GRPH: an unpronounceable[1] graph Java library focusing on performance

Luc Hogie and friends

I3S(CNRS-UNSA)/INRIA

[1] Still, you may pronounce it *groumph*.

Agenda

Motivations for developing another graph library

Global overview of GRPH

7 bullets to kill performance bottlenecks

Demonstration

Users

Conclusion

Motivations for developing another graph library

In the context of our projects, we need a graph library that is:

- enables the fast (CPU) manipulation of large (RAM) dynamic networks;
- portable;
- intuitive;
- adequate to network simulation;

Several graph toolkits already are available to the “graph” and “networks” communities. Among them:

- Boost exhibits the best performance, but is hard to use (even the simplest examples are scary);
- Jung and JGraphT offer the best portability: but they have poor performance (both computational and memory usage are bad);
- SageMath is the best candidate when it comes to graph experimentation. It is written in Python (with a bridge to C). Not suitable to large software developments.

In spite of the variety of tools, most often people anyway opt for developing their custom code (is re-inventing the wheel good or bad?).

Global overview of GRPH

Briefly, GRPH is a Java graph library augmented with a set of experimentation tools. Its development started in 2008. It focuses on efficiency. It supports the following class of graphs:

- simple graphs;
- multigraphs (two vertices can be connected by several edges);
- hypergraphs (edges may connect more that 2 vertices);
- any undirected, directed or mixed versions of these (a mixed graph consists of edges of various nature);
- dynamic graphs;

Undirected simple edge

- a RJ45 cable between two computers;

Directed simple edge

- a sattelite connection
- a optical fiber connection
- any directional abstract relation: (a knows b, a depends on b, etc)

Undirected simple edge

- a bus connection throught ethernet switch;

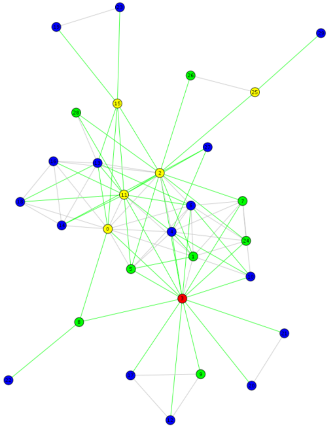
Directed hyper edge

- in a 1-n directed hyper edge, n represents the set of devices in range;
- in another such edge, n represents the set of devices that can entail the hidden node problem;
- in a n-1 directed hyper edge, n represents the set of devices to which a given device is in range;

Data structure

GRPH proposes a data structure.

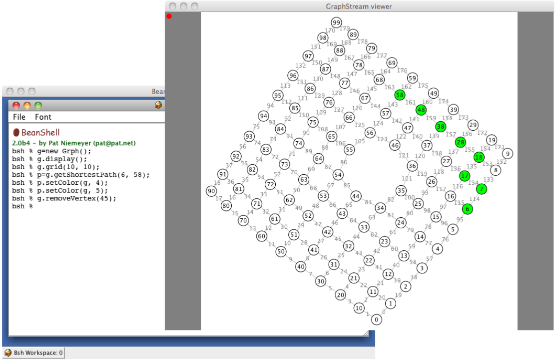
Dynamic graphical monitoring DEMO



Grph comes with a bridge to GraphStream which endorses it with automatic graph layout and customizable dynamic graph representation.

Interactive console DEMO

One of the greatest strength of the SageMath Python toolkit is its interactive Python interpreter. In the same way, GRPH comes with an interactive shell, by relying on BEANSHELL. The interaction language is a dialect of Java.



Discrete-event engine

GRPH comes with a discrete-event simulation engine that allows it to consider dynamic systems.

- mobility models for mobile networks;
- fault models for unreliable networks;

Experimentation framework DEMO

GRPH comes with an experimentation framework that greatly helps the elaboration of usable results by taking care of:

- the non-recomputation of already computed stuff;
- the processing of multi-runs;
- the generation of the plot files.

This greatly simplifies the graph experimentation codes of the user, and scale them down by a large factor.

Framework for evolutionary computing DEMO

GRPH comes with a evolutionary computing framework targeted to the generation of graph instances. This framework, called *Drwin* have the following particularities:

- it does not encode indivdiduals, thereby exploited meaningful application-level operators;
- it is multi-threaded;
- self-adaptive evolution and parallelism;
- OO.

Interoperability DEMO

A GRPH graph can be imported/exported as/to:

- Grph (compact binary GRPH native format);
- GraphText (compct text GRPH native format);
- GraphML (XML-dialect);
- GML;
- DOT/Graphviz (graph plotter);
- DGS (Dynamic Graphs);
- Inet/CAIDA Maps (topology generator);

But also as:

- JUNG graph;
- Mascopt graph;

Algorithms

GRPH comes with:

- basic topology schemes (grid, ring, chain, star, clique, etc), random schemes (GNP, GNM, random tree), GLP (Generalized Linear Preference), etc.
- implementations for common graph algorithms: eccentricity, radius, diameter, in/out vertex/edge degrees, clustering coefficient, density, connected components, minimal spanning tree, shortest paths, BFS/DFS/RS, distributions, maximum clique, minimum vertex cover, maximum independent set, maximum flow, (sub)graph isomorphism, etc.

6 bullets to kill performance bottlenecks

Not fully Object Oriented framework

Bullet: In Java, OO kills performance

- Problem
-
  - Java object memory management is too hungry
  - OO implies an indirection to obtain, anyway, its ID
- Solution: GRPH considers the vertex (or edge) IS its ID.
- Advantages:
-
  - save memory by factor 4
  - allow much faster management (use of HPPC)
  - use of low-level caching (speedup 15)

Fast-access incidence lists

Bullet:Both arrays and hash-table are accessed in constant time, but accessing an array is so much faster!

- The implementation of GRPH uses 2 coupled incidence lists.

Representing vertex/edge sets as daptative integer sets

Graph algorithms spend a great deal of their time manipulating vertices and/or edge sets. GRPH proposes three implementations for integer sets:

- based on hash tables, adequate when the set is sparse;
- based on bit sets, adequate when the set is dense;
- adaptative, adequate when the density evolves unpredicably.

Bitset-based intsets require at least 32 times less memory than hash-tables-based ones. When the density cross the threshold 1/32, the implementation of the set is switched. In order to avoid too frequent implementation switches, GRPH uses an hysteresis mechanism.

Caching already computed stuff

Bullet 2: does not compute twice the same thing! GRPH makes use of a *cache*.

- Basically, any given property will not be computed twice on the same graph; this *breaks* the complexity of graph operations.
- For example, the computation of the diameter will return immediately if all-pair shortest paths were computed previously. This is because diameter requires the distance matrix that was already computed by the shortest path algorithm.

Depending on the application, performance dramatically improves!

Use of C/C++ code

Bullet 4: if you cannot fasten your Java implementation anymore, then write it in C.

- hackers can expect a speed-up of 5;
- GRPH will find it and compile it on-the-fly using optimization flags for your specific computer;
- if compilation failed, an optional 100% pure Java alternative will allow the program to run, still.

JNI and JNA prove inadequate. Instead Grph resort to process piping. (number of triangles, max-clique, (sub)graph isomorphism, etc) are already implemented in C++.

Parallelizing algorithms

Bullet 5: take advantage of multi-core computers:

- algorithms that can be computed independently on every vertex in the graph can be automatically parallelized;
- GRPH generates a lot more threads that installed cores, hence reducing the probability of unfair load balancing;
- on my dual-core computer, I experimented speed-up between 1 and 1.7.

Resorting to linear programming

Bullet 6: benefiting from the advantages of linear programming:
Many graph problems can be expressed as linear programs:

- the linear program is often shorter than its corresponding algorithm;
- its resolution benefits from the high efficiency of the solver's strategies for solving.

Supported for CPLEX and GLPK (under progress). Invocation of remote solvers (through SSH) is also available.

Disabling verifications

Bullet 7: a good program is one that checks its parameters. But if you KNOW that the parameters are correct, you can disable these time-consuming verifications.

- method arguments are checked by assertions;
- hence they can be disabled;
- improves "production" mode.

Demonstration: let's see how to do things DEMO

- creating a graph (10×10 grid);
- computing the diameter;
- computing all pair shortest paths;
- adding a new Java algorithm;
- adding a new property to vertices;
- computing a distribution;
- console profiling;

Users

- 31 users;
- INRIA, INRA, UCL, Trier, Georgia, Politechnika Warszawska, etc

Almost the end

Conclusion

So we have a graph lib which design objectives are to maximize:

- memory usage (use of native types);
- computational efficiency (use of caching, parallelism, etc);
- unpronounceability.
- simplicity to use (simple but functional Java API and tools);

GRPH is currently used by:

- Mascotte team at INRIA (at the heart of DRMSim);
- You? (no worries, we do provide pro-active support).

Conclusion (why would you do so?)

I suggest you to take a look at GRPH if you want to:

- to write graph-based scientific applications;
- get rid of Java usual inefficiency;
- work with me. :)

http://www-sop.inria.fr/members/Luc.Hogie/grph/

End of the presntation.

Any qustions?