Glay

Graph Visualization and Analyzation System

Glay: Graph Visualization and Analyzation System
Published 2004 June 30.

Table of Contents

List of Tables

Chapter 1. Introduction

This is Glay, a system for handling, processing and visualizing graphs written in Perl and modern C++. It aims to follow the KISS (Keep It Stupid and Simple) design principles, and concentrates on implementing reusable pieces of actions. Hence people knowing the Perl programming language are able to exploit it in is entirety. Using only the built-in modules like layout algorithms or the Visualization module however does not require any programming or Perl whatsoever.

Glay is built up of two main modules. One of them --- the "Visualization module" --- eats vertex coordinates, connectivity data and style information, and produces nice 3D pictures of the graph. It just draws; it has no brains or any means at all to figure out where to put the vertices, or what color the edges should have. It is usable on its own: if you provide the simple text format that it recognizes you are able to visualize graphs.

The "main module" of glay is the place that the graph can be read into, and manipulated with. It is basically a collection of actions, like an action for inputting different formats, an action for coloring, etc. Most important is the easiness of creating user defined action-files. It means that if you would like to experience with the graph, recolor some parts, calculate some quantities, etc, you only have to code the very action you want, in a minimalistic format.

The "Visualization module" and the "Main module" are designated as independent software and you are encouraged to consider using them separately, if either one alone fits your needs. If you just need to manipulate a graph, probably a command "glay -i filename -a actionname" is a command for you. If you have a separate program that can produce appropriate input for the "Visualization module", you can use that alone to visualize your data. Feel free to experience, and good luck!

Chapter 2. The basics

Please be understanding: either I assume some prerequisites, or the introduction grows huge and boring. Here they are:

- I assume that Glay and all the packages it depends on are installed. See the section on Installation for information.
- I hope that you are using a unix-like machine, like GNU/Linux, Solaris, etc. Windows users please see the section on windows.
- I usually tell you what to type to the "command prompt" called "shell" in the Unix world. I will use the "bash" syntax for commands. To see what sort of shell do you currently use, issue the command

  ```
  ps -p $$
  ```

  and see the last word. If it is `bash`, `zsh`, `sh` or `ksh` that means you have a bash-compatible or "Korn" shell. If it is `csh` or `tcsh` than you have a "C" shell. In this latter case the commands may not work you verbatim. Please see the section on the C-like shells for information on the most important differences.
- I assume that the Glay binaries are in your `PATH`. To check this, try

  ```
   glay -a hello
  ```

  If you just want to use the Visualization module, you don’t even need this, just try the command

  ```
  cichlid_pipe -v
  ```

  and see if it greets you. Please refer to the section on Installation if these commands do not work.

2.1. How to use it just for visualization

2.1.1. Calling Glay

You need to tell Glay what to do by giving it arguments, for example:

See if glay is really able to read the file:

```
glay -i filename -d- 2>&1 | less
```

View a file with coordinate information present in it:

```
glay -i filename -a visual
```

The same, but specifying that the format of tha file is "lazy":

```
glay -i filename lazy -a visual
```

To run the graph layout algorithm called "J", then dump the result into layed.glay, and the results with a separate command:

```
glay -i filename -a layout J -d layed.glay
glay -i layed.glay -a visual
```

The same, but in one command. The `-p` parameter here is for reading a later discussed parameter file to adjust the behaviour of the layout algorithm:

```
glay -i filename -p J.mypars -a layout J -d layed.glay -a visual
```

The arguments are for telling what to do, so --- unlike is most unix programs --- the order of them is in fact important. There are four main parameters: `-i` stands for *`input`*, `-a` is *`action`*, `-d` is *`dump`*, `-p` means *`parameter`*. The usage of them is described in Table 2-1.

Table 2-1. Syntax and meaning of the parameters

`-a` *`actionname opt_arg1 opt_arg2 ...`*	Run an action. The actions are searched first in the environment variable `GLAYACTIONPATH`, then in the built-in action path. These action directories are searched recursively (which means that one cannot use two different actions of the same name). The actions can have an optional *`.pm`* extension. *`opt_arg`*s are optional arguments of the specific action.
`-i` *`filename optional_format`*	Input a file. It is simply a shortcut for `-a` *`input filename optional_format`*. The valid file formats are discussed later. If *`filename`* is `-`, then the file is read from the standard output. If the format argument is omitted, glay tries to use the extension of the filename as a format name. If the extension is not a valid format, it falls back to "native".
`-d` *`filename optional_format`*	Dump the graph and related data to a file. It is simply a shortcut for `-a` *`dump filename optional_format`*. The valid file formats are discussed later. If *`filename`* is `-`, then the file is dumped to the standard output. If an *`optional_format`* is not given, glay tries to use the extension as a format name. If the extension is not a valid format name, then it falls back to "native".
`-p` *`parameter_file`*	Read a parameter file. Parameter files are used by some specific actions, for exapmle layout algorithms. The file read by the `-p` argument is kept in memory, and used later if a corresponding action requires it. Hence one needs to place for example `-p` *`J.my_params`* ahead of the corresponding `-a` *`layout J`* arguments.

2.1.2. File formats and main properties

In this subsection you can see how to provide some data to Glay. Providing the graph for Glay is not the only way to go: you can also ask it to generate a graph that you need, see the next section for information on that.

If you know the Perl language, than it is extremely simple to implement "importers" for some particular graph format, see how to do that.

Here you can read about the formats recognized by the built-in "importers". Reading this also helps to learn about how Glay stores a graph, and what sort of properties of the vertices and edges are recognized by standard modules, for example the layout modules.

FIXME: megirni a sect_ownimporters-et!

All of the following format names can be used in the optional argumentum *`format_name`* of *`-i`*:

```
glay -i filename format_name ...
```

`gdc` format:

It is a syntax that is also a valid perl program, it looks like this:

```
vtx "V1", "weight=2.7", "size=0.3", "color=green";
vtx "V2", "weight=2.7", "size=0.3", "color=green";
edge "E1", "size=2", "from=V2", "to=V1", "color=yellow";
```

The general syntax of the lines is

```
vtx "label", "tag=value", "tag=value",... ;
edge "label", "tag=value", "tag=value",... ;
```

The format is not line oriented, the semicolons close a vertex or and edge definition (so you can break the lines if they grow too long). Vertex definitions start by *`vtx`*, edge definitions start by *`edge`*.

The tags are arbitrary with one exception: the *`neighbour`* tag for vertices is reserved for recording the actual neighbours of a vertex. Obviously the *`to`* and the *`from`* tags are better provided for the edges, as shown in the example above. Currently glay supports only the handling of undirected simple graphs, so interchanging the values of the *`to`* and *`from`* gives the same result.

Alhough tags are arbitrary, one does not only need to store data but also want the visualization module to interpret it. The currently processed tags are *`color`*, *`size`* and *`shape`*. See the valid values of them in Table 2-2

`lazy` format:

The same as gdc, minus the ridiculous amount of puctuation characters:

```
vtx V1 color=green weight=2.7;
vtx V2 color=brown weight=1.4;
edge E1 color=yellow width=2 from=V1 to=V2;
```

It is optimized for manual input. Conversion is trivial (and probably unnecessary):

```
glay -i filename lazy -d newfile gdc
```

`native` format:

It is a format useful for PERL programmers, others may skip this paragraph: Glay is a PERL program, with C++ extensions. It keeps all the "important" data under the *`Glay::Data`* namespace. When dumping in *`native`* mode, simply all the variables under this namespace are dumped via *`Data::Dumper->Dump()`*. When importing in *`native`* mode, the input data is evaluated under the *`Glay::Data`* namespace. The format that *`Data::Dumper->Dump()`* generates is something like this:

```
%edge = (
                'E2' => {
                          'width' => '5',
                          'color' => 'grey',
                          'to' => 'V3',
                          'from' => 'V2'
                        },
                ...
             );
      %vtx = (
               'V4' => {
                         'color' => 'gold',
                         'weight' => '3.2',
                         'neighbour' => {
                                           'V3' => 'E4',
                                           'V2' => 'E3'
                                         },
                       },
                ...
              );
```

This is the default format, it can be omitted as a format-argument. The parsing of this format is extreamly fast, since no perl functions are involved, the PERL interpreter does the parsing and the evaluation. However, a great amount of care should be taken if one generates this format by hand: the connectivity information stored in the *from* and the *to* tags of edges must be consistent with the content of the *neighbour* tags of the corresponding vertices.

gem format:

It is the format of the original implementation of the Gem algorithm written by Ingo Bruss and Arne Frick. That format is rather discouraged to use with Glay since it has many unimportant restrictions while not increasing speed or usefullnes in any way. It is included just for the sake of being able to read the files of the original Gem: for exampes to test the included Gem algorithm.

Table 2-2. Tags understanded by the visualization module

color	Its value can be any colorname found in the file rgb.txt of the current X11 disribution. You can use the `xcolors` program to select one of them visually, or if you do not have it, you can see the file `rgb.txt` by saying `glay -a showrgbtxt | less` You can also use RGB colors in hexadecimal format as for example *"color=#a312ff"*, or in a shorter form of *"color=#a1f"*. Further you can use decimal numbers for R, G and B in between 0 and 255 as *"color=123,0,12"*.
size	The width in case of an edge and the diameter in case of a vertex. A lot of rescaling is going on through the visualization process, so these are really just relative values. It worth to try values between 0.1 and 1, and adjust as needed.
shape	For vertices it can be any of *sphere*, *ball*, *cube*, *pyramid*, *tetrahedron* (*ball* is equivalent with *sphere* and *pyramid* is equivalent with *tetrahedron*). For edges the value of *shape* must be one of *line*, *cylinder*, *box*, *cube*. The defaults are *ball* and *cylinder*.

xyz	It is required by the visualization module that the *xyz* tag of all vertices should be filled in appropriately with the values of the three dimensional coordinate vectors of the points. You can either choose to have itt filled automatically by a layout algorithm, or fill it manually. See the following two sections on this topic.

2.1.3. Calling a layout algorithm

Layout algorithms assign coordinates to the points based on the connectivity information (topology) of the graph. The built-in layout algorithms are based on an energy-function, random walk, and simulated annealing. See Table 2-3 for a description of them.

You can call a layout algorithm *alg_name* this way:

```
glay -i filename -a layout alg_name optional_parameters -d filename
```

Naturaly only the *-a layout alg_name* is the important part. If you know how to adjust the behaviour of the algorithm you have chosen by constructing a parameter file *par_file*, you need to ask glay to read it in *before* running the layout algorithm:

```
glay ... -p par_file -a layout alg_name optional_parameters ...
```

See Chapter 3 for information on the valid contents of the parameterfiles.

The algorithms are capable of continuing their operation if restarted appropriately. See Chapter 3 for information on that.

Table 2-3. Built-in layout algorithms

Name	**Description**
Gem	A very fast algorithm using integer arithmetic and very simple rules. The quality of the result varies, it worth to try. It is developed by Ingo Bruss and Arne Frick, (Fakultät für Informatik, Universität Karlsruhe). Their actual code has been used while applying just some technical modifications.
J	An algorithm built from scratch utilizing some bright ideas of Gem, although quality and reusablility (hence variability) were considered as the most important goals, not speed. It is the main and best-tested algorithm we use. It has beed first implemented by Johanna Becker, later reworked by Baldvin Kovács.
JX	JX stands for "J eXtended", and has two important goals: First, demonstrating the possibility (and maybe the ease) of the extension of the *J* algorithm. Second, to implement something useful. That useful extra knowledge of *JX* is that one is able to constrain the coordinates of specific points to affin subspaces. In human words it means that one can constrain the positions of the points to planes or lines in the 3 dimensional space, specifying the particular plane or line point-by-point. See Chapter 3 for deatails on how to do it. There is no other difference between *JX* and *J*, so you can use *J* if you don't need this particular extra functionality.

2.1.4. Specifying coordinates manually

The only effect of the layout algorithms is that they fill out the `xyz` tags of the vertices. If you have some useful values, you can set aside of using the layout algorithms, however, you really need to fill the `xyz` tag of *all* the vertices.

Technically speaking the `xyz` tags needs to be arrays of length 3. Table 2-4 shows how to fill them using the built-in fileformats.

Table 2-4. Format of the coordinate parameters

Format	Example
gdc	`vtx "label", "xyz=[3, 1.6, 12]", ...;`
lazy	`vtx label xyz=3,1.6,12 ...;` Note that there are no spaces between the numbers, since spaces are the separators in this format.
gem	There is no way to specify coordinates for the points using this format.

native	Here is a snippet from a "native" file showing how the `xyz` information appears in it:
	`'V2' => {`
	`    'color' => 'brown',`
	`    'weight' => '1.4',`
	`    'neighbour' => {`
	`        'V4' => 'E3',`
	`        'V3' => 'E2',`
	`        'V1' => 'E1'`
	`    },`
	`    'xyz' => [`
	`        '4.7',`
	`        3,`
	`        7`
	`    ]`
	`}`

Note that simply filling out some coordinates does not instruct a layout algorithm to keep the particular vertices in that fixed position. If you start a layout algorithm, it usually clears the coordinate information, and provides completely new ones. It is up to the certain algorithm to honor some information arriving at the `xyz` data member of the vertices, see Chapter 3.

Also note that you don't need to provide the graph and the coordinates at once: you can write your own action (see Section 2.2) to fill out the coordinate information later:

```
glay -i filename -a my_xyz_action ...
```

2.1.4.1. Summary

It this section we saw how to start Glay, import files, run some actions (for example visualize graphs) and dump data to files. An all-in one example is

```
glay -i filename -p J.mypars -a layout J -d layed.glay -a visual
```

In the following section we see how to get a little more control by implementing some very simple own actions. Knowing the Perl language helps very much, although probably sometimes one can tweak a little some example actions to get the needed results without really getting acquainted with Perl itself.

2.2. Getting more control by action files

Reproduciblity can be a very important expectation while working with graphs. This can be fulfilled easier if you need to write down the actions that you initiate even if you are just asking Glay to set the sizes of all the vertices to 0.3.

Maybe the previous argument seems to be only a poor apology for not having a complete and advanced gui interface. However, experience shows that providing a scripting way of control is very important: good user interfaces can built on top of them, but it is hard to create a scripting environment for a gui-based program design. (Some well-known cad programs are very good examples for both category.)

In this section we will see how to use action files. It is very straightforward to use or combine them without any knowledge of programming. Creating completely new actions however requires the knowledge of the Perl language.

2.2.1. Using existing action files

You can initiate an action by using the *`-a`* parameter, followed by the name of the action file and maybe some arguments of that particular action, for example

```
glay ... -a sizes 0.2 0.1 ...
```

means to run the builtin action called *`sizes`* with parameters 0.2 and 0.1.

Actions are searched first in the path given for Glay in the *`GLAYACTIONPATH`* environment variable, and then in the builin action paths. If *`GLAYACTIONPATH`* is empty, then it is initialized to a simple "." which means "current directory".

You can list all the actions seen available by Glay with the command

```
glay -a list_actions | less
```

There are a lot of small actions needed when visualizing a graph, the already mentioned `sizes` being one of them. See for a list of the builtin actions.

FIXME: chap_reference megirando

2.2.2. Combining action files into a new action file

Action files are plain series of commands, no invocations like "#include <stdio.h>" are necessary. You can run actions with the "action" command, followed by the name of the action and optionally the parameters of that particular action:

```
action "keep_biggest_component";
action "layout", "J";
action "sizes", 0.2, 0.1;
action "dump", "layed.glay";
action "visual";
```

If you write these lines into a file with a text-editor, and give it a name, you can use that name after the *`-a`* argument of Glay:

```
glay -i filename -a my_brand_new_action_filename
```

Word for Windows as a text-editor is not an optimal solution, since you really need plain text with no formatting. If you don’t know what it means, please ask about it from your system administrator or some friends.

2.2.3. Creating new action files

This is how Perl programming comes into the picture. An action is simply a file evaluated in the namespace *`Glay::Action`*. It can contain any valid Perl code.

Since any action is run in the namespace *`Action`* it is able to use the subroutines defined there. They are the following:

`action "actionname", "opt_arg1", ...;`

Runs the specified action with optional arguments as if it were invoked from the command line with the command line with the *`-a`* parameter.

`input "filenaname", "opt_format";`

Inputs the given file. It is almost equivalent with *`action "input", "opt_format";`*. The only extra it doest that it checks the validity of the *`filename`* and dies if it is neither a minus nor is an existing file. If "filename" is a minus sign, then it reads from the *`IMPORT`* filehandle, which is *`STDIN`* by default.

`dump "filename", "optional_format_name";`

Dumps the "data" to the given file if format is "native". For other formats it usually dumps the appropriate portion of the "data". The "data" is defined as: all the variables of the "Glay::Data" namespace having names not starting with an underscore. In most cases it means that the *`%Glay::Data::vtx`* and *`%Glay::Data::edge`* hashes are dumped. Calling this subroutine is actually equivalent with calling *`action "dump", "filename", "optional_format_name";`*

parfile "filename";

Reads the given parameter file, and actually "unshifts" it into a list variable. Later the *Glay::Parameters::use_parameters();* function can be used to evaluate the contents of that list variable one by one in the calling namespace. Hence parameter files without a *package something;* in them are evaluated in all the namespaces of the actions using this method of parameter setting, that is calling the *use_parameter* function.

There are some very important objects that are imported from other places, namely the *Glay::Data;* namespace and the *Glay::Channels* namespace. These are:

%vtx

A hash in *Glay::Data* containg the edges of "the graph". The internal structure is that the vertices are stored in a hash, labelled by unique identifier labels, all of them being also hashes. The keys of a hash representig a vertex are called "tags". A vertex can have any "tags" you'd like, just the tag "neighbour" is used and maintained internally. See the example above for the format called *native* to see how a vertex is stored in memory.

You should never drop vertices that have appropriate edges going into them. Dropping vertices is best left to the function Glay::Data::Drop::drop_vtx.

Naturally you can store more graphs then one in the memory, even in the *Glay::Data* namespace, although actions manipulating only on one graph will look for this particular variable. A useful technique to switch between graphs is to store them under different names then the defaults and set the **Glay::Data::vtx* and **Glay::Data::edge* globs appropriately before calling a particular function. Be aware that the values of *&Glay::Data::vtx* and *&Glay::Data::edge* functions is better preserved, since they are used by a lot of actions!

vtx "label", "tag1=value1", "tag2=value2", ...;

A function in *Glay::Data* to help accessing the main vertex hash *%vtx*. If a vertex "label" is not yet exists it creates one, if it does, then it just adjusts the appropriate tags of it. If a value of a tag has braces or brackets around them, then it is evaluated, hence you can even use complete structures dumped by *Data::Dumper*. The most important use of this feature is that the command

```
vtx "some_vertex", "xyz=[2,3,4.2]", ...;
```

does not store the value for *xyz* as a string, but parses it to a list of scalars.

It is advisable to use this function for creating new vertices, although they can also be simply created by commands like

```
$vtx{new_label}{color}="red";
```

%edge

A hash in *Glay::Data* containg the edges of "the graph". The internal structure is that the edges are stored in a hash, labelled by unique identifier labels, all of them being also hashes. The keys of a hash representig an edge are called "tags". An edge can have any "tags" you'd like, just the tags "from" and "to" have fixed meanings: they are the labels of vertices that the edge connects (see the example above for `native` file format)

You should never rewire or drop edges (changing the values of `from` and `to` tags without taking care of the `neighbour` tags of the appropriate vertices. Rewiring edges is best left to the function "edge", and dropping edges is best left to the function *`Glay::Data::Drop::drop_edge()`*.

`edge "label", "tag1=value1", "tag2=value2", ...;`

A function in *`Glay::Data`* to help accessing the main edge hash *`%edge`*. If an edge "label" is not yet exists it creates one, if it does, then it just adjusts the appropriate tags of it. If a value of a tag has braces or brackets around them, then it is evaluated like for vertices.

It is advisable to use this function for creating new edges or altering their *`from`* and *`to`* tags, since it adjusts the *`neighbour`* tags for the appropriate vertices.

2.2.3.1. Example

Now we know the most important things to start writing new actions. Let's say we needed an action that sets the size of the vertices to 0.1 if they didn't have yet a size, and doubles the size of them if they have a size already. The action would look like something like this:

```
foreach my %vtxname (keys %vtx) {
  if ( exists $vtx{$vtxname}{size} and defined $vtx{$vtxname}{size} ) {
     $vtx{$vtxname}{size} *= 2;
  else {
     $vtx{$vtxname}{size} = 0.1;
  }
}
```

That's all! Putting that into a file being in the path *`GLAYACTIONPATH`* you can use it as your new action.í

Chapter 3. On the builtin graph layout algorithms

As already said in Chapter 2 there are currently three layout algorithms available in Glay: `Gem`, `J` and `JX`. `Gem` was developed by Ingo Bruß and Ame Frick at the University of Karlsruhe. Their webpage is "http://www.info.uni-karlsruhe.de/~frick/gd/". The `J` algorithm was developed by us. Algorithm `JX` is a simple extension of `J`. In this chapter you find the theoretical description of these three (basically two) different layout algorithms.

3.1. The GEM algorithm

The Gem algorithm was developed by Ingo Bruss and Arne Frick, (Fakultät für Informatik, Universität Karlsruhe). The homepage of Gem is http://www.info.uni-karlsruhe.de/~frick/gd/gem3Ddraw/ The original paper on Gem: ftp://i44ftp.info.uni-karlsruhe.de/pub/papers/frick/gd95p.ps.gz

We changed the input part and the visualization part of the gem3d program, but the algorithm is the original.

You can still use the original gem input format, but now the input/output is separated, and we refer only to the algorithm when we talk about gem.

3.1.1. The gem input format

The gem input file format is the following:

1st line:

> On the first line there are 2 numbers and a character separated by spaces: The number of the vertices, the number of the edges, and a character "n". ("n" would mean that the graph is undirected however the handling of directed graphs was never implemented by the authors of gem.)

vertices:

> The next lines define the vertices. The first number in these lines is the index of the vertex (The first vertex should always have the number 1), then follows the shape of the vertex (quader, sphere, cone, cube, ...), finally the color (red, green, blue, balck, white, yellow, magenta, pink). Glay actually allows any colors and shapes valid elsewhere also in the gem input files.

edges:

> The last part of the gem input file defines the edges. It's a simple edgelist: each line of it has 2 numbers separated by some space. Those are the indeces of the vertices incident to the edge.

3.1.2. The algorithm

The gem layout algorithm is a force-directed simluated annealing algorithm. Each vertex is affected by four forces: gravity, vertex repulsion, edge attraction, and a random force.

A very important and interesting aspect of the algorithm is how it cools the system. It defines a "local temperature" of each vertex, and modifies those independently. The "global temperature" is defined as the sum of the locals, but its only role is to stop the algorithm if it gets too low.

We talk about these later, now let's iterate the forces that act in the algorithm:

a) gravity:

> The program calculates the center of mass, and after each vertex movement updates it. The garvity is a force directed towards the center of mass and depending on the distance from the center of mass linearly. The linearity constant is a specifiable parameter of the algorithm.

b) vertex repulsion:

> Each vertex repulses each other. The repulsion depends on the constant "EdesSQR", which is typically equals to the (number of edges) / (number of vertices).

c) edge attraction:

> If two vertices are neighbours in the graph, they attract each other. The force of this attraction is asymmetric: the vertices with higher degree are attracted weaker. Additionally there's a rule that really close vertices don't attract each other.

d) random force:

> At each step there is a random force the magnitude of which is depends only on the number of vertices and edges.

3.1.2.1. A step of the algoritm

In each step the algorithm moves all the vertices one by one, in an order randomized at the beginning of the step.

The move of a vertex depends only on the direction of the resultant force, not on the magnitude of it. The length of the move is "equal" to the temperature of the vertex.

3.1.3. Specialities

The gem algorithm has protection against oscillation and rotation of vertices. These operate by heating and cooling the points depending on the directions of the sequential steps. In this algoritm this is the only source of the temperature-change: no global cooling at all! However, the vertices do not have impulse, they just step from a standing position each time, so this "oscillation protection" and "rotation protection" has a great deal of randomness in it: they realize roughly the global cooling, and a little "plus". And this little "plus" is what we think its power is...

a) oscillation-protection:

> If the direction of the new step of the vertex is almost opposite to the direction of the former step, then we cool the vertex. If those directions are nearly equal, then we heat the vertex creating a positive feedback to encourage the motion.

b) rotation-protection:

> The new temperature of the vertex also depends on the angle of the turn. If the vertex turns less, then it is cooled slower.

3.1.4. Our experiences and opinions

* The algorithm works very fast.

* It is very strong in the case of some symmetric, sphere like graphs.

* We feel that it lacks a little bit in the clarity of using notations of physics like temperature and heat, impulse, length, weight, degree, or force.

* When we experimented with it and we needed to produce a layout of some real life biological network we needed some extra parameters, like for adjusting the strength of the attractive or the repulsive forces. The algorithm actually proved to be very sensitive to a lot of constants.

* In some cases adding some global cooling (as present in traditional simulated annealing algorithms) made the results better.

Finding the ideas in gem very interesting but the original source having these issues made us decide to write our own algorithm. We tried to incorporate all the best ideas, and make it even better. What we got is certainly different. It is slower, but for some cases we had nicer results. However, for some cases -- like for the buckey-ball -- Gem seems to be unbeatable and we think the original Gem very well deserves to be kept as one of the choices among the layout algorithms we have in Glay.

3.2. The J algorithm

The J algorithm is a force-directed simulated annealing algorithm. The vertices are moving in a potential space generated by three type of forces: gravitation, vertex repulsion and edge attraction.

The algorithm is divided into rounds. In each round all vertices make a step.

3.2.1. A step of a vertex

The step of a vertex is constituted of 2 parts: a random movement and a force-driven move.

a) The random move

> At the beginning of each step we try to move the vertex randomly. The length of the movement depends on the temperature of the vertex and the length of the last step of the vertex. If by chance we stepped to a

position with a lower potential or we did not but the temperature of the vertex is high enough then we leave the vertex at the new position. Otherwise we roll back the move, and make a very small random step. The length of that small movement depends on the temperature and the (geometric) size of the system.

b) The force-driven move

We calculate the forces applied to the vertex:

i) Gravity: The program calculates the center of mass, and after each vertex movement updates it. The garvity is a force directed towards the center of mass and depending on the distance from the center of mass linearly. The linearity constant is a specifiable parameter of the algorithm.

ii) Vertex repulsion: Each vertex repulses each other. The repulsion force depends reciprocally on the square distance between the vertices and linearly on the mass. If the mass of the vertices is not specified then all vertices are considered having unit mass. There is a specifiable scaling parameter of the repulsion force.

iii) Edge attraction: The edges are similar to a spring, they converge to an uniform optimal length. (This is a specifiable parameter too.) If two vertices are neighbours in the graph and they are far away, then they attract each other. The attraction force is a square function of the distance between the vertices. There is a specifiable scaling parameter of the attraction force. If the two vertices are close then they repulse each other like in ii).

After we have calculated the forces we try to move the vertex towards the direction of the resultant force. We find the new position of the vertex by an iteration.

1. At first the length of the movement is equal to the length of the previous step. If the potential in the new position is lower then we put the vertex there, otherwise we leave it where it is.

2. Now depending on the success of the step, we either double or halve the step length.

3. We try to step again. The step is succesful if it goes to a lower potential place and unsuccessful if it does not.

4. Go to 2. if we did not exceed the number of cycles yet.

The number of the cycles we make is a specifiable scaling parameter of the algorithm. There is also a maximal and a minimal limit of the length of the movement of the vertex.

3.2.2. The cooling function

After moving the vertex we modify the temperature of it. The new temperature depends on the old temperature, on the average potential of the vertices, on the rate of the new and the old potential, and on a lot of specifiable parameters.

We tried to cook the best soup out of all the available parameters, and make up the best working cooling function. The result is what we left in our program. However, we found that a simple 0.95-exponential function is almost as good as this.

3.2.3. Specialities

If two vertices are distant and they aren't neighbours then the repulsion force between them is very small. When we calculate the forces or the potentials we don't care about these small forces.

We try to detect the oscillations. The vertices have a "history", each vertex remember some previous positions. (The size of the history is a specifiable parameter.) When If the vertex has oscillation, we cool it better.