snGraph*
Optimal software to manage scale-free networks

R. Maestre-Martínez**
Geographic Information System Unit
Center for Humanities and Social Sciences
Spanish National Research Council
Madrid 2010, Spain

Abstract

snGraph package provides a flexible and efficient tool for manage graphs representing scale-free network. Can be integrated into others informatic systems. It can be read easily from databases, for example using Hibernate, and build graphs using models. It can serve of bridge between data and software in order to make analisis, for example we can use Ucinet. Also permite design and implement new custom algorithms.

Resumen

El paquete de software snGraph proporciona una herramienta eficaz y flexible para la manipulación de grafos que representen redes de escala libre. Puede ser integrado en distintos sistemas informáticos. Permite leer desde base de datos fácilmente, a través de los conectores, como por ejemplo Hibernate, y construir grafos en base a modelos previamente definidos. Puede servir de puente entre los datos que se están analizando y programas del tipo Ucinet. Además permite la programación de nuevos algoritmos específicos.

Keywords: Graphs, social networks, scale-free network, data structures.

Palabras clave: Grafos, redes sociales, red libre de escala, estructuras de datos.

1. Introduction

snGraph is a tool implemented in *Java* [Java, 2010] like a package, which provides an appropriate interface to work with networks into programming environment. Encapsulates handling an array of simple and intuitive methods from a main class, but it works with special data structures in order to optimize the application performance.
This package comes from the need to represent cooperation social networks as part of the project

*Developed in Geographic Information System Unit (GIS), Center for Humanities and Social Sciences atl Spanish National Research Council. http://humanidades.cchs.csic.es/cchs/sig

**roberto.maestre@cchs.csic.es, sig.cchs@cchs.csic.es

SIG Dyncoopnet [Crespo, 2010] [1] of European Science Foundation, these networks presents a special topology, proposing the research team a flexible and efficent tool in order to view and easily share different models of cooperation social networks in a visual and simple image format. In addition, and from this package, we can make to exploit databases quickly and efficiently, and finally export the data from networks to programs that allow a deeper analysis.

Can be integrated with other software development, using the package as a bridge between systems, e.g. in Dyncoopnet project was used to read data stored in PostgreSQL [PostgreSql, 2010] through Hibernate, generate a network from a model and export the data to be analyzed Ucinet [Steve Borgatti, 2010] ,and secondly with Graphivz [Ellson et al., 2003] generating networks in visual format.
Being implemented in *Java* allowing execution on any system that has a Java virtual machine, making multi-platform tool.

2. What is snGraph

This package it is not a tool for network analysis, is a tool that allows you to create graphs representing networks with special topology from a from a programming environment. It can be used to export the graph data to other tools that includes specific algorithms for analysis [R. Hanneman, 2005] and visualization tools.

3. Internal data structures

Two main structures are used to manage networks, firstly a **hash table** and secondlyn **linked lists** , a complete description of these data structures can be found in [Thomas H. Cormen and Stein, 2001], are widely used to manage indexes.

Use is justified, 'tablas hash' and 'linked lists', due to in a data structure array type we need to know in advance the number of nodes to reserve space for each link of the graph, and also if we have n nodes, would have to reserve into the memory $n*n$ locations of memory with x size each one, where x represents the size in bytes of the data type, to represent the weight or the connection between vertices, although these vertices are not used in the future. This kind of structure is not optimal to manage scale-free networks [Barabási and Bonabeau, 2003] in which some nodes are highly connected but in general the degree of connectivity of the nodes is low which would waste a lot of memory locations. This justifies having a specific software package to manage this information correctly and allows programming a specific algorithms to model the network before analysis.

Figure [1] shows the complete data structure that uses the package snGraph.
The collision function for the hash table is $f(k) = k \text{ mód } n$, where $k \in \mathbb{N}$ is the key node to insert ans $n \in \mathbb{N}$ is the hash table size.
The nodesV_n will be positioned in order (they will be linking) within the hash table in the position that indicates the collision function.

The hash table allows us to redistribute the links as nodes grow in real timel, i.e. this hash table will grow when it exceeds a certain threshold of use, is dynamic, because they may not be known in advance the number of nodes that contain the graph. This allows us to optimize the weight of the graph, keeping only the links between nodes.

[1] References ESF: FP: 004DynCoopNet y Acciones Complementarias del Ministerio de Ciencia e Innovación: SEJ2007-29226. This project is also being funded by Macrogroups Program of the Community of Madrid: Research Network: 'Sólo Madrid es Corte. La construcción de la Corte de la Monarquía Católica',Macrogroups Program of the Community of Madrid. References CAM: HUM2007-29143-E/HIST

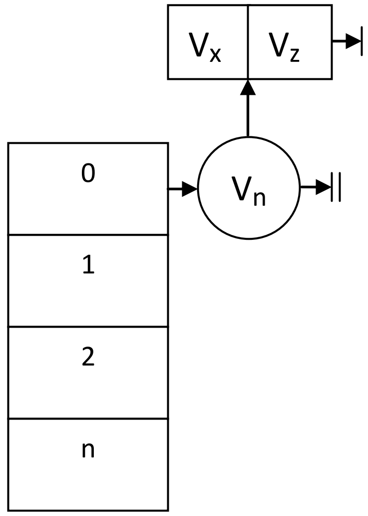

Figure 1.
Data Structures

The redistribution function $r(z) = \frac{z*15}{100}$ is activated when $r(z) > m$ where z is the number of nodes in the network and m is the size of the hash table index. Initialization $m = 1$ y logically $z = 0$, when $r(z)$ exceeds the threshold, function *rebuildIndex(s)* will be called with parameter *s* indicating the new size of hash index.

As shown in the following code, function *rebuildIndex(s)* creates a new data structure with the new size of the hash table index and insert one by one the nodes into the empty data structure.
Whenever a new node is inserted, it checks whether the threshold has been exceeded.

```
private void rebuildIndex(int size) {
    System.out.println("Rebuilding index");
    List<Integer> nodes = this.getTable().getNodeList();
    this.table = new Table(size);
    Iterator i = nodes.iterator();
    while (i.hasNext()) {
        this.getTable().insertNode(new Node((Integer)i.next()));
    }
}
```

When a link is inserted between two nodes,it checl it if the nodes exists, it seeks to hash index for the key nodeand runs through its list until it is located. The nodes, as can be seen in figure [1],have a pointer to another structure called *AdjacentNode* which keeps $V_x \in \mathbb{N}$ which is the key to the destination node of the edge and $V_z \in \mathbb{R}^+$ which is the weight of that edge,and has another pointer to the structure *AdjacentNode* as a linked list to indicate more edges from the node source V_n.

The insertion of nodes is always done in order, both nodes and edges, for this purpose go over the linked list looking for the right place to insert. Searches were cut when the key is being sought, both vertex and edge, does not exist because the current key is greater or has been found, therefore not completely go through the lists, only the worst cases.

4. Class diagram

The associated class diagram can be found in Annex figures [4].

5. Use

To begin working with the package, we import into the project with the follow sentence: **import sngraph.node.Sngraph;**
Once done, we can create a Sngraph class object, if not necessary to establish in advance the number of nodes as explained above. In the code described below shows how to insert nodes and insert edges between nodes with weight. When we insert an edge, we can establish that if the edge exists *true* add new weight to the existing weight and, on the contrary, if the value is *false* to insert the new value regardless of the previous value.

```
1 Sngraph g = new Sngraph();
2
3 g.insertNode(1);
4 g.insertNode(8);
5 g.insertNode(9);
6 g.insertNode(4);
7
8
9 g.insertWeightBetweenNodes(1,8,2.3,true);
10 g.insertWeightBetweenNodes(1,9,1.72,true);
11 g.insertWeightBetweenNodes(8,9,4.002,true);
12 g.insertWeightBetweenNodes(8,4,0.4,true);
13
14 System.out.println(g.toGraphviz());
15
16 System.out.println("");
17
18 System.out.println(g.toUcinet(null));
```

resulting in the following graph in the figure above[2] (pay attention to the thickness of the edges)
On line 14 (**System.out.println(g.toGraphviz());**) , we can see how we can get the code to generate the graph graph of Figure [2]

```
digraph Grafo{
 1->8 [label="2.3" style="setlinewidth(2.3)"];
 1->9 [label="1.72" style="setlinewidth(1.72)"];
 8->9 [label="4.002" style="setlinewidth(4.002)"];
 8->4 [label="0.4" style="setlinewidth(0.4)"];
}
```

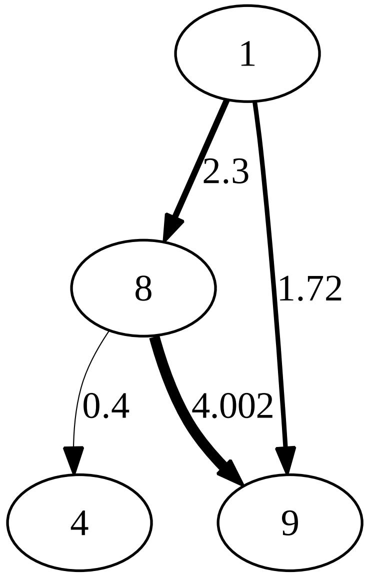

Figure 2.
Example 1

for export to Ucinet, we use the method of line 18 (**System.out.println(g.toUcinet(null));**) , and we would obtain:

```
dl n = 4 format = edgelist1
data:
1 2 2.3
1 3 1.72
2 3 4.002
2 4 0.4
```

that the graph generated for Ucinet requires a vector with associated identifiers to the nodes that were inserted in order. As follows $1 \rightarrow 1, 2 \rightarrow 8, 3 \rightarrow 9, 4 \rightarrow 4$.The following code shows the creation of the label vector v and the generation of code to Ucinet.

```
Vector<String> labels = new Vector<String >();

labels.add("1");
labels.add("8");
labels.add("9");
labels.add("4");

System.out.println("");
System.out.println(g.toUcinet(labels));
```

screen output would be as follows:

```
dl n = 4 format = edgelist1
labels:
1,8,9,4
data:
1 2 2.3
1 3 1.72
```

```
2 3 4.002
2 4 0.4
```

and representation in Ucinet of this code is shown below in Figure [3]:

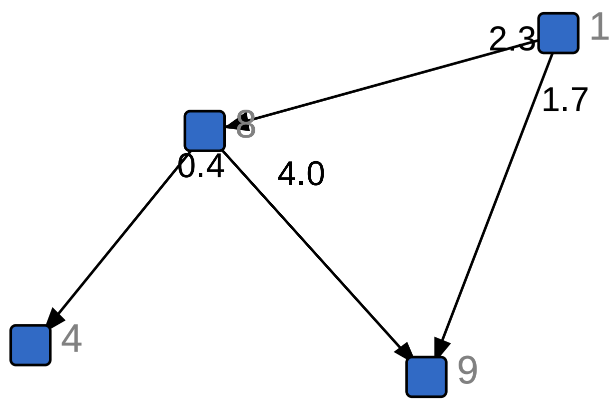

Figure 3.
Red generada para ucinet

6. Complete example

In this example, a *backtracking* [Knuth, 1968] algorithm has been programmed that will show all possible routes between two nodes of the graph given. Code is displayed, the graph generated by Graphivz visually and output performance.

```

import java.util.Vector;
import sngraph.node.Sngraph;

/*
 *  Copyright 2009-2010
 *  Consejo Superior de Investigaciones Cientificas
 *  Centro de Ciencias Humanas y Sociales
 *  Unidad de sistemas de informacion geografica
 *  Attribution-Noncommercial 3.0 Unported - http://creativecommons.org/
     licenses/by-nc/3.0/
 */
/**
 *
 * @author Roberto Maestre Martinez <roberto.maestre AT cchs.csic.es>
 */
public class Uso1 {


    //Indica si un nodo esta en el vector de visitados
    private static boolean isVisitado(int i, Vector<Integer> visitados) {
        boolean cent=false;
        int c=0;
```

```java
        while (c<visitados.size() && !cent){
            if (i==visitados.get(c)) cent=true;
            c++;
        }
        return cent;
    }

    //Procedimiento para calcular el camino mas corto entre dos nodos
    public static void backtracking(Sngraph g,Vector<Integer> visitados, Vector<Integer> camino,Vector<Integer> mejorcamino,int actual,int hasta,int N,double total,Vector<Double> mejortotal){
        //Compruebo mejor solucion
        if (actual==hasta && total<mejortotal.get(0)){
                    System.out.println("mejor encontrado, conste="+total);
                    System.out.println(camino);
                    //Copio el mejor total y el mejor camino
                    mejorcamino = new Vector<Integer>();
                    for (int x=0;x<camino.size();x++)
                        mejorcamino.add(camino.get(x));
                    mejortotal.add(0, total);
        } else {
            //Calculo los nodos a los que puedo ir desde el nodo actual
            Vector<Integer> a=new Vector<Integer>();
            for (int i=0;i<N;i++){
                if ((g.getWeightBetweenNodes(actual, g.getNodeList().get(i)) !=g.getInfiniteValue()) && !isVisitado(g.getNodeList().get(i),visitados))
                    a.add(g.getNodeList().get(i));
            }
            //Si hay nodos a los que pueda ir
            if (a.size()>0){
                //Los recorro
                for (int i=0;i<a.size();i++){
                    //Anoto
                    total=total+g.getWeightBetweenNodes(actual, a.get(i));
                    visitados.add(actual);
                    camino.add(a.get(i));
                    //Llamada
                    backtracking(g, visitados, camino, mejorcamino, a.get(i), hasta, N, total, mejortotal);
                    //Desanoto
                    total=total-g.getWeightBetweenNodes(actual, a.get(i));
                    visitados.remove(visitados.size()-1);
                    camino.remove(camino.size()-1);
                }
            }
        }
    }


    public static void main (String args[]) throws Exception{
        Sngraph g = new Sngraph();
        g.insertNode(1); g.insertNode(8); g.insertNode(9);g.insertNode(12);g.insertNode(16);g.insertNode(24);g.insertNode(50);
        g.insertNode(69);g.insertNode(70);g.insertNode(76);g.insertNode(79); g.insertNode(80); g.insertNode(89);g.insertNode(91);
```

```java
g.insertNode(93);g.insertNode(94);g.insertNode(95);g.insertNode(100);g.insertNode(101);g.insertNode(103);g.insertNode(106);
g.insertNode(110); g.insertNode(115);g.insertNode(122);g.insertNode(129);g.insertNode(135);g.insertNode(140);g.insertNode(169);
g.insertNode(170);g.insertNode(176);

g.insertWeightBetweenNodes(1, 8, 2.3, true);
g.insertWeightBetweenNodes(1, 16, 1.72, true);
g.insertWeightBetweenNodes(9, 12, 1.0, true);
g.insertWeightBetweenNodes(16, 24, 3.2, true);
g.insertWeightBetweenNodes(24, 50, 1.05, true);
g.insertWeightBetweenNodes(50, 9, 3.2, true);
g.insertWeightBetweenNodes(9, 8, 2.05, true);
g.insertWeightBetweenNodes(12, 16, 2.72, true);
g.insertWeightBetweenNodes(94, 122, 1.72, true);
g.insertWeightBetweenNodes(95, 100, 1.02, true);
g.insertWeightBetweenNodes(94, 95, 2, true);
g.insertWeightBetweenNodes(100, 1, 3.56, true);
g.insertWeightBetweenNodes(69, 8, 1.26, true);
g.insertWeightBetweenNodes(170, 176, 0.26, true);
g.insertWeightBetweenNodes(110, 129, 1.64, true);
g.insertWeightBetweenNodes(129, 12, 2.26, true);
g.insertWeightBetweenNodes(176, 50, 0.76, true);
g.insertWeightBetweenNodes(89, 176, 0.06, true);
g.insertWeightBetweenNodes(16, 122, 0.32, true);
g.insertWeightBetweenNodes(140, 169, 1.32, true);
g.insertWeightBetweenNodes(129, 135, 2.32, true);
g.insertWeightBetweenNodes(140, 140, 0.12, true);
g.insertWeightBetweenNodes(140, 94, 0.16, true);
g.insertWeightBetweenNodes(94, 140, 0.16, true);
g.insertWeightBetweenNodes(95, 100, 0.16, true);
g.insertWeightBetweenNodes(101, 176, 0.16, true);
g.insertWeightBetweenNodes(24, 95, 0.16, true);
g.insertWeightBetweenNodes(100, 110, 1.16, true);
g.insertWeightBetweenNodes(89, 101, 0.26, true);
g.insertWeightBetweenNodes(89, 170, 1.3, true);
g.insertWeightBetweenNodes(16, 89, 0.4, true);
g.insertWeightBetweenNodes(110, 169, 0.54, true);
g.insertWeightBetweenNodes(169, 69, 2.264, true);

System.out.println(g.toGraphviz());
System.out.println("");
Vector<String> labels = new Vector<String>();
labels.add("1"); labels.add("8"); labels.add("9");labels.add("12");labels.add("16");labels.add("24");labels.add("50");
labels.add("69");labels.add("70");labels.add("76");labels.add("79"); labels.add("80"); labels.add("89");labels.add("91");
labels.add("93");labels.add("94");labels.add("95");labels.add("100");labels.add("101");labels.add("103");labels.add("106");
labels.add("110"); labels.add("115");labels.add("122");labels.add("129");labels.add("135");labels.add("140");labels.add("169");
labels.add("170");labels.add("176");
System.out.println(g.toUcinet(labels));
System.out.println("");
g.print();
```

```java
            System.out.println("");

            Vector<Double> mejortotal=new Vector<Double>();
            mejortotal.add(Double.MAX_VALUE);
            Vector<Integer> visitados=new Vector<Integer>(),camino=new Vector<
                Integer>(),mejorcamino=new Vector<Integer>();
            double total=0.0;
            backtracking(g, visitados, camino, mejorcamino, 1,50, g.
                getNodeList().size(), total,mejortotal);
            System.out.println(mejorcamino);

        }




}
```

the output of above code execution consists of several parts:

1. Number of times to redistribute the hash table index
2. Output for Graphivz
3. Output for Ucinet
4. Representation of the internal structure
5. Minimum path obtained by the *backtracking*

```
//Se redistribuye el indice
Rebuilding index

//Salida para representar el grafo con Graphivz
digraph Grafo{ 1->8 [label="2.3" style="setlinewidth(2.3)"];
1->16 [label="1.72" style="setlinewidth(1.72)"];
9->8 [label="2.05" style="setlinewidth(2.05)"];
9->12 [label="1.0" style="setlinewidth(1.0)"];
12->16 [label="2.72" style="setlinewidth(2.72)"];
16->24 [label="3.2" style="setlinewidth(3.2)"];
16->89 [label="0.4" style="setlinewidth(0.4)"];
16->122 [label="0.32" style="setlinewidth(0.32)"];
24->50 [label="1.05" style="setlinewidth(1.05)"];
24->95 [label="0.16" style="setlinewidth(0.16)"];
50->9 [label="3.2" style="setlinewidth(3.2)"];
69->8 [label="1.26" style="setlinewidth(1.26)"];
89->101 [label="0.26" style="setlinewidth(0.26)"];
89->170 [label="1.3" style="setlinewidth(1.3)"];
89->176 [label="0.06" style="setlinewidth(0.06)"];
94->95 [label="2.0" style="setlinewidth(2.0)"];
94->122 [label="1.72" style="setlinewidth(1.72)"];
94->140 [label="0.16" style="setlinewidth(0.16)"];
95->100 [label="1.18" style="setlinewidth(1.18)"];
100->1 [label="3.56" style="setlinewidth(3.56)"];
100->110 [label="1.16" style="setlinewidth(1.16)"];
101->176 [label="0.16" style="setlinewidth(0.16)"];
```

```
110->129 [label="1.64" style="setlinewidth(1.64)"];
110->169 [label="0.54" style="setlinewidth(0.54)"];
129->12 [label="2.26" style="setlinewidth(2.26)"];
129->135 [label="2.32" style="setlinewidth(2.32)"];
140->94 [label="0.16" style="setlinewidth(0.16)"];
140->140 [label="0.12" style="setlinewidth(0.12)"];
140->169 [label="1.32" style="setlinewidth(1.32)"];
169->69 [label="2.264" style="setlinewidth(2.264)"];
170->176 [label="0.26" style="setlinewidth(0.26)"];
176->50 [label="0.76" style="setlinewidth(0.76)"];
}


//Salida para analizar el grafo con Ucinet
dl n = 30 format = edgelist1
labels:
1,8,9,12,16,24,50,69,70,76,79,80,89,91,93,94,95,100,101,103,106,110,115,122,129,135,140,169,170,

data:
1 2 2.3
1 5 1.72
3 2 2.05
3 4 1.0
4 5 2.72
5 6 3.2
5 13 0.4
5 24 0.32
6 7 1.05
6 17 0.16
7 3 3.2
8 2 1.26
13 19 0.26
13 29 1.3
13 30 0.06
16 17 2.0
16 24 1.72
16 27 0.16
17 18 1.18
18 1 3.56
18 22 1.16
19 30 0.16
22 25 1.64
22 28 0.54
25 4 2.26
25 26 2.32
27 16 0.16
27 27 0.12
27 28 1.32
28 8 2.264
29 30 0.26
30 7 0.76



//Representacion de la estructura interna de datos
```

```
[ 0 ]=> 8 -> 12 ( 16=2.72 ) -> 16 ( 24=3.2,89=0.4,122=0.32 ) -> 24 (
    50=1.05,95=0.16 ) -> 76 -> 80 -> 100 ( 1=3.56,110=1.16 ) -> 140 (
    94=0.16,140=0.12,169=1.32 ) -> 176 ( 50=0.76 )
[ 1 ]=> 1 ( 8=2.3,16=1.72 ) -> 9 ( 8=2.05,12=1.0 ) -> 69 ( 8=1.26 ) ->
     89 ( 101=0.26,170=1.3,176=0.06 ) -> 93 -> 101 ( 176=0.16 ) -> 129
    ( 12=2.26,135=2.32 ) -> 169 ( 69=2.264 )
[ 2 ]=> 50 ( 9=3.2 ) -> 70 -> 94 ( 95=2.0,122=1.72,140=0.16 ) -> 106
    -> 110 ( 129=1.64,169=0.54 ) -> 122 -> 170 ( 176=0.26 )
[ 3 ]=> 79 -> 91 -> 95 ( 100=1.18 ) -> 103 -> 115 -> 135


//Caminos minimos obtenidos con un algoritmo de backtracking programado en
    el ejemplo
mejor encontrado, coste=5.97
[16, 24, 50]
mejor encontrado, coste=3.3
[16, 89, 101, 176, 50]
mejor encontrado, coste=2.9400000000000004
[16, 89, 176, 50]
```

the graph generated for Graphivz shown in Figure 2 [5] and the graph generated for Ucinet in figure [6].

7. Future work

1. Generate GIS layers for georeferenced networks.
2. Implementation of the erasing operation nodes and edges in the graph.

8. Copyrigth

9. How to cite

... was built [Maestre-Martinez, 2010] sample network using the weights of kinship and trade relations ...

```
@misc{snGpraph,
  author       = {R. Maestre-Martinez},
  title        = {{snGpraph}},
  howpublished = "\url{http://www.csic.es}",
  year         = {2010}
}
```

10. Annex Figures

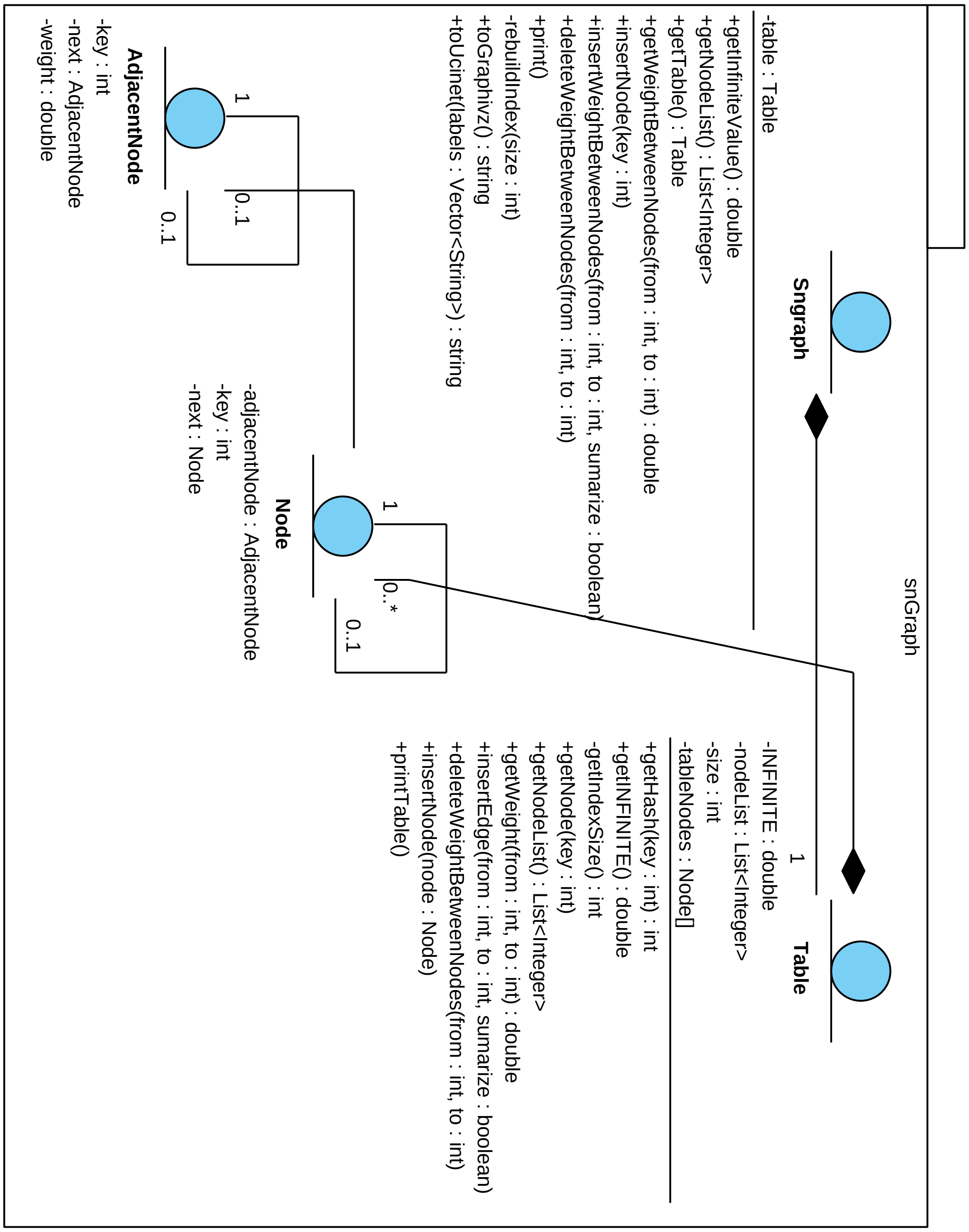

Figure 4.
Class diagram

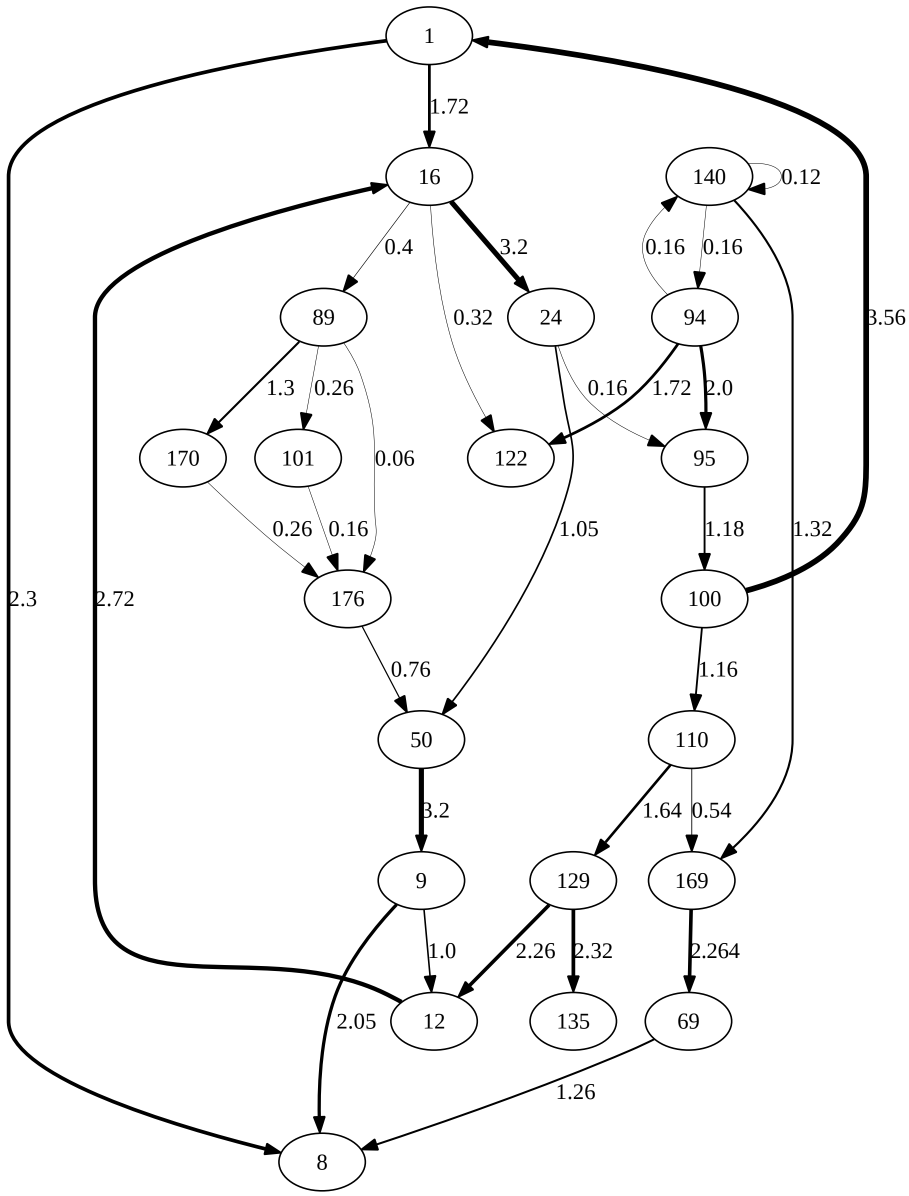

Figure 5.
Example of Graphivz graph

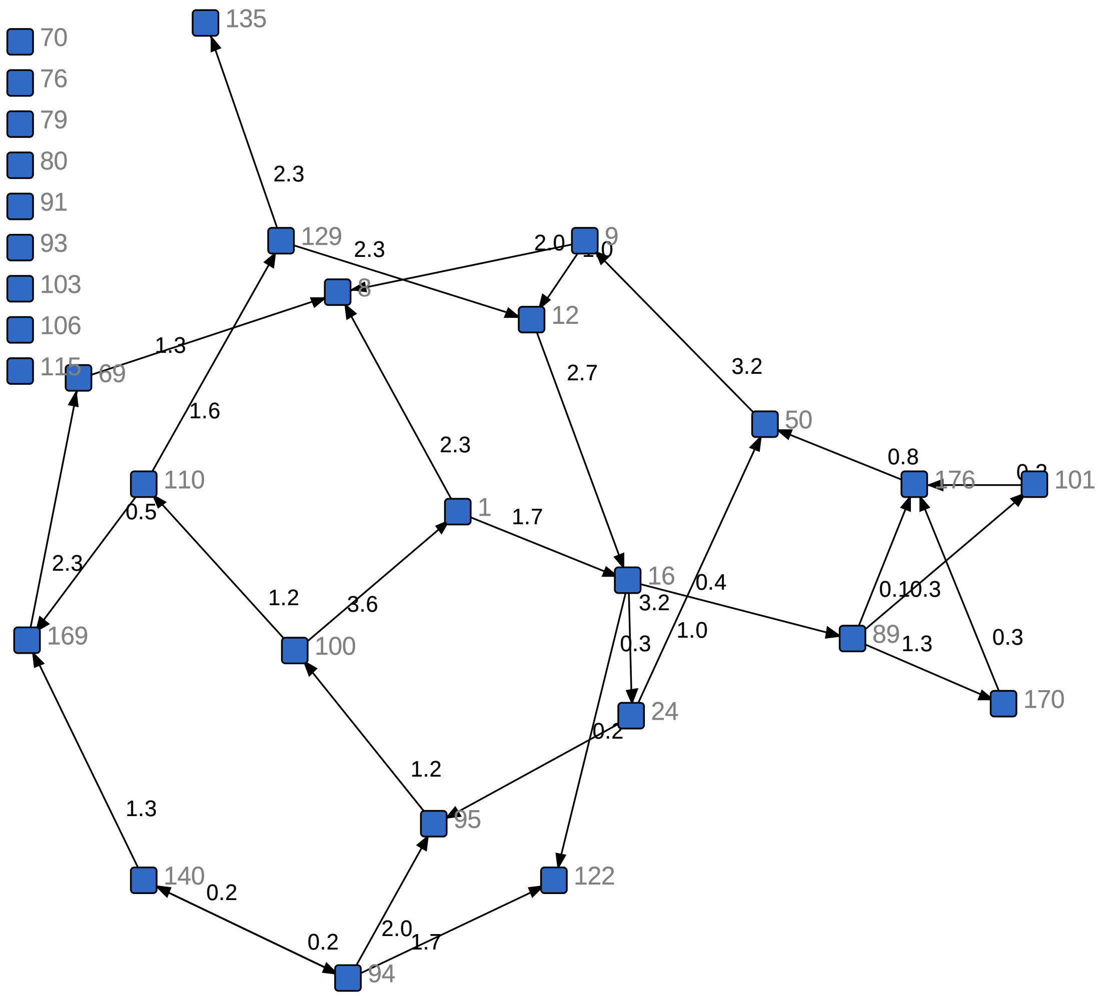

Figure 6.
Example of Ucinet graph

References

[Barabási and Bonabeau, 2003] Barabási, A.-L. and Bonabeau, E. (2003). *Scale-Free Networks*. Sci. Amer. 288, Issue 5, 60.

[Crespo, 2010] Crespo, A. (2010). Dyncoopnet. `http://www.esf.org/activities/eurocores/running-programmes/tect/projects/list-of-projects.html`.

[Ellson et al., 2003] Ellson, J., Gansner, E., Koutsofios, E., S.C.North, and G.Woodhull (2003). Graphviz and dynagraph – static and dynamic graph drawing tools. In Junger, M. and Mutzel, P., editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag.

[Java, 2010] Java, O. (2010). Java. `http://www.oracle.com/lang/es/technologies/java/index.html`.

[Knuth, 1968] Knuth, D. E. (1968). *The Art of Computer Programming*. Addison-Wesley.

[PostgreSql, 2010] PostgreSql (2010). PostgreSql. `http://www.postgresql.org/`.

[R. Hanneman, 2005] R. Hanneman, M. R. (2005). *Introduction to social network methods*. University of California, Riverside.

[Steve Borgatti, 2010] Steve Borgatti, Martin Everett, L. F. (2010). Ucinet. `http://www.analytictech.com/ucinet/`.

[Thomas H. Cormen and Stein, 2001] Thomas H. Cormen, Charles E. Leiserson, R. L. R. and Stein, C. (2001). *Introduction to Algorithms*. MIT Press, Masachussets, second edition edition.