Bipartite graph and traffic grooming on *MASCOPT* library

Paolo Pastorelli

February 1, 2007

Abstract

This is a report on my work done at INRIA from the 1st july 2006 to 17th november 2006.
I am so grateful to professor Jean-Claude Bermond, for this opportunity, and to professor Michel Syska and the ing. Fabrice Peix who followed my work and helped me every time I needed him.

Contents

1 Introduction

The matter of my five months of work at INRIA is: *Mascopt*.
The main objective of *Mascopt (Mascotte Optimization) project* is to provide a set of tools for network optimization problems. Examples of problems are routing, grooming, survivability, or virtual network design.
Mascopt will help implementing a solution to such problems by providing a data model of the network and the demands, libraries to handle networks and graphs, and ready to use implementation of existing algorithms or linear programs (e.g integral multicommodity flow).
Mascopt is Open Source (under LGPL) and intends to use the most standard technologies as Java Sun and XML format providing portability facilities [1].

2 About data structures on *mascoptLib*

In this section I itemize, and try to present very fast, the basic *mascoptLib* objects that I used to work on the graph during my project. This basic objects are the same objects (with the same name) that we meet on graph theory definitions (very helpful for a very fast comprehension of the library):

- *vertex*: the basic entity on a graph. The vertex, defined with a pair of coordinates or not.
- *vertex-set*: a set of vertices, necessary to define an edges or arcs set.
- *edge*: defined on a vertices pair. / *arc*: defined on a vertices pair, the order here is important: the first vertex is the tail and the second the head.
- *edge-set*: a set of edges defined on a vertex-set, necessary to define a graph. / *arc-set*: a set of arcs defined on a vertex-set, necessary to define a di-graph.
- *graph*: an undirected graph defined on an edge-set. / *di-graph*: a directed graph defined on an arc-set.
- *map*: the not immediately comprehension object. A map is the way adopted on *Mascopt* to associate a value at a *Mascopt* object.
  Like we can see later this is very useful to associate a capacity to an arc when an arc represent a pipe. The values that we can associate to a *Mascopt* object are more than one because to recover a value is necessary: the object, a context (a parameter with type *Mascopt* object)

and a name a parameter with type String.

$(object, context, name) \Rightarrow value$

The importance of this structure is clear with this little example: a same arc (or an edge, or a vertex...) can belong to two (or more) different graphs, is sufficient to use for context a graph or another to associate different values to the same arc.

3 First steps

3.1 General description

For take confidence with the library and move my first steps in it I started with the implementation of two simple algorithms: a *bipartite graphs generator* and an algorithm to solve *matching problems* on this type of graphs.
The motivation for that was the study of a multi client/server model: we want to assign (optionally) client to servers depending on topological constraints.

3.2 Bipartite Graph Generator

3.2.1 Theoretical study of the problem

In the mathematical field of graph theory, a *bipartite graph* is a special graph where the set of vertices can be divided into two disjoint sets U and V such that every link has one end-point in U and one end-point in V [2].

3.2.2 Implementation details

My class *BipartiteGraphGenerator* has two contructors:

- one to generate a graph where the user can specify the cardinality of the two separated vertex set.
- one to generate a graph where the user can submit two pre-existant vertex set.

Anyway for the two constructors are necessary two other parameters: the *average* and the *variance.*
The number n of outgoing edges (or arcs) from a vertex on the first set, that link this vertex with the same number n (*no multigraph allowed but only simple graphs*) of vertex on the second set, is determined through a

gaussian-distribution with this *average* and this *variance*.

```
public class BipartiteGraphGenerator
{

...[cut]....

public BipartiteGraphGenerator(int cardA, int cardB, int avg, double dev) {
...[cut]...}


public BipartiteGraphGenerator(MascoptVertexSet A, MascoptVertexSet B,
int avg, double dev) {
...[cut]...}

...[cut]...

}
```

After the creation of the object, the user can call the two methods:

```
public class BipartiteGraphGenerator
{

...[cut]....

public MascoptDiGraph getNextDirectedBipartite() {
...[cut]...}


public MascoptGraph getNextUndirectedBipartite() {
...[cut]...}

...[cut]...

}
```

with this two methods we can obtain a directed (or undirected) graph. In fact, a new graph is created (with a new link set) for each call to methods *getNextDirectedBipartite()* and *getNextUndirectedBipartite()*.

3.3 Bipartite Matching

3.3.1 Theoretical study of the problem

The *bipartite graphs* are useful for modelling matching problems. For this reason my next step was, naturally, an algorithm for *bipartite matching.* The problem is also known as *maximum bipartite matching*: a perfect matching between vertices of a bipartite graph, that is, a subgraph which pairs every vertex with exactly one other vertex.
We can observe that a *matching problems* in a *bipartite graph* having a reference to *flow problems.* Notably in this context we can see the *matching problem* like a *max flow problem*: find a max, directed, flow on the bipartite graph with a source before a first vertex set and a destination after a second vertex set. In the *Fig.1* we can see this representation: on a bipartite graph we choose a direction for the flow and,coherently with this choice, we add an over-source and under-destination. Only the flow is oriented, not the matching, because for a matching problem the pair (a, b) (where $a \in vertexsetA$ and $b \in vertexsetB$) is equal to the pair (b, a).

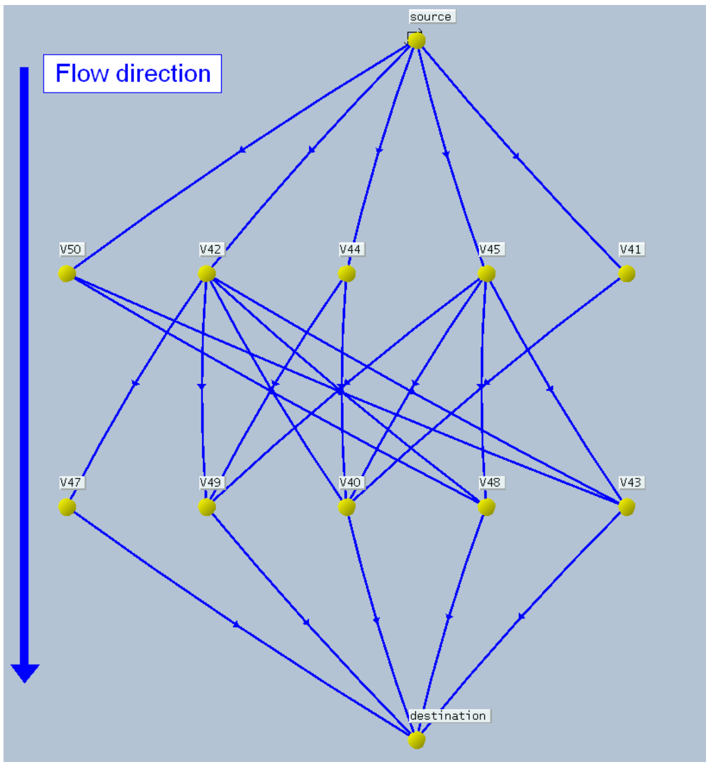

Figure 1: Reference between *bipartite matching problem* and *max flow problem.*

3.3.2 Implementation details

After this theoretical consideration the implementation of algorithm is quite simple. In fact the algorithm of max flow is already implemented in *Mascopt* following the *Edmonds and Karp* algorithm. It was only necessary:

① prepare the graph like you can see in *Fig.1*

② calculate the *max flow* with a unitary capacity on arcs

③ delete the arcs without flow (between the two sets of vertex in the *bipartite graph*) and all the arcs linking source or destination with the vertex of *bipartite graph*

4 Grooming

4.1 General description

The traffic grooming is the process of taking telecommunications traffic and sorting it into the most efficient arrangement possible. This process includes considering the topology of the network and the different routes in use.

The exponentially increasing traffic demands require higher speed transmission and in many years the *optical technology* dominates the *backbone networks.*

A model to explain these optical networks is the *layered wavelength-division multiplexing (WDM) network model*: it encapsulates the signal in three different layers: *wavelength / band / fiber (WBF).* With this network transport model the nodes must be equipped with specific devices to switch high-speed optical signals: *the optical cross-connector (OXC).* We can enumerate three type of OXC, like the three layers in the model: F-OXC, B-OXC, W-OXC. The difference, clearly, is in the capacity that they have to switch signal at fiber level, band level, wavelength level.

Since a little while the high speed that we can obtain with optical network affected also the *access networks* and originated the *optical access network (OAN).* Now, therefore, we can see that the optical networks, link not only the big access points in the backbone networks, but also they connect end-users for specific requests.

After these considerations we can already identify some reasons that can explain why the research in matter of *grooming* algorithms is so active and important:

- the high cost to deploy new links between nodes in optical networks

- the different cost to equip with a different OXC type a node (a W-OXC is clearly more expensive that a F-OXC)
- the fact that in a OAN the setting up cost must remain low because it is shared by a little set of end-users
- the fact that usually the network capacity is never full
- the necessity to minimize the number of multiplex and demultiplex operations (that usually is the "bottle neck" in the network)

In short, the *grooming* problem is a "bottle neck" problem with the necessity to minimize the cost, time...to serve a generic request in a generic network. In general, we can apply this theory to solve, with optimal solutions, all problems that have a reference to graph mathematical model: for example move people in a transport network formed of different means of transporting.
In this example the theory is explained clearly: when a person must travel from a place to another he use (and, very important, combines) different means of transoporting according the travel that he has to do (car for short displacement, plane for long. . .). Like this a wave lenght according its request path it will be encapsulated in different bands or fibers during its path between source and destination.

4.2 Theoretical study of the problem

For these reasons, in my stage I began a study of grooming problems and tried to implement some of this algorithm in *Mascopt*. For explain clearly the problem it is necessary, before, understand the instruments that I had already at my disposal and agree on some terms that I will use.

I speak about *two layers grooming algorithms* because this is a general case that we can repeat, two times likewise, for grooming the wavelengths on bands and, then, the bands on fibres. The hypothesis under them I worked is the same that we have on the project "RNTR PORTO" [3]:

- input data are a directed graph that represent the network and another directed graph that represent the requests with one arc between source and destination.
- the network graph is a graph of type FBW (Fibers Bands Wavelenght) and the capacity of any arc (fiber's number) is given.

- a request correspond to an integer wavelenght number (or request's tail) that we must reserve from source to destination. A single wavelenth on the request is called *elementary request*.

Before the *grooming* phase, a *routing* phase is necessary to obtain flow associated to every requests. We adopt this two phase algorithm, but the general problem is more complex: a mix between *routing* and *grooming*. To do the *routing* phase we use an existing algorithm disponible on *Mascopt*: *LPMultiFlowMonoRouting*. With this algorithm all the flow of requests follow one path on the network graph. After, it is simple to recognize on network the links where a flow of a certain request pass through and, with these arcs, reconstruct the path associated to the request. You must note that this choice (*mono routing*) is arbitrary and you can use other *MultiFlowRouting* algorithm to obtain a set of path instead of one path.

After this phase we can start the *grooming* phase: the goal of my study is implements different grooming heuristics and, then, do a comparison with test on different network graph with different request sets. In conclusion, extract some considerations from it.

To do this comparison it is necessary set one or more parameters. One of these parameters can be the number of *pipes* that an heuristic make on a network graph with a fixed request set and fixet *routing* of these requests. The wavelenghts *grooming* on the bands (or the bands *grooming* on fibers, remember, we speak about a general two layers case) bring to build bands (fibers) pipes. We can consider an heuristic better than another if it builds a smaller number of pipes.

But, what is a pipe?
One band (fiber) can be considered a pipe with length one. But, also, if we have a pipe P_{ij} on arc $i \rightarrow j$, and another pipe P_{jk} on arc $j \rightarrow k$, we can "stick" P_{ij} with P_{jk} to create a pipe with length two, if, and only if, all *elementary requests* that P_{ij} contains are also contained by P_{jk} *(fig 2)*.

4.3 Heuristics description

Done a *mono routing* of request set on a network, we can follow more strategy to do a grooming. The order of pipes reservation is from the source to destination for the request that we consider at moment, at every arc, we try to reserve: a free pipe, if it is possible, the smaller number of pipes, otherwise. The differences between the heuristics that I have studied are in the order that we use to treat the requests:

- Greedy - Random order.

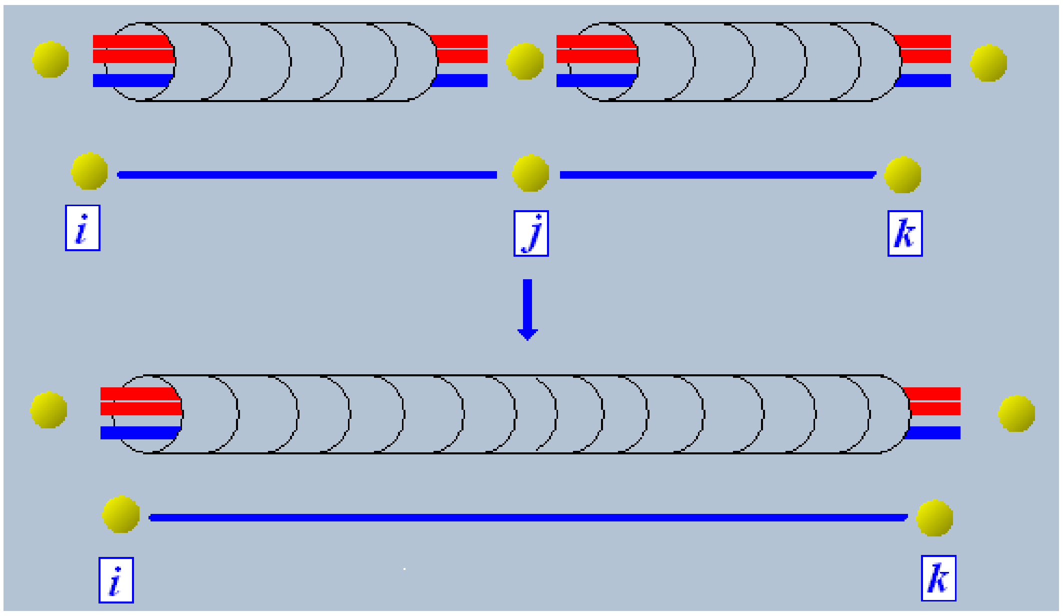

Figure 2: We can "stick" two pipes P_{ij} and P_{jk} if and only if the two pipes contains the same elementary requests. The length of the new created pipe is equal to the sum of the length of the two original pipes.

- Biggest First - First we treat the request with the biggest size.
- Longest First - First we treat the request with the longest routing path.
- Biggest Longest First - A combination of the two previous heuristics.
  - ① We ascribe a weight at a request: the length of routing path multiplied with her size.
  - ② We treat the requests in decreasing weight order.
- By Start Node.
  - ① We ascribe a weight at a request: the number of requests that begin from the its own source vertex.
  - ② We treat the request in decreasing weight order. Between requests with the same weight (clearly, requests with the same source vertex) we treat in random order.
- By End Node.
  - ① We ascribe a weight at a request: the number of requests that end in the its own destination vertex.

② We treat the request in decreasing weight order. Between requests with the same weight (clearly, requests with the same destination vertex) we treat in random order.

Another speech is necessary, for me, for the grooming heuristic:

- With Ceilling Factor - The novelty introduced by this heuristic is in the pipes reservation side: after a pipe reservation for a done request if the busy capacity of this pipe is over a fixed per cent rate, the utilization of this pipe is blocked for the following requests. With this "pipes reservation policy" we can combine any order to treat the requests (BiggestFirst, LongestFirst, BiggestLongestFirst, ByStartNode, ByEndNode) and we obtain five new, different, requests.
  This is only a presentation: more details and clarifications are in the next section of this report.

4.4 Implementation details

4.4.1 HeuristiquesTest

During my work I produced some classes. Before to begin to explain the implementation of every single class it is necessary, for me, explain clearly and definitively the order that I followed to construct my *HeuristiquesTest* class: the class that constitute the link between all other classes.
My *HeuristiquesTest* class:

① Ask to user (graphic interface) the **.mgl* file that contains the Cable Graph, the Request Graph and a *Map* (see *About data structures on mascoptLib*) with data about Pipes Capacity and Requests Size. The same *MascoptObject* can be associated at more value because the key for recover a value is a "triple key" composed by the *MascoptObject* a field called *Name* (of type String) and a field called *context* (another *MascoptObject*).

② Routing phase with the class *MonoRoutage*.
Set the *parameters* and call the method *execute*.

③ Prepare the requests for grooming. Its implementation pass from a *MascoptArc* (with tail: source vertex and head: destination vertex) to a *MascoptDiPath*.

④ The requests are collected on a *MascoptSet<MascoptDiPath>* to be passed at grooming class.

⑤ The network graph are prepared for grooming: the total capacity between two vertices (now concentrated on one arc) are distributed on the new arcs to implement the different bands or wavelenghts on the network.

⑥ Grooming phase with the different grooming heuristics: set parameters and call execute method.

⑦ Calculate the pipes number with the class *Tubes*.

The last two points of the HeuristcsTest algorithm are repeated (for every heuristic) an arbitrary number of times. This to warrant (at every execution of grooming algorithm) a different mixing for requests before to put its in the right order. The number of pipes, ascribed at an heuristic, is computed like an average between all the different executions.

4.4.2 GrilleGenerator *and* GrilleToriqueGenerator

This class generates a graph with *mesh* or *toroidal mesh* topology and fix a requests set on it. The user must only set:

- Number of rows and columns for the mesh graph.
- Average of the number of requests that beginning in every vertex.
- Respective standard deviation.
- Capacity of ONE pipes between two vertices.

The size of every requests is fixed to 1.

4.4.3 MonoRoutage

The class *MonoRoutage* use the class *LpMultiFlowMonoRouting* and a solver for linear program (Cplex [4] in this case). It is sufficient to set the parameters:

- network graph,
- request graph,
- requests size,
- pipes capacity,

and, then, compute the multi-flow problem.

4.4.4 Grooming

The test with the class *HeuristiquesTest* have been effectued with an old version of grooming algorithms (present in the repository). The differences with the new version that I will go to describe are only at code/design level. In fact at first I have started to code every heuristic in a different class but, after a little study, we can see that: the different grooming heuristics that I have presented in the previous chapter can be obtained with a simple combination between:

① an order to treat the requests

② a policy to reserve the pipes for a request

For this reason, at code level, I have coded only one class *NewGroupage* that need two interfaces to be defined. An interface to sort the requests *OrderRequests* and another to reserve the pipes *PipesReservation*. At the moment the coded classes that implement the *OrderRequests* interface are 6:

- *OrderRequestsRandom*
- *OrderRequestsBiggestFirst*
- *OrderRequestsLongestFirst*
- *OrderRequestsBiggestLongestFirst*
- *OrderRequestsByStartNode*
- *OrderRequestsByEndNode*

And 2 are the classes that implement the *PipesReservation* interface:

- *PipesReservationDefault* - That provide a pipes reservation with default policy described in the previous chapter: free pipe between two vertex if it is possible, the smallest number of pipes otherwise.
- *PipesReservationCeillingFactor* - That provide the *ceilling factor* policy described in the previous chapter: if the capacity busy in a pipe (after a reservation for a request) arrives at a fixed percentage this pipe is declared full.

With this strategy (two interfaces) we can see that it is very easy obtain the different heuristics described in the previous chapter and it is possible to obtain more heuristics for example with the combination of different "‘order requests"’ policy with "‘ceilling factor reservation pipes"’ policy.

The output data are stored in two HashTable returned by this two methods:

- *getTabPathRequestToPathPipesThatBuildIt*
  The HashTable returned by this method has type:
  MascoptDiPath -> Set(Pair(MascoptDiPath,MascoptAbstractScalar))
  This HashTable in fact relate a MascoptDiPath that form a request to a set of MascoptDiPath formed by pipes that form it. There are the different path of pipes found after the grooming and everyone have the associated capacity of request served.
  The MascoptDiPath that are key in this HashTable are the result of Routing. In the *MonoRouted* case is simple to take back the *MascoptDiPath* associated with a request, but in the *MultiRouted* case this operation need more time because we must find all the *MascoptDiPath* associated with a single request.

- *getTabPipesToRequestInside*
  The HashTable returned by this method has type:
  MascoptArc -> Set(Pair(MascoptDiPath,MascoptAbstractScalar))
  This HashTable relate a pipe of physical graph (a MascoptArc) with the set of MascoptDiPath that pass in it and relative capacity served. In this table the MascoptDiPath considered are the path that constitute a request (for understand: the MascoptDiPath there are keys of the previous table).

4.4.5 Tubes

This class provide the computation of *pipes* created in the network graph after the *grooming* phase. The class necessitates only the network graph and the hashtable with the grooming correspondences between requests and pipes of the graph.
The class takes a vertex and begin a test: if some incoming pipe have the same requests (with same busy size) of an outgoing pipe we can link the two pipes. The resulting pipe has a length equal to the sum of the length of the two "original" pipes. The resulting graph is the graph of pipes.

5 Conclusions and future works

I did some test to search the best choice (about requests order and pipes reservation) for a grooming algorithm. For my testing phase the best choice

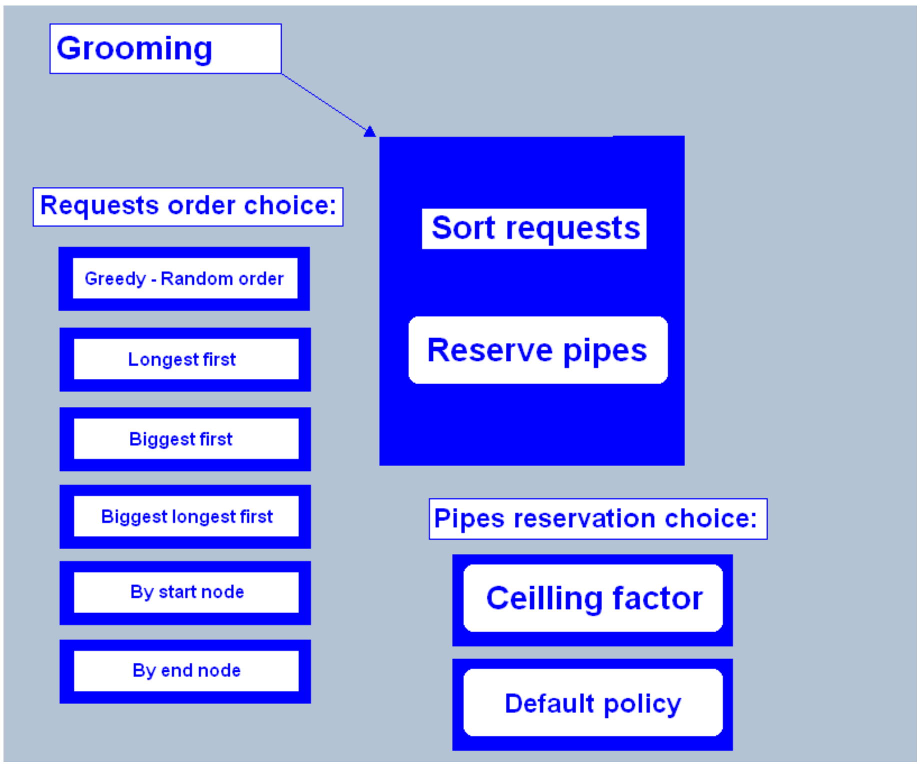

Figure 3: Grooming composition

are a combination *OrderRequestsBiggestLongestFirst* with *PipesReservation-CeillingFactor* The test algorithm involves a lot of parameters that I have set arbitrarily and, for this reason, the choice can't be defined the best. This parameters are: the size of the graph (*Grille* or *GrilleTorique*), the number of requests...
For the future I think that the most important points to develop are:

- an improvement on the test algorithm to identify (and strong justify) a better choice on the already disponible heuristics,
- the implementation of most complicated grooming heuristics,
- a combination with *multi routing* for the requests.

References

[1] *http://www-sop.inria.fr/mascotte/mascopt/*

[2] *http://en.wikipedia.org/wiki/Bipartite_graph/*

[3] *http://www-sop.inria.fr/mascotte/porto/*

[4] *http://www.ilog.com/products/cplex/*

[5] Jean-francois Lalande,
Conception de reseaux de telecommunications : optimisation et experimentations.
http://www.inria.fr/rrrt/tu-1139.html

[6] Cormen, Leiserson and Rivest, *Introduction to Algorithms.*

[7] *http://www.math.uni-hamburg.de/home/diestel/books/graph.theory/*