Notes On ORAJ's design and implementation

by David St-Hilaire

August 30, 2006

Contents

1 Introduction

Operations research is a vast field of study which possesses many concrete applications and that is also closely related to computer science. Many commercial software of applied operations research currently exist on the market, but to this day and to the best of our knowledge, none existed as open source software projects.

An applied operations research open source project would be of great benefit to the community because it will be possible to use that software in its whole or just some of its modules and because the community will be able to contribute to the advancement of the project. Thus, the more people that would use and contribute to the project, the better and more complete the software will become, benefiting from the experience of many programmers or users of completely different background fields.

Our project focuses on such ideals that promote free operations research software of good quality. The project was named ORAJ, which stands for Operations Research Algorithms in Java [1] and which also means thunderstorm in French. The project has currently three main different uses. First, it implements different types of graphs that constitute the basis of a majority of operations research algorithms. Secondly, the project collects algorithm to apply on these graphs. Finally, a graphical user interface exists to enable an easy usage of thoses algorithms for non-programming users.

To ensure the quality of the software, its detailed design follows best practices, in particula design patterns. The GoF [9] design patterns *Decorator*, *Observer* and *Abstract Factory* were used in ORAJ's design/implementation. By using these patterns, the modularity and reusability of the software is promoted. Indeed, a modular open source software facilitates the concurent contributions of different programmers, if each one focuses on different modules. Also, reusable and modular software promotes the different ways that the software can be used by different users. For example, a user might use the software through its graphical user interface, but another might just want to include an algorithm implementation in his own software.

2 Technical specification of the project

General project architecture

The project is divided into four main modules. These modules are delimited in four main packages. Figure 1 shows the general package structure of ORAJ. The first module, *oraj.graph*, constitutes the core of the project. It contains the different graph implementations that are used by the algorithms and by the graphical user interface (GUI). The second module is located in the *oraj.algo* package. This package contains all the implemented algorithms that operates on the graph model defined in *oraj.graph*. The third module, in the *oraj.io* package, is an independant module containaining utilities to export graphs from the *oraj.graph* package to XML files and to parse XML files into graphs. The last module is the graphical user interface which is located in the package *oraj.gui*. This GUI is an interface that makes possible an easy use of the algorithms for users not familiar with programming. This package depends on all the other packages because it is an interface providing all the capabilities implemented in the software.

algo.graph package's design and architecture

To factorize some common code between the different graph implementations, a set of abstract classes (BasicEdge, BasicVertex, BasicGraph) was created. These contain partial implementation that roughly corresponds to the behaviour of a directed graph. Thus, a graph implementation could either extend theses classes and overwrite the methods that need some modifications.

The family of classes extending IFactory are the application in ORAJ of the *Abstract Factory* design pattern. Figure 2 illustrates the *Abstract Factory* pattern implementation. The use of this pattern reveals itself in the implementation of the input/output package and of the GUI package. It abstracts and standardizes the creation of graph elements thus making this process standard and completely flexible as an IFactory reference might point toward a DirectedFactory or an UndirectedFactory element. We also added the functionnality getIFactory() to the IGraph, IVertex and IEdge interface so that the type of a graph can be easily obtained. An other elegant use of this design pattern is achieved through the definition of Euclidean graphs.

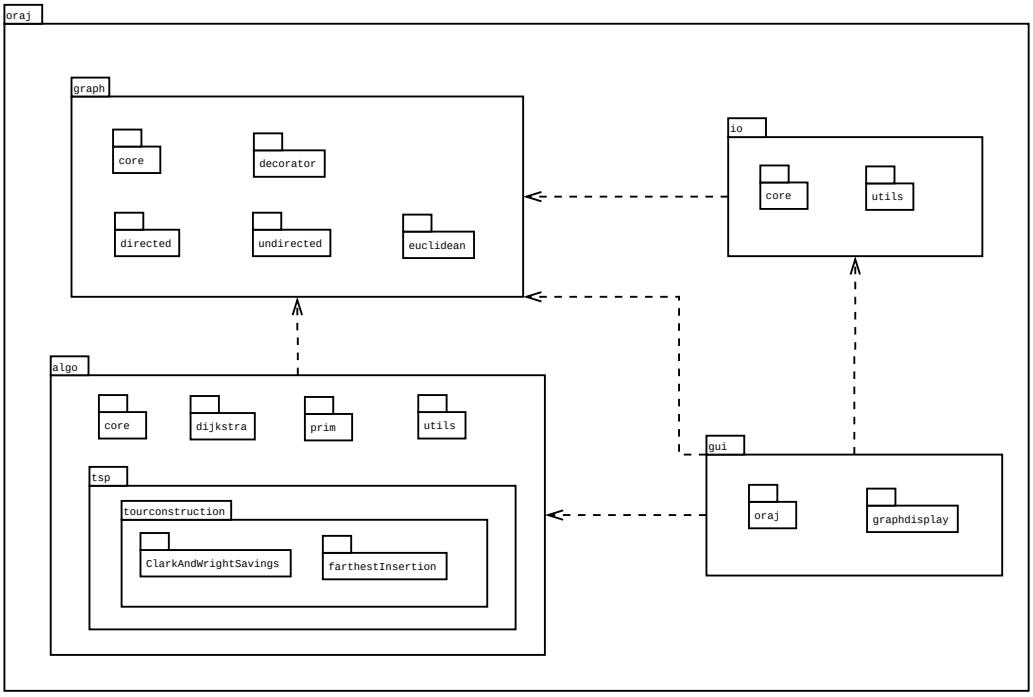


Figure 1: ORAJ's global package diagram. Some of the inner package are illustrated to show the content of the main packages, but not all of the inner packages are shown.

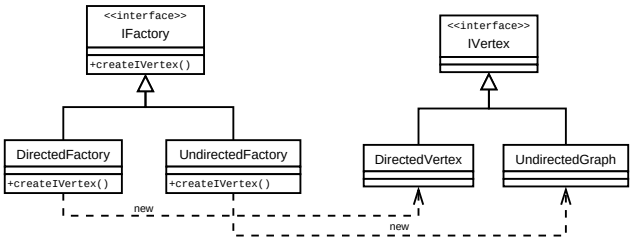


Figure 2: Diagram illustrating the accomplishment of the *Abstract Factory* design pattern.

An Euclidean graph IFactory takes an IFactory as parameter at construction. Thus an Euclidean graph can be either directed by passing a DirectedFactory as parameter or undirected by passing an UndirectedFactory as parameter.

It is possible to add information on graph elements by using the *Decorator* design pattern. The use of this design pattern is shown in Figure 3. This implementaion is particular because a Component has bidirectionnal references to adjacent elements in the decorator scheme. It can access the element that is decorating and the element decorating it. The need of bidirectionnal references comes from the fact that at one point, the IGraph element might have a references pointing the an inner decorator element, but not the core implementation. This means that to be able to access all data stored in this element, it must be possible to go to the highest decorator and then go through all the decorator elements until the core implementation is reached. A way to improve this would be to make sure that the IGraph element always possess a reference to the innermost element, the core implementation. Then access to the decorator would be from the interior and going toward the exterior of the decorator shell, until the highest decorator is reached. This has not yet been implemented.

Two types of decorators have been implemented in ORAJ, *external* decorators and *internal* decorators. External and internal decorators distinguishes decorators that adds data supplied and known by the user and decorators that adds data only known by the internal procedures of ORAJ, such as an Algorithm implementation, and should not be known by a user. For example, if an algorithm needs to construct a certain path, it might use a

PreceedingVertexDecorator layer but, this process should be transparent to a user that person does not need to know how the algorithms operate. The user only wants to input a graph and get the result from the algorithm. In contrast, a user who wants to set Cartesian coordinates to a vertex will use a CartesianCoordVertexDecorator and will be aware of the presence of this layer in the graph used.

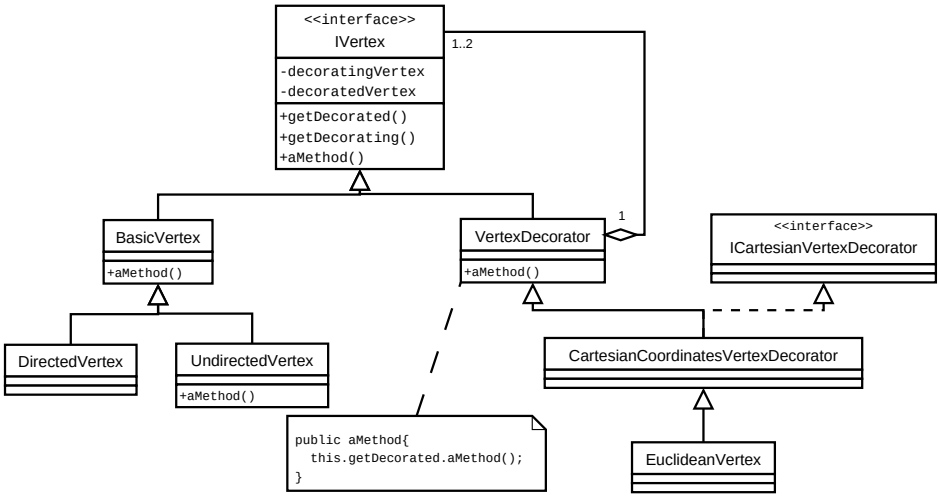
Detailled explanation on the implementation of Euclidean graphs must be given because this process has hacked a bit the architecture mentionned above. Euclidean graphs are conceptually considered as a graph type, and thus should be implemented as a core implementation of the interface IGraph. But, in fact, the difference between an Euclidean graph and a normal graph is the presence of additionnal data, which are Cartesian coordinates for the vertices and a distance for the edges. Still, an EuclideanFactory has been implemented which takes as argument another IFactory, see page 2. For technical reasons that are related to the I/O module and the use of a consistant IFactory interface, this factory does *NOT* return an EuclideanVertex or an EuclideanEdge when calling the methods createVertex or createEdge. A user will need to explicitly add this layer by himself. For example:

```
IFactory factory = new EuclideanFactory( new DirectedFactory );
IVertex euclideanDirectedVert
    = new EuclideanVertex( factory.createVertex(), xCoord, yCoord );
```

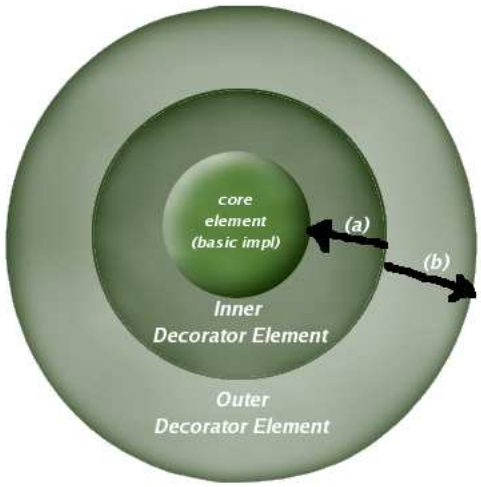
This use is certainly everything but intuitive, but it is the only way so far that it can be compatible with the graph readers and writers located in the *oraj.io* package.

Notes on the data structures in ORAJ

The data structure used to represent graphs in ORAJ is close to an adjacency list abstract data type (ADT). Edges only know the two associated vertices. The vertex elements, upon instanciation, have no references to adjacent vertices. Only once the vertices were inserted in the graph, the edges may be added into the graph, and the neighbouring vertex references are updated. Thus, once all elements have been correctly added in a graph, a vertex knows all its adjacent vertices by enumerating all its out-going edges and looking at the destination vertices. A code example is shown in Figure 4 that illustrates the case in which a user wanted to retreive the vertex going from an IVertex



(a) Decorator design pattern implementation



(b) Example of the uses of decorators

Figure 3: Decorator usage illustration. ORAJ's *Decorator* implementation is illustrated in 3(a). An example of the usage of a decorator is shown in 3(b). In this example, an inner element can access it's core element by calling the getDecorated() method (3(b).a) or it can access it's outter element by calling the getDecorating() method (3(b).b).

```
IEdge searchedEdge = null;
Iterator edgeIt = vertA.getOutEdges();
IEdge currentEdge;
while(edgeIt.hasNext() && searchedEdge == null){
  currentEdge = (IEdge)edgeIt.next();
  if(currentEdge.getDestination().vertexEquals(vertB)){
    searchedEdge = currentEdge;
  }
}
return searchedEdge;
```

Figure 4: Example code that finds the edges going from vertA to vertB.

'vertA' and going to an IVertex 'vertB'.

Figure 5 illustrates the choice of the abstract data structure implied by the interfaces of the *oraj.graph.core* package and used in the implementation of ORAJ.

oraj.algo package's design and architecture

The behaviour of an algorithm is defined by the abstract class Algorithm from the *oraj.algo.core* package. Figure 6 illustrates the sequence of calls occuring when preparing an algorithm instance and executing the algorithm on a certain graph. This class possesses two abstract methods, getAlgoName() and execute(). The first method is used in the GUI to display the algorithm name and the second method defines the algorithm itself. The initialisation process of an algorithm should be made through the verifyAndPrepareAlgo() method, which locates the class named *AlgoName*GraphVerificator, where *AlgoName* is the name of the algotithm, which should be a class extending the class GraphVerifier and be located in the same package as the algorithm implementation. Then, the GraphVerifier class, if found, is instanciated and used to verify that all the required external decorators are present, if applicable, and to initialize the internal decorators. This verification is achieved by implementing the required abstract methods of the GraphVerifier abstract class in the appropriatly named Algorithm's graph verificator. All the methods which are not needed should be implemented as empty body methods.

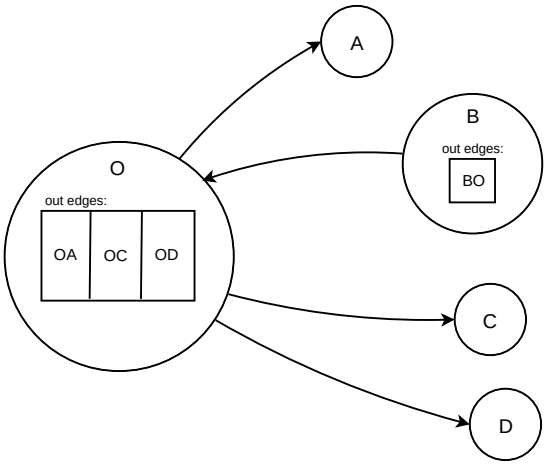


Figure 5: Diagram illustration the ADT used in ORAJ to represent graphs. Every vertex knows only its adjacent vertices by enumerating its out-going edges list and verifying the destination of each of does.

Current algorithm implementations are located in separate packages in the *oraj.algo* package. In particular, algorithms which are related to the travelling salesman problem are currently in the *oraj.algo.tsp* package.

oraj.io package's design and architecture

The input/output module is a simple parser that exports a graph into an XML data file format and that imports an XML file into a graph. The XML file format focuses on simplicity for users unfamiliar with such data file formats. A simple example is shown below.

```
<?xml version='1.0' ?>
<!DOCTYPE graph SYSTEM "graph.dtd">

<graph title="ExampleOfDirectedEuclideanGraph"
  type="directed" euclidean="true">

  <vertex name="A">
    <EuclideanVertex x="0" y="0"/>
  </vertex>
```

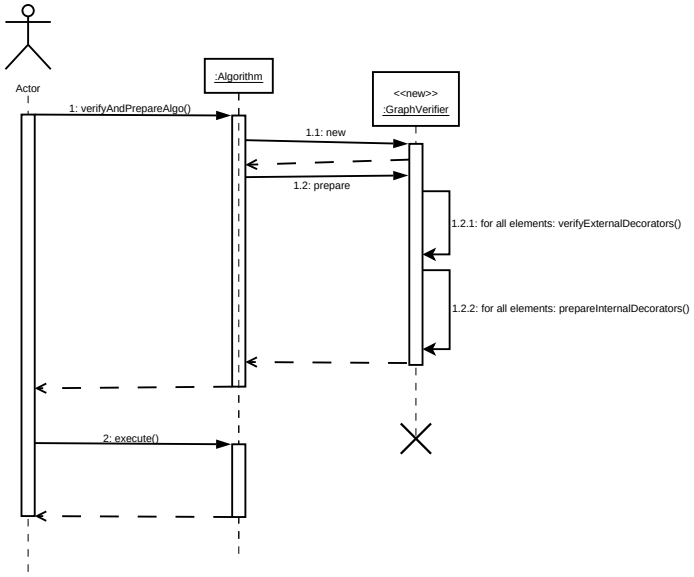


Figure 6: Sequence diagram illustration the use case of preparing and executing an algorithm.

```
  <vertex name="B">
    <EuclideanVertex x="2" y="2"/>
  </vertex>
  <vertex name="C">
    <EuclideanVertex x="-2" y="5"/>
  </vertex>

  <edge name="eAB" src="A" dest="B">
    <EuclideanEdge />
  </edge>
  <edge name="eAC" src="A" dest="C">
    <EuclideanEdge />
  </edge>

</graph>
```

The file graph.dtd is used to validate the general structure of the XML file but this verification is not strict to enable the use of reflection to create or read a graph. A strict document type definition (DTD) would need to hard code all the existing decorators, which would deprive the software of flexibility. Thus, a graph is strictly defined has possessing only edge and vertex XML elements, but the vertex and edge elements may hold any elements nested inside them. The expected use is as follow, vertex and edge inner elements must be *external* decorators that are to be added to the core vertex or edge element. These decorators XML elements must have *the exact same name* as the external decorator class that will be instantiated. A decorator XML element's attribute name has no importance in the xml file, but these attributes order must be the same order of appearance as in the decorator class decorators. As in the example above, the EuclideanVertex(IVertex decorated vertex, Double xCoord, Double yCoord) may be added to a vertex by adding the attribute

```
<EuclideanVertex x='aDoubleValue' y='anotherDoubleValue' />
```

but may not be

```
<EuclideanVertex y='aDoubleValue' x='anotherDoubleValue' />
```

because the instance EuclideanVertex decorator will have aDoubleValue as its x coordinate and anotherDoubleValue as its y coordinate, because the

constructor located in the EuclideanVertex class has the order of parameters appearance: xValue, yValue. Finally, a decorator element may not have any element nested inside it.The graph.dtd file is well documented and should be referenced for further explanations on the file format used.

As mentioned above, the implementation of the I/O package was achieved through the use of reflexion. For example, when a reader reads a decorator, it searches the *oraj.graph.decorator.implementation.external* package and try to locate the corresponding class. If found, the constructor with the corresponding number of arguments will be retrieved and the new instance will be created by using the parameters found in the XML decorator element as values for the constructor's parameters. Thus, *an algorithm implementation should not have more than one constructor for a given number of parameters.* The use of reflection implies that no decorator names were hard coded and when some new external decorators are added in the futur, no changes will be required in either the reader nor the writer.

An extra tool has been added in the *oraj.io.util* package to create random graphs from certain parameters. These parameters are the number of vertices, the type of graph (directed or undirected) and the percentage of completeness between 0 to 1.0. Thus, a graph with 0 completeness will possess no edges and a graph with 1.0 completeness will be a complete graph. A graph with a percentage of completeness of x, stricly between 0 and 1.0, will possess $x * N$, where N is the number of arcs/edges present if the graph whould be complete and these arcs/edges will be added randomly in the graph.

oraj.gui package's design and architecture

The GUI package contains Swing elements used to create a simple interface that can be used to apply implemented algorithms on graphs created from an XML file. Again, reflection was used to create the menu items associated with the implemented algorithms to ensure that the addition of new algorithms to ORAJ will not require any modifications to the GUI. All class contained in the oraj.algo package that extends the Algorithm abstract class are used to create a menu item, assuming that no errors happened during the instantiation of the Algorithm.

The GUI divides in two small modules. *oraj.gui.graphdisplay* allows diplay-

ing a graph instance in a modified JPanel (GraphDisplay). The GraphDisplay class uses the OrajModel class as its model, following MVC architetural pattern. Thus, the graph is contained in the model and the GraphDisplay displays the graph. Controllers in the graphdisplay package are mainly used to manage events relevant to the graph display. The GraphDisplay can be included in a JPanel which adds a functionnal bar that adds different functionnalities such as vertex selection from a list on the graph displayed.

In the package *oraj.gui* can be found the main graphical user interface classes and interfaces. The main gui frame, named Oraj extends the javax.swing.JFrame class and is the access point to the graphical user interface. This Oraj class is also designed following an MVC approach, the Oraj class being the view, the OrajGuiController being the controller of that view, and the OrajGuiModel being the model. OrajGuiModel serves as model to both Oraj and GraphDisplay and extra care must be taken to ensure that both GraphDisplay and the main frame are linked to the same model to keep coherence in the GUI. In the main frame, the event controller as been kept in the view class for simplicity and the controller has the responsability of modifying the model as needed. This frame enables the imporation and exportation of graphs, the application of all implemented algorithms on that graph, and the generation of a random graph.

Messages between GUI elements are not sent through the MVC architecture to the model. Instead, a more flexible and easy to modify implementation was chosen, with the *Observer* design pattern. Figure 7 illustrates the collaboration diagram of the selection of a new root vertex by clicking directly on the GraphDisplay instance.

3 Contributor's guide

Addition of a New Algorithm

The suggested steps for the implementation of a new algorithm in ORAJ are as follow.

1. Create an appropriate package in the root packate *oraj.algo*.
2. (optionnal): If internal decorators must be implemented to add data

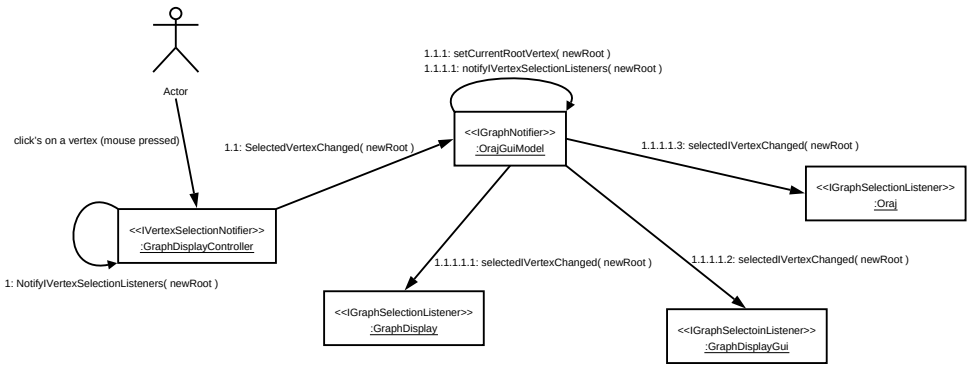


Figure 7: Collaboration diagram which illustrates the selection of a new root vertex by clicking directly on the graph display instance.

on the graph used by the algorithm, a new internal decorator should be implemented in the *algo.graph.decorator.implementation.internal* package by extending the abstract class. {Graph,Vertex,Edge}Decorator. If desired, an interface can also be created, defining the behaviour expected by that decorator and placed in the *oraj.graph.decorator.internal* package.

3. Implement a new class extending GraphVerifier. In the verifyExternalDecorator methods, a GraphVerifierException should be thrown if the scanned graph is problematic. For examples, please refer to the already implemented GraphVerifiers.

4. Implement a new class extending the abstract class Algorithm. This class must implement the two abstract methods getAlgoName() and execute(). The execute method should execute the algorithm on the initial graph and set the resulting graph at the end of the execution of the algorithm by calling the method super.setResult(IGraph resultGraph). This graph must be built manually by the contributor so that it reflects the result of the algorithm. The toString() method should return a String which, if the execute() method has been run, should return a verbose output of the algorithm's results. Others ascessors can be implemented for the usage of programming-capable users.

4 User's guide

Simple IGraph Creation

The graph creation process is quite simple. The process divides in two major steps: instantiation and insertion of vertices and edges in the graph. The first sub-step of the instantiation process is the instantiation of the desired IGraph implementation. The second sub-step is the instantiation of all the vertices with the corresponding IVertex implementation. The third sub-step is the instantiation of the IEdges which marks the end of the instantiation step. The IEdges are instantiated last because they need references on the IVertex on which they are associated at instantiation. The first sub-step of the insertion step is, the insertion of the IVertices in the IGraph. The second sub-step is the insertion of the IEdges instances in the graph, which marks the end of the insertion process. Here is a short example of a directed Euclidean graph creation containing two vertices and one edge linking them together.

```
IFactory factory = new EuclideanFactory(new DirectedFactory());
IGraph g = factory.createGraph('graphName');
IVertex v1 = new EuclideanVertex(factory.createVertex('vertexName'), x, y);
IVertex v2 = new EuclideanVertex(factory.createVertex('vertexName2'), x2, y2);
IEdge e = new EuclideanEdge(factory.createEdge('edgeName', v1, v2 ));
g.addVertex(v1);
g.addVertex(v2);
g.addEdge(e);
```

Usage of an Algorithm

The use of an algorithms is straighforward. After instantiating of the desired algorithm, the initial graph should be set, if it has not been done at instantiation. Then, the graph should be verified by the verifyAndPrepareAlgo() method to ensure that the graph is in a correct state before running the algorithm. This step has been explicitly extracted from the execute() method to add more flexibility. Thus, a user knowing that the graph used is ready, he can omit the verification to increase the performance. Here is an example of the usage of an algorithm class with a graph g.

```
Algorithm algo = new Dijkstra();
```

```
algo.setGraph(g);
algo.verifyAndPrepareAlgo();
algo.execute();
System.out.println(algo);
```

5 Metrics analysis

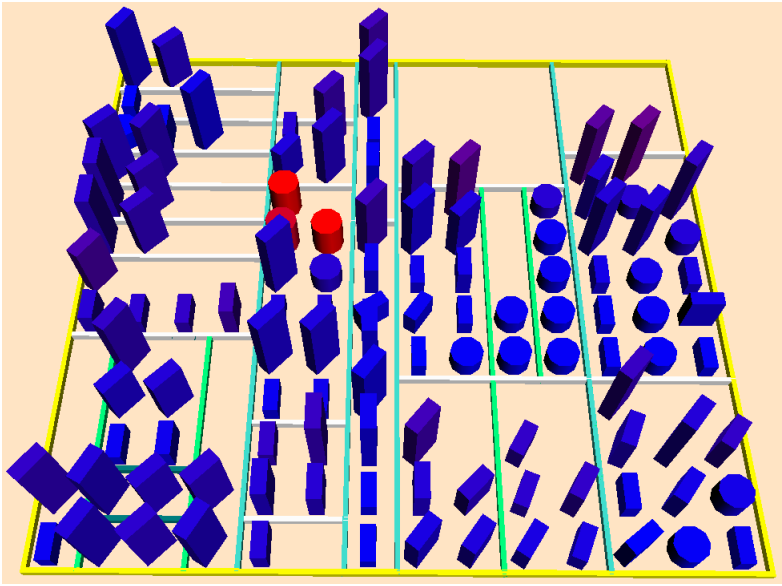
Oraj's metrics were first extracted using POM [10]. Then, the results were visualized using the VERSO metric visualisation tool [11]. Figure 8 illustrates the results obtained. It can be seen in 8(a) that coupling is limited to the 3 main core interfaces IGraph, IVertex and IEdge, which are extensively used by all other classes. In contrast the other classes are relatively weakly coupled. In 8(b), we can see that core packages, in bright red, are used by all other packages, but other modules are independent from one another. Thus, modifications of the oraj.graph.core package would have numerous repercutions on the entire software, but changes in specific modules should be relatively transparent.

6 Operations Research algorithms

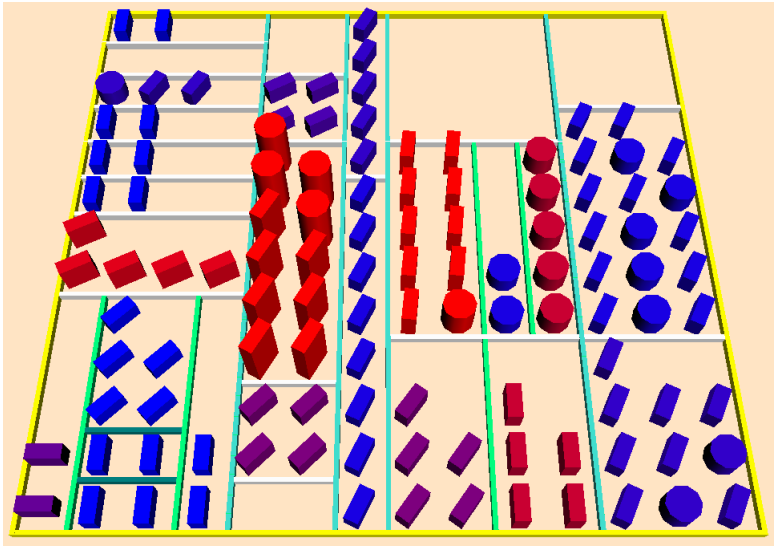
Eight algorithms for different operations research problems are implemented so far. The problems solved by the implemented algorithms are the shortest path from a root node to a destination node, the minimal spanning tree, and the travelling salesman problem (heuristic approaches). These problems can be summerized as stated below:

- **Shortest Path Problem**: The goal is to find the shortest path, if any, from a root vertex to a destination vertex. Depending on the implementing algorithm, negative cost edges might or might not be accepted in the graph used by the algorithm. The implementation is such that an Algorithm instance resolving a shortest path problem in ORAJ can take either directed or undirected graphs. The implemented algorithms are:

  - *Dijkstra*: This is the classic shortest path problem algorithm. This algorithm does not support negative cost edges. Running the Dijkstra algorithm in ORAJ on a graph with a chosen root vertex



(a) ORAJ' coupling blue to red



(b) ORAJ's package dependencies blue to red

Figure 8: VERSO's output of two different metrics mapping. First, in 8(a), the coupling of the classes are map from blue (weakly coupled) to red (strongly coupled). Then, in 8(b), the package dependencies were mapped from blue to red.

will find the shortest path from that root vertex to all other vertices in the graph. If no such path exists, the cost will be set to positive infinity.

  - *Bellman-Ford*: Bellman-Ford algorithm improves the Dijkstra by accepting negative cost edges, as long as no negative cost cycles are found in the graph. A negative cost cycle is a cycle such that the sum of the costs of its edges is smaller than zero.
  - *Floyd-Warshall*: Floyd-Warshall [8] computes the shortest path from a root vertex to all the other vertices in the graph but does this calculations for all vertices in the graph as the root node. This algorithm can take negative cost edges as long as no negative cost cycles exists and is more optimized than the Dijkstra or Bellman algorithms.

- **Minimal spanning tree**: A minimal spanning tree is the set of undirected edges that connects all vertices, if possible, with the minimal cost sum. In wikipedia's words [5] "A minimum spanning tree is in fact the minimum-cost subgraph connecting all vertices, since subgraphs containing cycles necessarily have more total weight.". The only implemented algorithm is:

  - *Prim*: The Prim algorithm elegantly solves the minimal spanning tree problem with no restriction on the costs of the edges. The input graph has to be an undirected graph because this problem is only defined on such graphs.

- **Travelling salesman problem**: The goal is to find the lowest cost cycle path that includes all vertices of the graph. The complexity of this problem rapidely increases defined only upon *complete graphs*. This problem which may look simple at first sight may with the vertex number. Solving this problem exactly implies a computationnal complexity of $\|V\|!$, where $\|V\|$ is the number of vertices. Since this is impractical for most interesting problems, heuristic approaches are used instead of exact methods. In the field of operations research, a solution to the travelling salesman problem (which may not be optimal) is often called a tour. Thus, different tour generation heuristics may be used. The first type is called tour construction and builds a tour following certain heuristic rules, but does not use any other information on the graph

except a root vertex. A second type of heuristic is the tour amelioration heuristic. This type of heuristic takes a non-optimal solution and try the improve it. More information on the travelling salesman problem and on the implemented heuristics may be found in the litterature [7]. The implemented algorithms are:

- *Nearest neighbor*: This tour construction heuristic starts with a root node. Then, it adds to the current initial path the edge going to the closest node from this newly added vertex and this closest vertex is added automatically to the current path. This step is repeated until all vertices are included in the path. To finalize the path and create a tour, the edge going from the last vertex added to the root vertex is automatically added to the path to create the tour. This heuristic is repeated for all vertex in the graph as root vertex and returns the best solution found.
- *Clark and Wright's savings*: The Clark and Wright's savings heuristic begins by creating $\|V\|$ times, where $\|V\|$ is the number of vertices, two nodes subtours that all originates from a root vertex, or depot vertex and contains one of the other vertices (for all vertices). Then, savings are computed. A saving's value is computed as the sum of the edge's ID + DJ - IJ costs where D is the depot node, I is a vertex at the start/end of a subtour A and J is the vertex at the start/end of a subtour B (where A $\neq$ B). A saving is chosen such that the fusion of the two subtours gives a minimal saving value. Then, the two subtours included in that saving are melted into one subtour. Then, the next lowest valid saving is taken and two subtours are again put into one. This is done until all subtours are put together into a single subtour, or tour. Again, this heuristic is automatically ran on all vertices as the depot node and the best solution is kept and returned.
- *Nearest insertion*: The nearest insertion heuristic begins by creating a subtour of only two vertices, the depot and the closest vertex to this depot. Then, while not all the vertices are in the subtour, the closest vertex to the subtour is inserted in the subtour in the position that minimizes the size of the subtour. This heuristic is ran automatically with all vertices and the best solution is kept and returned.

– *Farthest insertion*: The farthest insertion heuristic is very similar to the nearest insertion heuristic. The only difference lies in the selection of next the vertex, in the selection step. Instead of adding the closest vertex to the subtour, the farthest vertex from the subtour is selected as the next vertex to insert. It is important to note that the insertion step remains the same as in the nearest insertion heuristic: This selected vertex is inserted in the position that minimizes the subtour total cost. This heuristic is automatically used on all vertices as depot node and the best subtour is returned.

7 Conclusion

The contribution of the author to this project as been rich and varied in his training in computer science, notably in software engeneering and operations research. The author was able to learn the usage of the development platform Eclipse and the use of the concurrent version system (CVS). It was the first time that the author had the chance to contribute to an already existing open source project. He also had the chance to push the ORAJ project onto the world of open source software by posting the ORAJ project on the renouned open soure software repository, sourceforge. He had the chance to design and implement 117 classes using design patterns to improve the software quality. In operations research, the author was able to achieve new knowledge in the traveling salesman problem heuristics by sucessfully implmenting four different tour construction tsp heuristics. Finally, this project has not only made the author improves in his coding technique but also in his project management abilities, thanks to his two project co-supervisors Dr. Yann-Gaël Guéhéneuc and Dr. Jean-Yves Potvin.

8 Acknowledgments

This project's achievement was greatly supported by the members of the GEODES lab of the university of Montreal and by the two co-supervisor of the project, Dr. Yann-Gaël Géhéneuc and Dr. Jean-Yves Potvin. This project was supported financially by NSERC undergraduate student research award.

References

[1] http://sourceforge.net/projects/oraj.

[2] http://en.wikipedia.org/wiki/Dijkstra_algorithm.

[3] http://en.wikipedia.org/wiki/Bellman-Ford_algorithm.

[4] http://en.wikipedia.org/wiki/Floyd%27s_algorithm.

[5] http://en.wikipedia.org/wiki/Minimal_spanning_tree.

[6] http://en.wikipedia.org/wiki/Prim-Jarnik_algorithm.

[7] L. Bodin, B. Golden, A. Assad, and M. Ball. Routing and scheduling of vehicules and crews. volume 10, pages 63–211, 1983.

[8] T. H. Cormen, C. E. Leiserson, and R. Rivest. *Introduction to Algorithms*. 1990.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Professional Computing Series. 1995.

[10] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the
UML cake. In D. C. Schmidt, editor, *proceedings of the* 19^{th} *conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, October 2004.

[11] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *IEEE/ACM International Conference on Automated Software Engineering 2005*, pages 214–223, Nov. 2005.