graPharo

Nina Medić

2019

Content:

Introduction **2**

Quick graph review **2**

Installation **3**

Layouts **4**

4.1 Linear Layouts 4

4.2 Grid and Cell Layout 6

4.3 Circle Layouts 7

4.4 Tree Layouts 9

4.5 Cluster Layout 11

4.6 Force Based Layout 13

Further work **14**

1. Introduction

GraPharo is a library which contains basic graph structure and a set of layouting algorithms of that structure.
Different ordering techniques will give us the opportunity to see the problem from multiple points of view. Many of the layouting algorithms featured here, have already been implemented, but they are usually a part of some other project, and the only place you can apply them is as a part of the mentioned project. Therefore we made our new layout library as independent as possible, so it can be used with many different visualization engines (like Telescope and Roassal) and with any already given or newly made graph structure.

All the visualisation in this booklet will be done in Telescope.

2. Quick graph review

In this chapter we will give some basic definitions about graph as a structure, which are necessary to understand for the future explanations and usage of this library.
Let's start with a basic definition of graph:

> *A graph is a set of vertices and a collection of edges that each connect a pair of vertices*.[1]

Mathematically written G=(V, E). Set V can represent any kind of objects, while edges are relationships between them.
Next thing we need to know are adjacent vertices:

> *Neighbours are vertices connected with the same edge.*
>
> *A set of neighbours of one given vertex v is called neighbourhood.*
>
> *In-neighbours are vertices, whose arcs (edges) end in a given vertex v.*
>
> *Out-neighbours are vertices whose arcs (edges) start from given vertex v.*

[1] Definition from: Sedgewick R., Wayne K. (March 2011). Algorithm, Fourth edition. United States: Pearson Education, Inc.

Explained in another way: vertex v has children if his set of out-neighbours is not empty. By analogy, vertex v has parents if a set of in-neighbours is not empty.

> *Path graph is the one where all the vertices can be arranged in a linear sequence.*
>
> *Cycle graph is a path where the first and the last vertex are connected.*
>
> *Acyclic graph contains no cycles.*
>
> *Graph G is connected if for any partition A and B such that A ∪B= G, exist at least one e∈E with one end in A and the other in B.*[2] *Simply said: A graph is connected if there is a path from every vertex to every other vertex in the graph.* [3]

3. Installation

Execute the following code in the Playground in your image.

```
Metacello new
 baseline:'GraphLayout';
 repository: 'github://medicka/graPharo:master';
 load.
```

This way you will import both the graph structure that we made and layouts that can be applied on it. It is possible to use the layouts on another graph structure, one that you created on your own, but in this situation you need to add traits to appropriate classes. As they are describing the vertex and edge, we put them in the structure package.

[2] Definition from: Bondy J. A., Murty U. S. R. (September 2007). Graph theory.
[3] Definition from: Sedgewick R., Wayne K. (March 2011). Algorithm, Fourth edition. United States: Pearson Education, Inc.

4. Layouts

There are a number of layouts which we implemented in this library. Here we will give an example and explanations about each of them.

4.1 Linear Layouts

It is easy to conclude the type of ordering in this subset of layouts, as their name is really self explanatory.
We position vertices on a straight line. There are two options: horizontal (Image 1) and vertical (Image 2) line.

Code example:

```
|visu layout i|
visu := TLVisualization new.
layout := GraphVerticalLineLayout new.
i := 1.
(visu > #group1) styleSheet
        width: [ :e |  i * 5 + 10. i := i + 1. ];
        height: [ :e | i * 10 + 7. i := i + 2. ];
        shape: TLRectangle.
(visu > #group1)
        addNodesFromEntities: GEdgeDrivenLayout allSubclasses ;
        connectFollowingProperty: #superclass;
        layout: layout.
visu open.
```

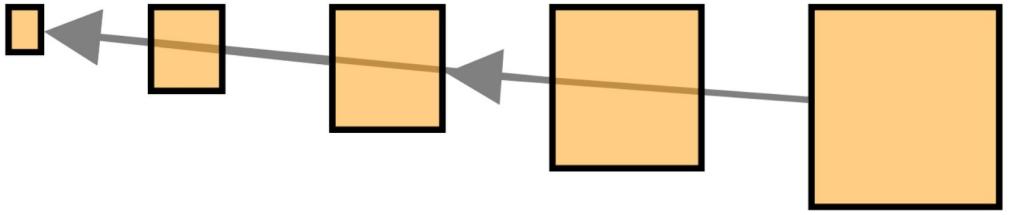

Image 1: Horizontal Line Layout

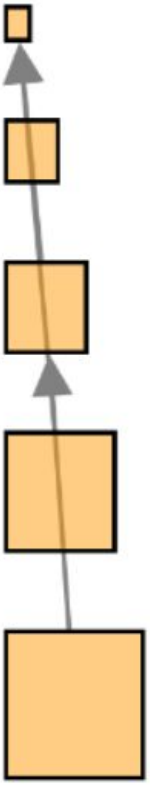

Image 2: Vertical Line Layout

This layouts allow you one more thing, and that is to change the alignment of the positioning (the part of the drawing pane to which the picture of a graph is going to be placed). In case of horizontal one, the default alignment is top, but you can change it to be bottom or center. In case of vertical, the default is left, but we can adjust it to be both center or right.

4.2 Grid and Cell Layout

Grid layout (Image 3) takes the number of vertices and calculates how many columns and rows will be needed to position vertices in a grid line structure.

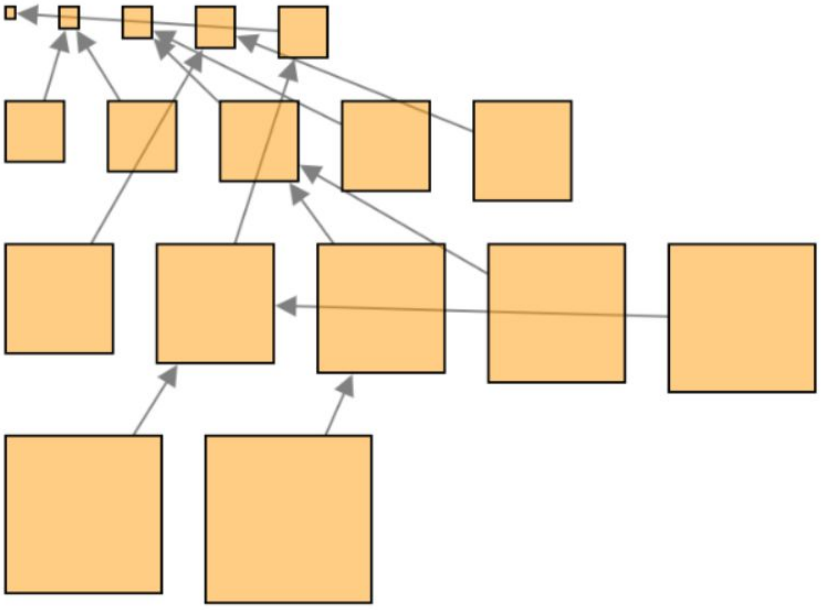

Image 3: Grid layout

Code example

```
|visu layout i|
visu := TLVisualization new.
layout := GraphGridLayout new.
i := 1.
(visu > #group1) styleSheet
          width: [ :e | i +10. i:=i+1.];
           height: [ :e | i+10. i:=i+1];
           shape: TLRectangle.
(visu > #group1)
           addNodesFromEntities: GraphLayout  allSubclasses ;
           connectFollowingProperty: #superclass;
           layout: layout.
visu open.
```

Similar to that Cell layout (Image 4) also makes the grid structure, but each field in a grid is a cell in which we will place a vertex. The size of the cells is the same in each row, based on the biggest vertex that will be placed in it.

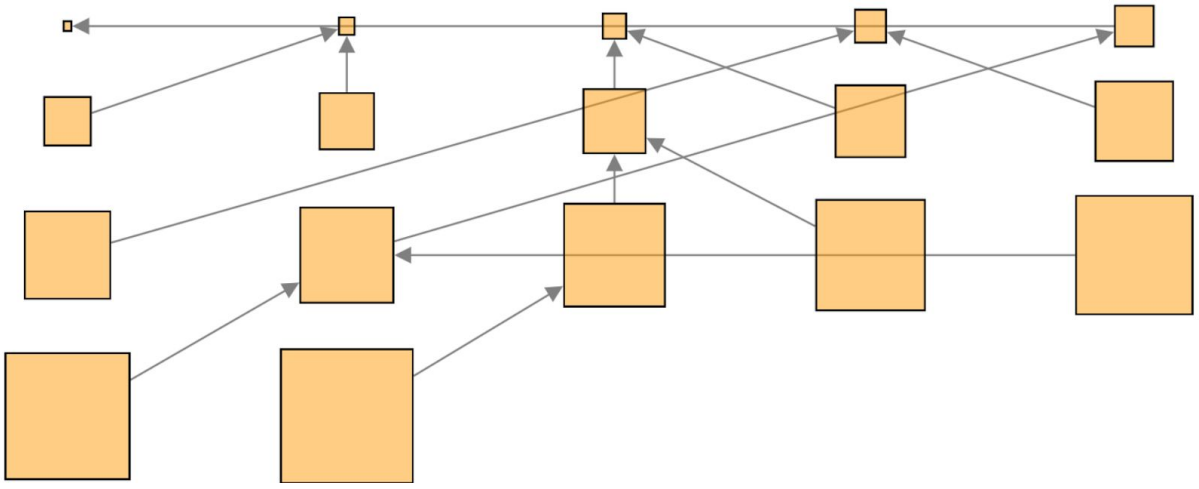

Image 4: Cell Layout

4.3 Circle Layouts

As the name clearly says, we choose to position out vertices on a circle. There are three types of circle layout:

1. Circle (Image 5), it is a regular circle, with no special conditions.

Code example:

```
|visu layout i|
visu := TLVisualization new.
layout := GraphCircleLayout new.
i := 1.
(visu > #group1) styleSheet
        width: [ :e | i * 5 + 5.  i := i + 1. ];
        height: [ :e | i * 2 + 7. i := i + 1. ];
        shape: TLRectangle.
 (visu > #group1)
 addNodesFromEntities: GraphLayout allSubclasses ;
 connectFollowingProperty:  #superclass;
 layout: layout.
 visu open.
```

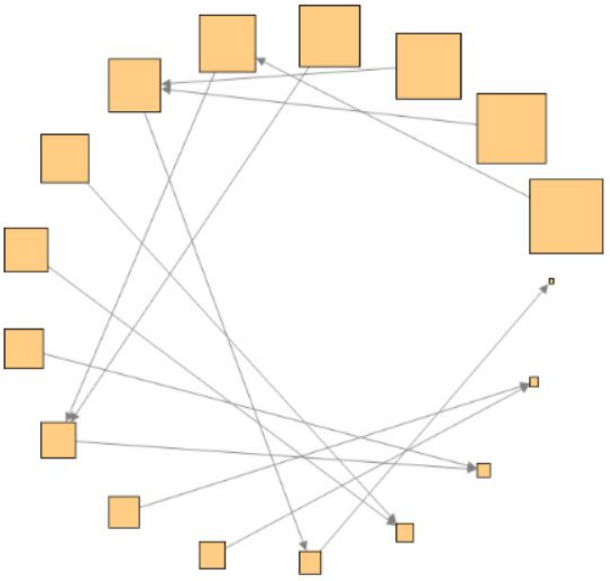

Image 5: Circle Layout

2. Equidistant Circle (Image 6), here we want the gap between the edges of vertices positioned next to each other, to be equal.
3. Weighted Circle (Image 7), in this case the size of the gap depends on the size of a vertex (the bigger the vertex, the bigger the gap).

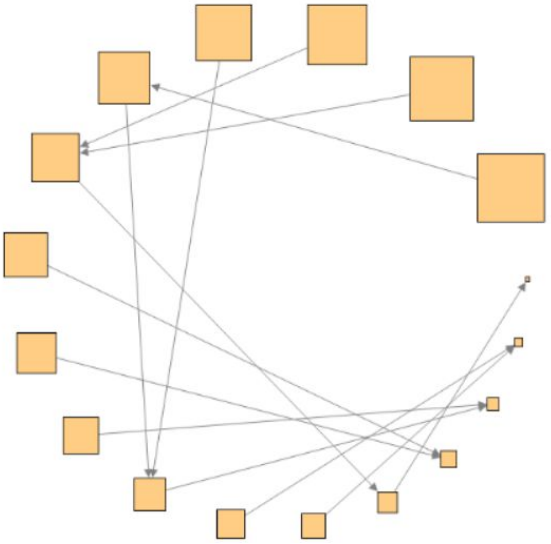

Image 6: Equidistant Circle

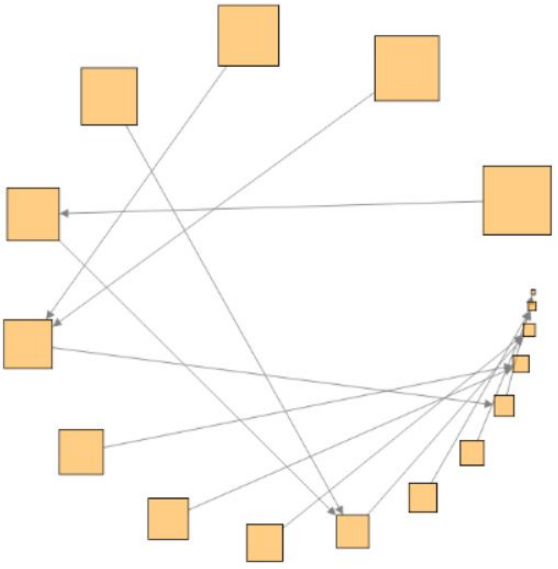

Image 7: Weighted Circle

4.4 Tree Layouts

Trees are special types of graphs, they are connected and acyclic.[4]

This layout will make trees out of our given graph. If the graph is connected we will have only one tree, opposite, we will have as many trees as there are disconnected parts in the graph. As we know every tree has a root node, the algorithm looks for vertices without parent, and chooses those for the roots. Then he puts all his children on the next level, after that he looks for the children of those children and puts them in the third level, and so on.

Code example:

```
|visu layout i|
visu := TLVisualization new.
layout := GraphTreeLayout  new.
i := 1.
(visu > #group1) styleSheet
            width: [ :e | i +10. i:=i+1.];
            height: [ :e | i+10. i:=i+2];
            shape: TLRectangle.
(visu > #group1)
            addNodesFromEntities: GAbstractCircleLayout   allSubclasses ;
            connectFollowingProperty: #superclass;
            layout: layout.
visu open.
```

Traditionally trees are built from top to bottom (Image 8), meaning that the root is on top, and the leafs (vertices with no children) are on the bottom of the drawing.
But sometimes changing the orientation of a tree can give us a better understanding of the data we are looking at.

[4] Definition from: Bondy J. A., Murty U. S. R. (September 2007). Graph theory.

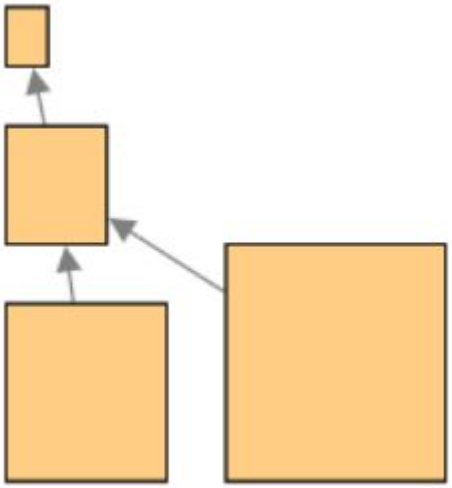

Image 8: Tree Layout with top orientation

Therefore, we have implemented 4 different orientations: top, left (Image 9), bottom and right.
There is a simple line of code that is allowing us to choose which one of these we want to apply.

```
layout orientation: #left.
```

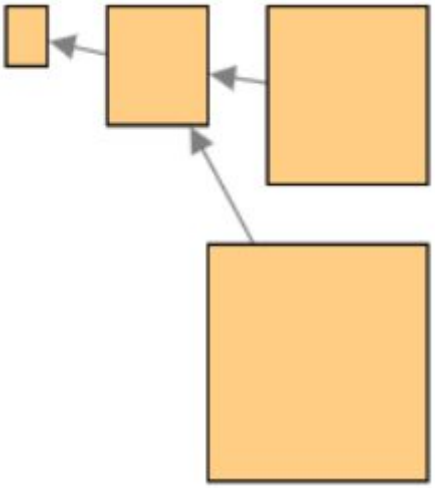

Image 9: Tree Layout with left orientation

While studying trees we noticed that there are two different types of them: layered and non layered. Difference between the two is that in one every new level of the tree structure is put on the same layer, therefore all the children on the third level, regardless of their size and the size of their parents, will be in the same line.

All the examples you have seen so far have belonged to non layered type. Now we will illustrate the layered ones (Image 10 and Image 11).

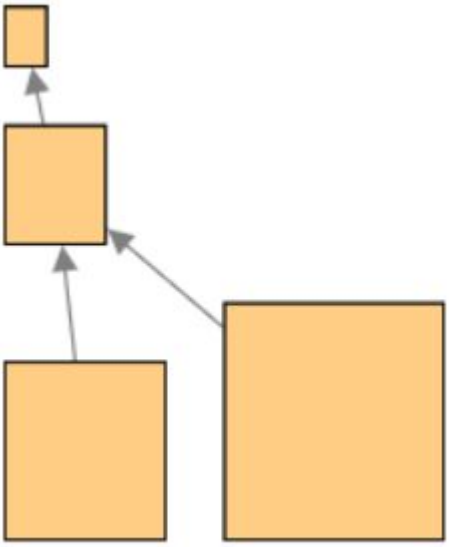

Image 10: Layered Top Tree Layout

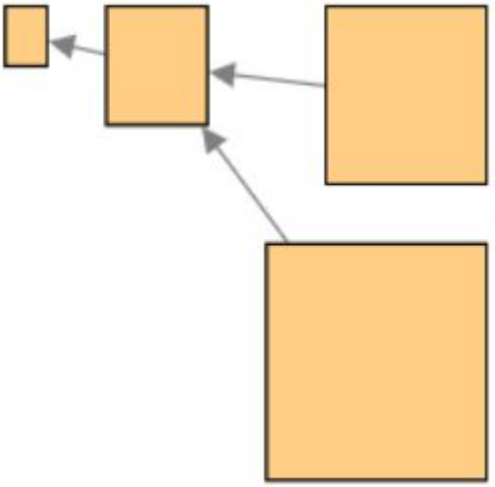

Image 11: Layered Left Tree Layout

This change we achieve by applying the next line of code:

```
layout isLayered: true.
```

4.5 Cluster Layout

Let us first explain what a cluster is.

Cluster represents a group of objects that are positioned close together.

Looking at it from a more practical perspective, in context of servers, cluster is a group of computers that are connected with each other and operate closely, as if they were one computer.
Now, cluster layout (Image 12) positions vertices in such a way that they form clusters.

Code example:

```
|visu layout i|
visu := TLVisualization new.
layout := GraphClusterLayout  new.
i := 1.
(visu > #group1) styleSheet
            width: [ :e | 10];
                  height: [ :e | 15];
                  shape: TLRectangle.
(visu > #group1)
                  addNodesFromEntities: GraphLayout   allSubclasses ;
                  connectFollowingProperty: #superclass;
                  layout: layout.
visu open.
```

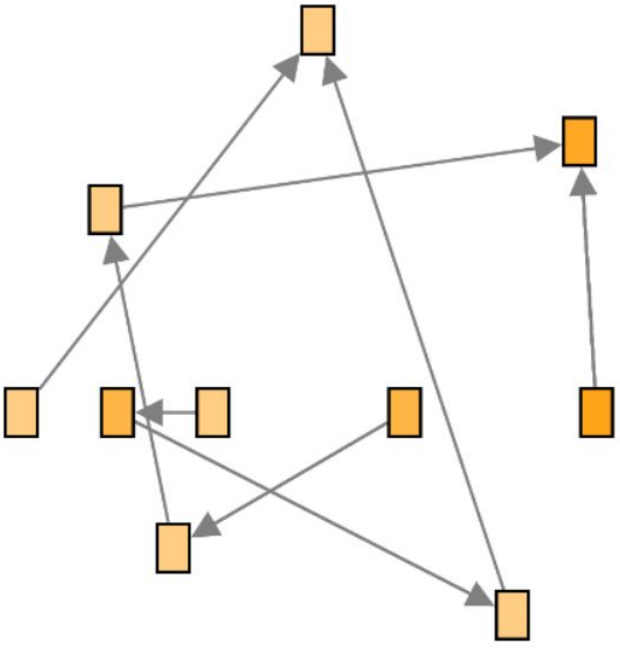

Image 12: Cluster Layout

4.6 Force Based Layout

Force based layout or force directed layout[5] (Image 13) is a drawing algorithm used to position the vertices (nodes) in a graph by using force system applied between edges and vertices. Different kind of forces used, make for different types of this algorithm.

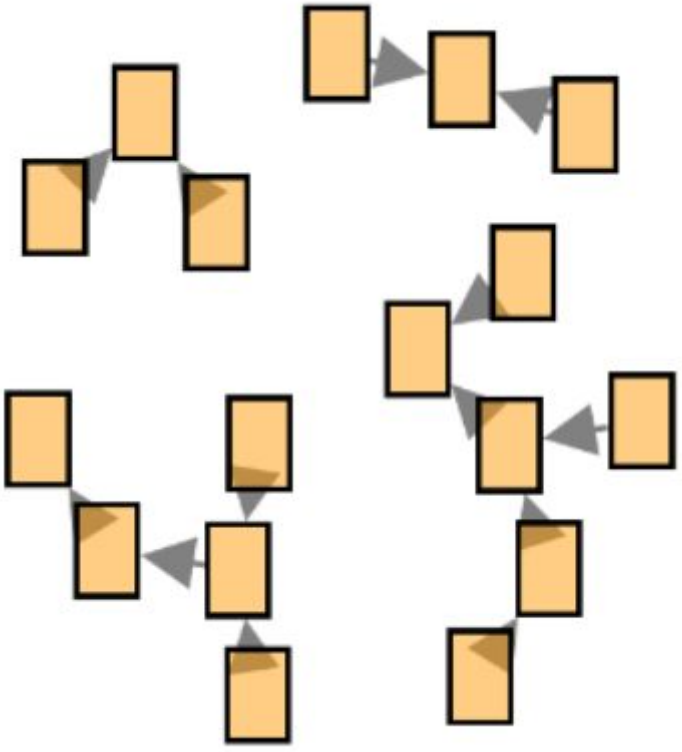

Image 13: Force Based Layout

In our library we implement spring-like attractive forces to attract pairs of endpoints of edges towards each other, and at the same time, repulsive forces, similar to ones of electrically charged particles, to separate all pairs of nodes. In a balanced state of this system, the edges tend to have a uniform length (because of spring force) and nodes that are not connected with an edge, tend to be further apart (because of electrical repulsion).

[5] Interactive Force based layout: https://observablehq.com/@d3/force-directed-graph

Code example:

```
|visu layout i|
visu := TLVisualization new.
layout := GraphForceBasedLayout new.
i := 1.
(visu > #group1) styleSheet
            width: [ :e | 10];
            height: [ :e | 15];
            shape: TLRectangle.
(visu > #group1)
            addNodesFromEntities: GraphLayout   allSubclasses ;
            connectFollowingProperty: #superclass;
            layout: layout.
visu open.
```

5. Further work

This library is not finished. There are many layouts we haven't implemented so far: different types of trees, force based ones with another set of forces used, planar and so on.
Beside this, we can optimize the existing ones. In this library we manage to make a better version of equidistant and weighted circle layouts. What does that mean?
Previous algorithm was working, but there were specific cases which caused the problem (changing the shape of vertices to rectangle, making all vertices different sizes). By optimizing it, we manage to change existing algorithm in such a way to make sure that it is working properly even in those specific cases.