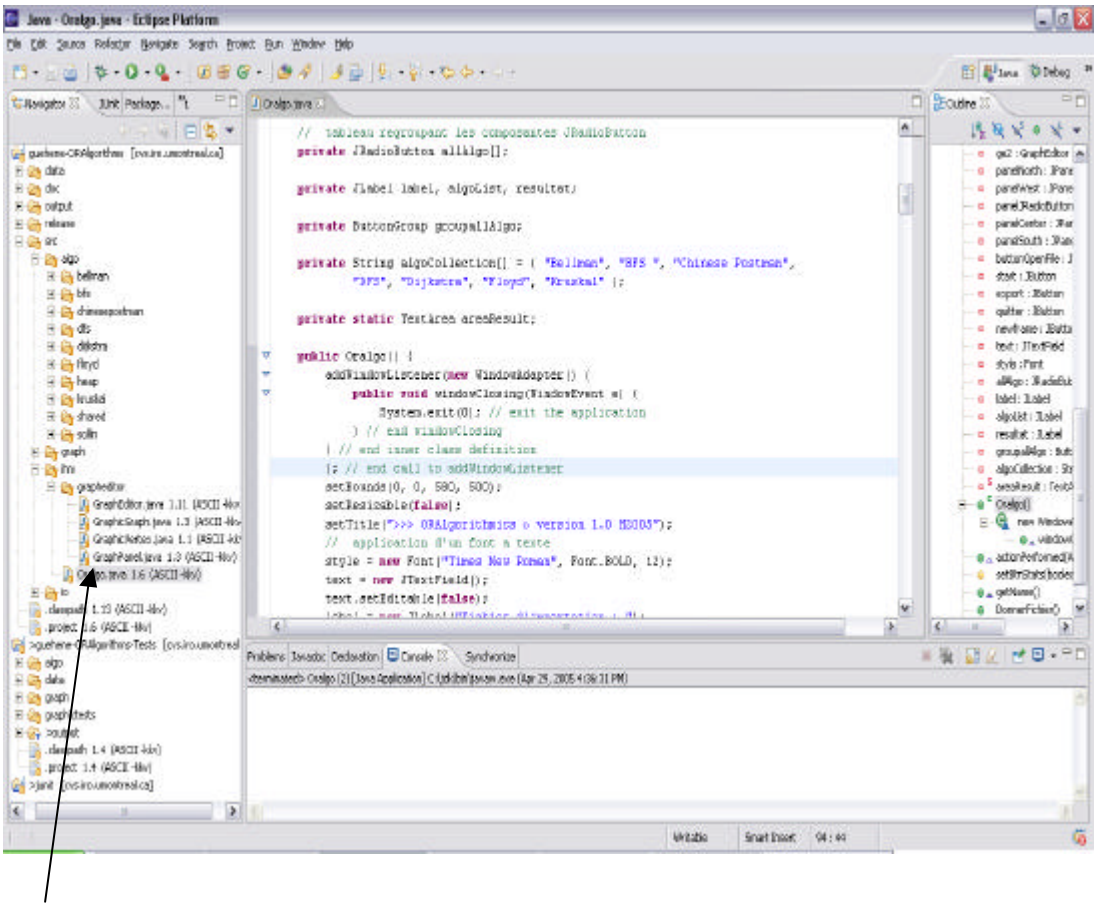
  - **ihm** : représente l'interface graphique. Il est composé de deux parties :
    - interface de base donne le choix des algorithme et celui du fichier d'importation.
    - éditeur graphique du graphe

  - **io** : le pakage des entrées/sorties du projet comme l'exportation.

V. Manuels

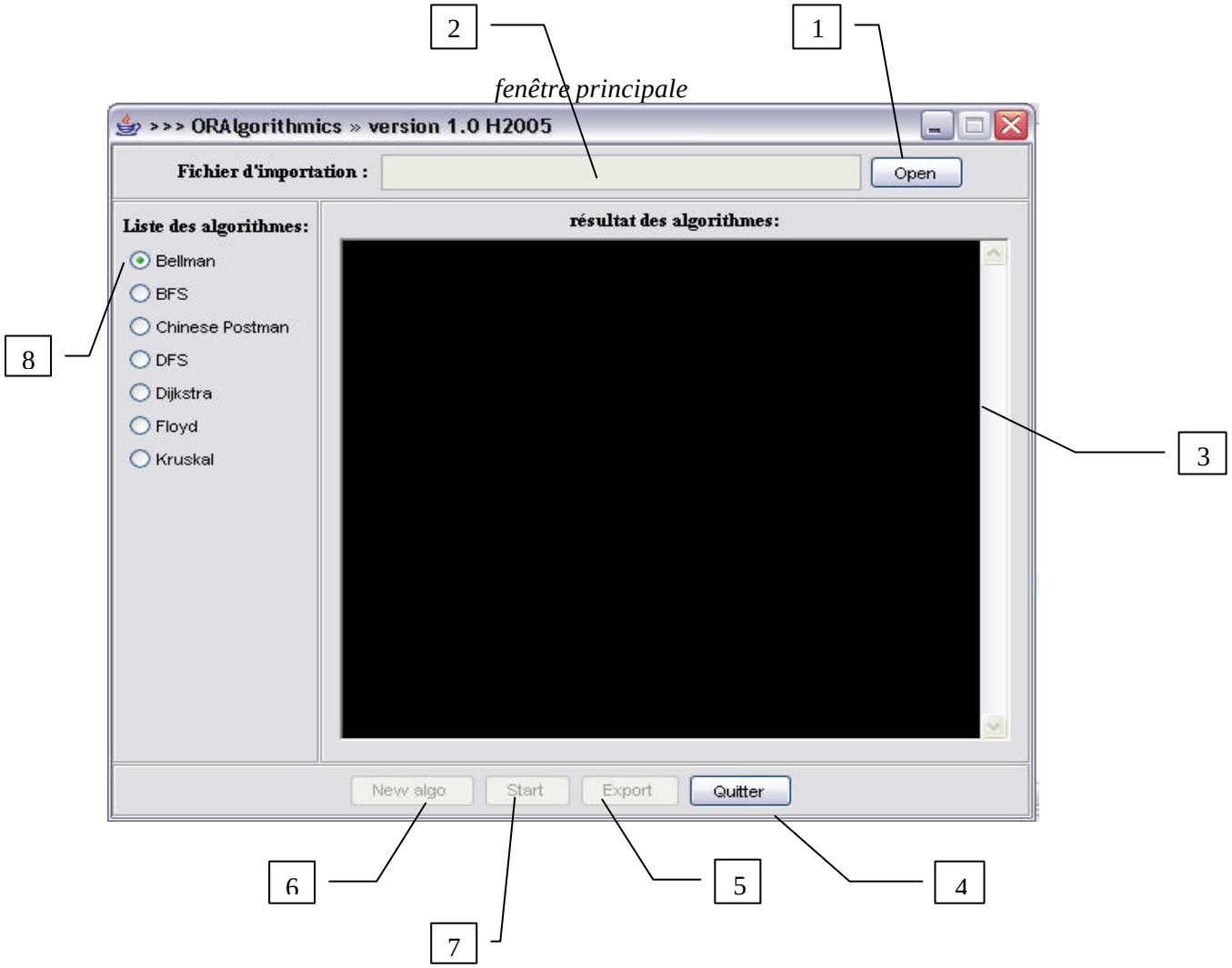
1) Manuel utilisateur : Présentation de l'interface graphique.

Pour lancer l'exécution de *ORAlgorithm*, il faut exécuter le fichier principal qui se trouve dans *guehene-ORAlgorithms\src\ihm\oralgo.java*.



La classe principale du projet

fenêtre principale



1- Ouvrir l'explorateur pour sélectionner le fichier d'importation.
2- Zone d'affichage du chemin absolu du fichier à importer.
3- Zone d'affichage du résultat des algorithmes sous forme de texte.
4- Fermer la fenêtre principale.
5- Enregistrer le résultat d'un algorithme dans un fichier.
6- Exécuter un nouvel algorithme.
7- Démarrer l'exécution de l'algorithme.
8- Liste des algorithmes implémentés.

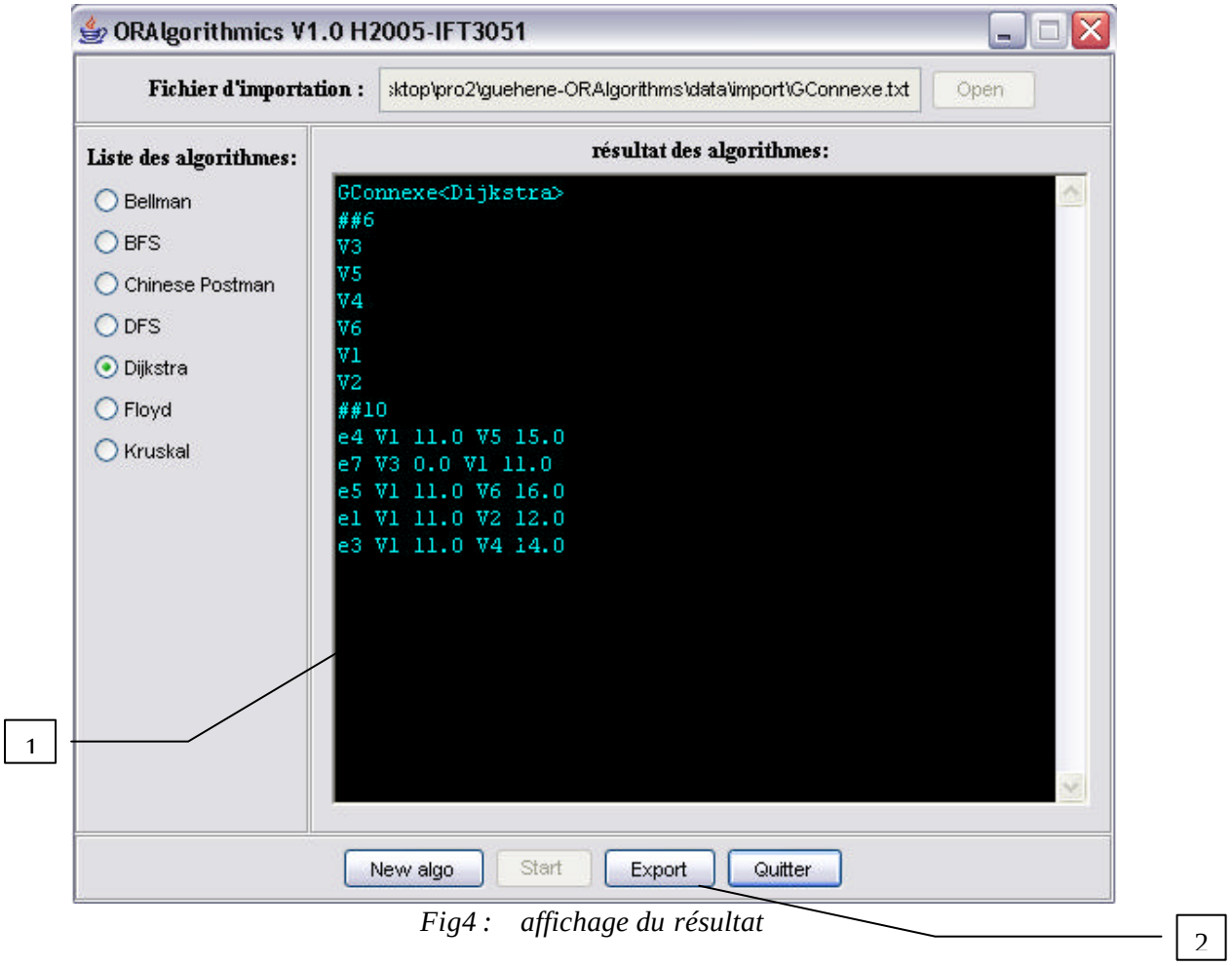


Fig4 : affichage du résultat

1- Affichage du résultat de l'algorithme.

2- Sauvegarder le résultat obtenu dans un fichier.

2) Manuel programmeur (procédure d'implémentation d'un nouvel algorithme)

Pour l'exemple nous prenons comme nom d'algorithme à ajouter : ***algoNouveau.***
Voici les phases à suivre pour ajouter ce nouvel algorithme.

2.1 Création du package propre à l'algorithme *algoNouveau.*

Dans le répertoire **src/algo**, ajouter un nouveau répertoire sous le nom *algoNouveau*. Il représentera le package propre à *algoNouveau*.

2.2 Création du reader.java (extends ConcreteReader)

- Préparer un reader pour l'importation du fichier correspondant au nouvel algorithme. (On peut reprendre l'un des reader déjà existant)
- Dans ce reader, mettre les bonnes valeurs aux deux constantes **NBCOLVERTEX** et **NBCOLEDGE** correspondant respectivement au nombre de colonnes de sommets et au nombre d'arcs;
- Personnaliser les paramètres d'instanciation du constructeur *DVertex* dans la méthode *createVertex* et du constructeur *DEdge* dans la méthode *createEdge*.

2.3 Étape de décoration :

a) Vertex

- Créer la classe *DVertex.java* qui doit étendre la classe *VertexDecorator*.
- Personnaliser les arguments du constructeur *DVertex***.**
- Ajouter les attributs correspondants à la décoration comme variables de classe.
- Ajouter les méthodes relatives à la décoration (comme *set* et *get*) .
- Ajouter la méthode *toString*, qui est appelée lors de l'édition graphique.
- Ajouter la méthode *accept(Visitor v)* relative au patron de conception *Visitor*.
  - Ajouter dans le fichier *io/visitorexporter/ExporterVisitor.java* l'implémentation de la méthode déclarée dans l'interface.

b) Edge

- Créer la classe *DEdge.java* qui doit étendre la classe *EdgeDecorator*
- Personnaliser les arguments du constructeur *DEdge*
- Ajouter les attributs correspondants à la décoration comme variables de classe .
- Ajouter les méthodes relatives à la décoration (comme *set* et *get*)
- Ajouter la méthode *toString*, qui est appelée lors de l'édition graphique.
- Ajouter la méthode *accept(Visitor v)* relative au patron de conception *Visitor*.
  - Ajouter dans le fichier *io/visitorexporter/ExporterVisitor.java* l'implémentation de la méthode déclarée dans l'interface.

c) Graph

- Créer la classe *DGraph.java* qui doit étendre la classe GraphDecorator
- Personnaliser les arguments du constructeur DGraph
- Ajouter les attributs correspondants a la décoration comme variable de classe.
- Ajouter les méthodes relatives a la décoration (comme set et get)
- Ajouter la méthode toString, qui est appelée lors de l'édition graphique.
- Ajouter la méthode *accept(Visitor v)* relative au patron de conception Visitor.
  - Ajouter dans le fichier *io/visitorexporter/ExporterVisitor.java* l'implémentation de la méthode déclarée dans l'interface.

2.4 Étape d'implémentation de l'algorithme *algoNouveau* :

- Créer la classe *algoNouveau.java*, qui implémente concrètement le nouvel algorithme.
- Personnaliser les arguments du constructeur algoNouveau qui doit au moins recevoir un DGraph.

2.5 Modification au niveau interface utilisateur : (classe oralgo.java)

- Ajouter dans le tableau *algoCollection* le nom de l'algorithme.
- Ajouter la méthode *traitalgoNouveau* qui lance l'exécution de l'algorithme.
- Dans la partie qui traite le bouton *start* de la méthode *actionPerformed*, ajouter l'appel à la méthode *traitalgoNouveau*.

La nouvelle méthode *traitalgoNouveau*

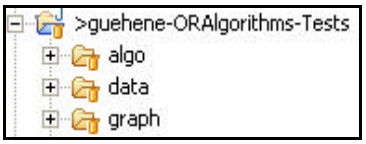
```
public void traitalgoNouveau () {
      algo.algoX.Reader r = new algo.algoX.Reader(getName());
      //…
      if (r.isReadOk()) {
            algoX dj = new algoX(r.getGraph());
            areaResult.setText(dj.getResult());
       //…
       }
}
```

Dans la partie qui traite le bouton 'start' de la méthode 'actionPerformed'

```
/* indexAlgoNouveau : est l'index de algoNouveau dans la zone algorithme de
l'interface graphique*/
   if (allAlgo[indexAlgoNouveau].isSelected()) {
         traitalgoNouveau();
}
```

VI. Test JUNIT

1. **Structure du projet test**



- **algo** : pakage des tests des algorithmes.
- **data :** les fichier d'import/export.
- **graph :** contient les tests du graphe de base

Remarque: Ce projet doit référencer JUnit.

2. Statistiques des tests

Package	**Nombre de tests**
graph	895
algo.bellman	112
algo.bfs	85
algo.dfs	98
algo.chinesepostman	187
algo.dijkstra	171
algo.floyd	181
algo.kruskal	179
algo.prim	194
algo.shared.heap	28
io (import, export)	121
Total	**2251**

VIII. Discussion

1. Difficultés rencontrées

- Difficulté au niveau de certains algorithmes tel que le Postier chinois (pour un cycle non Eulérien) et le voyageur de commerce.
- Difficulté d'application des patrons de conception

2. Points forts du logiciel

- L'implémentation des algorithmes est indépendante du noyau.
- L'interface graphique est indépendante de la structure interne des graphes et des algorithmes.
- Durant l'implémentation des algorithmes on a pris en compte la complexité de certaines méthodes afin de les optimiser. Par exemple on a introduit la notion du monceau (Heap.java) qui s'exécute en ordre de log (n).

3. Amélioration et ajouts possibles/futures

- Implémenter d'autres algorithmes de recherche opérationnelle.
- Choisir dynamiquement la structure de données du graphe lors de l'instanciation, en fonction du nombre d'arcs et du nombre de sommets.
- Appliquer le MVC au niveau du GUI.
- Utiliser le parallélisme pour mieux exploiter le Heap qui gère l'accès à certaines méthodes considérées comme sections critiques.
- Implémenter un graphe non orienté basé sur la structure actuelle (graphe oriente).
- Impression du graphe

4. Profits personnels.

L'apport de ce projet dans notre formation est très important. Il nous a permis de revoir, d'appliquer et d'approfondir plusieurs notions vues dans des cours tels que le génie logiciel, programmation java, recherche opérationnelle, structures de données et enfin l'algorithmique.

D'autre part il nous a permis de nous familiariser avec l'environnement de développement Eclipse très utilisé dans l'industrie, de comprendre et d'utiliser JUnit et enfin de comprendre l'importance de CVS dans le travail de groupe.

Finalement, de par l'aspect open source, le travail permettra également de faire connaître le département informatique de par le monde.

5. Remerciement

Nous tenons à remercier nos Responsables du projet, Mr Yann-Gaël Guéhéneuc et Mr Jean-Yves Potvin pour leur aide, leur disponibilité et leur support précieux durant les différentes phases de réalisation du projet.

En plus nous remercions toute personne ayant participé d'une façon directe ou indirecte à la bonne marche de ce projet.

VIII. Bibliographies

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Design Patterns – Elements of Reusable Object-Oriented Software.

Addison-Wesley, 1st edition, 1994. isbn: 0-201-63361-2.

- Frederick S. Hillier and Gerald J. Lieberman. Introduction to Operations

Research. Holden-Day, 4th edition, January 1986. isbn: 0-8162-3871-5.

- Ian H. Witten and Eibe Frank. Data Mining: Practical Machine Learning

Tools and Techniques with Java Implementations. Morgan Kaufmann,

1st edition, October 1999. isbn: 1-55860-552-5.

IX. Références

- Alfred Aho, John Hopcrooft, Jeffrey Ullman, The design and analysis of computer algorithms.
- http://gee.cs.oswego.edu
- http://students.old.qmul.ac.UK/aduni/05_algorithm/handouts/reciation_07.html
- http://www.patterndepot.com/put/8/JavaPatterns.htm
- http://www.dofactory.com/Patterns/PatternDecorator.aspx