Using a Tool Case `decomposeAlgo` for Designing a Reusable Algorithms.

by Kozyr Dmitri.

1 Introduction

In this document we will try to present you how to use the package `decomposeAlgo.zip` (further package or tool case) in order to design your algorithms using reusable operators of this package. The package `decomposeAlgo.zip` is distributed under BSD licence.

We will also explain very summarily how can you make your own operators. If you want some more explanations you can read our university thesis[1] (in French).

2 Settings and Executing Some Examples

The package contains some examples that you can execute and then analyze how they are designed. But before we can start executing them, you need to download some libraries. Please go to http://jung.sourceforge.net/index.html, then to *Download* section and download JUNG library (at least 1.7.2 version) and all third party libraries requested by the latter.

Uncompress `decomposeAlgo.zip` where you want it to be. It will create a folder `decomposeAlgo/` containing 2 folders (`include/` and `src/`) and an ANT `build.xml` file. Put all downloaded libraries (`.jar` files) inside `decomposeAlgo/include/`.

From here you have 2 possibilities : either you execute directly `build.xml` file using ANT (http://ant.apache.org/) or you can build an Eclipse project (or similar).

2.1 Executing ANT

You need to have ANT installed on your computer, you can find it on this page: http://ant.apache.org/.

Run terminal and place your self inside `decomposeAlgo/`, execute:

```
$> ant
```

All files will be compiled and then you can execute 5 examples we've designed by executing :

```
$> ant run1
$> ant run2
...
$> ant run5
```

- `run1`: a simple example with 2 operators. It finds the neighborhood of a path.
- `run2`: a Steiner Problem
- `run3`: articulation Points of a Graph
- `run4`: 2-Approximation Euclidean of TSP
- `run5`: a simple example that illustrates how to transform control flow statements (conditional `if`, loops like `while`, etc.) in operators.

1 "Outils d'Analyse de Réseaux Biochimiques: Décomposition et Assemblage d'Algorithmes Réutilisables" by Kozyr Dmitri. Université catholique de Louvain, June 2006.

If you want to see the design of those examples you can look at their source files in `decomposeAlgo/src/`

- `run1` at `examples/PathNeighborhood.java`
- `run2` at `haut_niveau/SteinerApprox.java`
- `run3` at `haut_niveau/ArticulationPoints.java`
- `run4` at `haut_niveau/GeomTSPApprox.java`
- `run5` at `examples/TestIfOp.java`

2.2 Executing with Eclipse

Open your Eclipse IDE, then go to *File>New>Project* choose *Java Project from Existing Ant Buildfile*. Click *Next*. Then click to *Browse* button and select `decomposeAlgo/build.xml` file. Click *Finish*. This procedure will create a new Eclipse Project with all our source code.

To execute examples from Eclipse, you can go to `toolCase` package and then open `Simulation1.java`. Normally, if you execute directly `Simulation1.java`, you will execute the 3rd example, that is, the Articulation Points of a graph. If you want to execute other examples, you need to comment the line with `ex3(cli);` and uncomment another one.

3 Composing your own algorithms

All reusable operators are situated in packages : `bas_niveau`, `haut_niveau`, `functions`, `predicates`.

We will take the example number 1 (path neighborhood) to explain you how to use operators.

Roughly speaking the code for this example can be reduced to:

```
// 1 component - Dijkstra
Graph D = dijkstraPath(G, start, end);

// 2 component - Expander
R = expand(G, D, r);
```

that is, only 2 components. One that calculate the path between any two nodes (parameters `start` and `end`) with simple Dijkstra. And another one that expand this path as deep as you want (parameter `r`).

Let's have a look on `decomposeAlgo/src/examples/PathNeighborhood.java`

The method `execute()` contains two calls we've shown above, and it's in those methods that we make invocations of our reusable operators. We will only see the case with `dijksraPath()`, the other one is similar.

```
/**
 * @param G
 *            a graph in which we want to find the shortest path
 * @param start
 *            the start node in G
 * @param end
 *            the end node in G
 * @return a graph that contains the path between 'start' and 'end' in G
 */
protected Graph dijkstraPath(Graph G, Vertex start, Vertex end) {
      Operator DP = cli.getOpOnRemoteServer(Op_Names.DijkstraPath);

      DP.reinitOp(new Vector());

      Vector pEDP = new Vector();
      pEDP.add(G);
      pEDP.add(start);
      pEDP.add(end);
      return (Graph) DP.executeOp(pEDP);
}
```

The code above correspond to the method `dijkstraPath`, we will explain you how does it work.

- First line of the method fetch the instance of the operator (`getOpOnRemoteServer()`) on a server(s) that manage reusable operators[2].
  - o As you can see we execute the method `getOpOnRemoteSerever()` on the instance `cli` of object `Client`. `cli` has an access to all remote servers that it knows, so if we want to fetch an instance of an operator on a remote server we need a client reference.

    The `main()` method of `Simulation1.java` contains an example of how to initialize a valid client instance:

    ```
    Client cli = Client.getInstance();

    Proxy proxy = Proxy.getInstance();
    proxy.addServer(s);

    Server s = Server.getInstance(cli);
    s.initializeRepositoryForTests();

    cli.addServer("server1", proxy);
    ```
  - o It's important to notice that a server can also be a client. Indeed, many operators are composed of others and one server doesn't obliged to have all instances of them. When a server need an operator that it doesn't have, it will react as a simple client.
- The second line reinitialize this operator (`reinitOp()`), that is, in case were it was used before and some global or other variables are still initialized. In our case, we want that `DijkstraPath` operator becomes as it was at its first (default) initialization. Some operator can be reinitialized with personal parameters, like custom predicate to determine equality, etc.
- All the remaining lines set execution parameters and execute the `DijkstraPath` operator.

2 We will not explain here in details the architecture of our tool case, however if you want to know more, you can always find more information in [1].

- First of all, we initialize a `Vector` that will contains all execution parameters. Attention, the order is important! This is a little drawback of our tool case and it can be improved later.
- We put all required parameters for execution of `DijkstraPath` operator in `pEDP` vector.
- And finally, we launch the execution of the operator (`executeOp()`). The returned result will be always an object, so we need to cast it.

Every operator of our tool case implement an interface `Operator` in order to offer methods for reinitializing and executing itself.

To sum up, when you want to execute an operator you need:

1. Fetch the operator on the server(s).
2. Reinitialize it.
3. Execute it.

Unfortunately, operators are sometime insufficient to design an algorithm and we are obliged to fall back on crude code. This problem could be probably resolved when our tool case will have more operators.

4 Designing Your Own Operators

If you want to design your own operators and integrate them in our tool case or, more generally, if you want to design reusable algorithms, you can follow our small tutorial.

We've developed an approach for constructing reusable operators that consists mainly in 3 points:

1. *When make an operator?*

   This is the most important and complicated process. We can subdivide it in 3 other sub-points:

   i. Analyze a **set** of problems. It's important to insist on “set”, indeed, grater the set will be, grater reuse we will be able to guarantee to our operators. And more our operators are reused, better it will be.

   ii. Find likenesses between them, that is, some operation that are performed in the same way or can be performed in the same way, etc.

   iii. Design an operator that will replace those likenesses.

2. *Apply Template Method Pattern [2].*

   This will give a structure to all our operators and a good internal decomposition. We recommend you to read about this pattern in “Design Patterns : Elements of Reusable Object-Oriented Software” by Gamma & al.

3. *Implement `AbstractOperator` interface.*

   This interface is required by our tool case, because all accesses to the new operator will be made through this interface.

This process can be reapplied from time to time on existing algorithms designed with operators that still contain crude code in order to minimize it.

5 Bibliography

[1] D. Kozyr, “Outils d'Analyse de Réseaux Biochimiques: Décomposition et Assemblage d'Algorithmes Réutilisables”. Université catholique de Louvain, 2006.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides, « Design Patterns : Elements of Reusable Object-Oriented Software ». Addison-Wesley, 1995.