snGraph*
Software óptimo para manipulación de redes libres de escala

R. Maestre-Martínez**
Unidad de Sistemas de Información Geográfica
Centro de Ciencias Humanas y Sociales
Consejo Superior de Investigaciones Científicas
Madrid 2010,España

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. Introducción

snGraph es una herramienta implementada en *Java* [Java, 2010] en forma de paquete, lo que proporciona una interfaz apropiada para trabajar con redes desde un entorno de programación. Encapsula la manipulación de una matriz con métodos sencillos e intuitivos desde una clase principal, pero representa la matriz internamente con estructuras de datos que optimizan el rendimiento de la aplicación. Este paquete surge de la necesidad de representar redes sociales de cooperación en el marco del

*Desarrollado en la Unidad de Sistemas de Información Geográfica (SIG) del Centro de Ciencias Humanas y Sociales del Consejo Superior de Investigaciones Científicas. http://humanidades.cchs.csic.es/cchs/sig

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

proyecto SIG Dyncoopnet [Crespo, 2010] [1] de la European Science Foundation, redes que poseen una topología especial, proponiendo al equipo de investigación una herramienta flexible y eficiente para poder visualizar y compartir de forma sencilla distintos modelos de redes sociales de cooperación de una manera visual y sencilla en formato imagen (JPG, PNG, GIF, ect...). Además, y a partir de este paquete, se pueden realizar modelos para explotar bases de datos de una manera rápida y eficaz, y finalmente exportar los datos de las redes a programas que permitan un análisis más profundo.

Puede integrarse con otros desarrollos de software, utilizando el paquete como herramienta puente entre sistemas, por ejemplo, en el proyecto Dyncoopnet se utilizó para leer los datos almacenados en PostgreSQL [PostgreSql, 2010] a través de Hibernate, generar una red a partir de un modelo y exportar los datos para ser analizados con Ucinet [Steve Borgatti, 2010] por un lado, y por otro con Graphivz [Ellson et al., 2003] para generar las redes en formato visual.
Al estar programada en *Java* permite su ejecución en cualquier sistema que disponga de una maquina virtual de java, haciendo esta herramienta multipltaforma.

2. Qué es snGraph

Este paquete no es una herramienta de análisis de redes, es una herramienta que permite crear grafos que representen redes con unas características especiales desde un nivel de programación y que puede usarse para exportar la información del grafo a otras herramientas que si incluyan algoritmos específicos de análisis [R. Hanneman, 2005] y herramientas de visualización.

3. Estructura de datos interna

Se utilizan dos estructuras principales para gestionar las redes internamente, una es una **tabla hash** y la otra son **listas enlazadas** , una descripción completa de estas estructuras de datos puede encontrarse en [Thomas H. Cormen and Stein, 2001], son ampliamente utilizadas para gestionar índices.

Se justifica el uso de 'tablas hash' y 'listas enlazadas' debido a que en una estructura de datos de tipo matriz, tenemos que conocer de antemano el número de nodos para reservar espacio para cada uno de sus enlaces, y además si tenemos n nodos reservaríamos en memoria $n*n$ posiciones de memoria de tamaño x, donde x representa el tamaño en bytes del tipo de dato que representará el peso o la conexión entre vértices, aunque estos vértices nunca se utilicen. Este tipo de estructura no es óptima para gestionar redes libres de escala [Barabási and Bonabeau, 2003] en el que algunos nodos están altamente conectados, aunque de forma general el grado de conexión de los nodos es bajo lo que llevaría a desperdiciar muchas posiciones de memoria, esto justifica disponer de un paquete software a nivel programático específico que gestione correctamente este tipo de información y que permita programar algoritmos específicos para modelar la red antes de su análisis.

En la figura [1] se muestra la estructura de datos completa que utiliza el paquete snGraph. La función de colisión de la tabla hash es $f(k) = k$ mód n, donde $k \in \mathbb{N}$ es la clave del nodo a insertar y $n \in \mathbb{N}$ es el tamaño de la tabla hash.
Los nodos V_n se irán posicionando por orden (es decir, enlazándose) dentro de la tabla hash en la posición que indica la función de colisión.

La tabla hash nos permite redistribuir los enlaces según los nodos van creciendo en nuestro grafo en

[1] Referencias ESF: FP: 004DynCoopNet y Acciones Complementarias del Ministerio de Ciencia e Innovación: SEJ2007-29226. Este proyecto también está siendo financiado por el Programa MacroGrupos de la Comunidad de Madrid: Red de investigación: 'Sólo Madrid es Corte. La construcción de la Corte de la Monarquía Católica', Programa MacroGrupos de la Comunidad de Madrid. Referencia CAM: HUM2007-29143-E/HIST

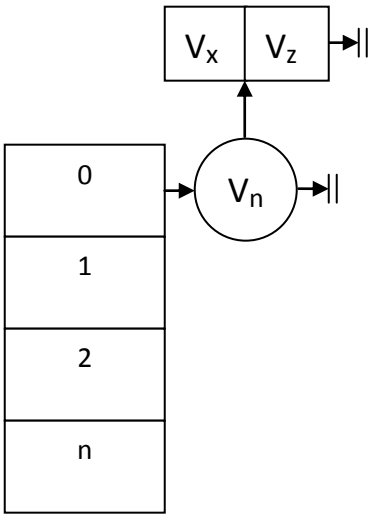

Figura 1.
Estructuras de datos

tiempo real, es decir, esta tabla hash irá creciendo cuando se supere un cierto umbral de uso, esto es dinámico, debido a que a priori puede no saberse el número de nodos que contendrá el grafo. Esto nos permite optimizar el peso real del grafo, guardando solamente los enlaces que existen entre nodos.

La función de redistribución $r(z) = \frac{z*15}{100}$ se activa cuando $r(z) > m$ donde z es el número de nodos en la red y m es el tamaño del índice de la tabla hash. En la inicialización $m = 1$ y lógicamente $z = 0$, cuando $r(z)$ supere el umbral, se llamará a una función *rebuildIndex(s)* con parámetro s que indica el nuevo tamaño del índice hash.

Como se observa en el siguiente código, la función *rebuildIndex(s)* crea una nueva estructura de datos desde cero con el muevo tamaño de índice para la tabla hash e inserta uno a uno los nodos redistribuyendo así los datos por la estructura.
Siempre que se inserta un nuevo nodo, se comprueba si se ha superado el umbral.

```
1 private void rebuildIndex(int size) {
2     System.out.println("Rebuilding index");
3     List<Integer> nodes = this.getTable().getNodeList();
4     this.table = new Table(size);
5     Iterator i = nodes.iterator();
6     while (i.hasNext()) {
7         this.getTable().insertNode(new Node((Integer)i.next()));
8     }
9 }
```

Cuando un enlace se inserta entre dos nodos, se comprueba que existen los nodos, para ello se busca el índice hash correspondiente a la clave del nodo y se recorre su lista asociada hasta que se encuentre. Los nodos, como puede comprobarse en la figura [1], tienen un puntero a otra estructura denominada *AdjacentNode* la cual guarda $V_x \in \mathbb{N}$ que es la clave del nodo destino de la arista y $V_z \in \mathbb{R}^+$ que es el peso de esa arista, además dispone de otro puntero a la estructura *AdjacentNode* en forma de lista enlazada para indicar más aristas origen desde el nodo V_n.

La inserción siempre se hace en orden, tanto en los nodos como en sus aristas, para ello se recorren las listas enlazadas buscando el lugar correcto para la inserción. Las búsquedas se cortan cuando la clave que se está buscando, tanto vértice como arista, no existe porque la clave actual es mayor o se ha encontrado, por lo que no se recorren las listas completamente, solo en el peor de los casos.

4. Diagrama de clases

El diagrama de clases asociado puede consultarse en el anexo de figuras [4].

5. Uso

Para empezar a trabajar con el paquete, debemos importarlo al proyecto con la sentencia: **import sngraph.node.Sngraph;**
A partir de aquí tendremos que crear un objeto de la clase Sngraph, si nos fijamos no hay que establecer a priori el número de nodos por lo explicado anteriormente. En el código que se expone a continuación se muestra cómo insertar nodos y cómo insertar aristas con peso entre nodos. Hay que destacar que cuando insertamos una arista, podemos indicar que si existe la arista, añada el nuevo peso al ya existente *true* y, por el contrario, si el valor es *false* que inserte el nuevo valor sin tener en cuenta el anterior.

```
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));
```

que da como resultado el siguiente grafo expuesto en la figura [2] (prestar atención al grosor de las aristas)
En la línea número 14 (**System.out.println(g.toGraphviz());**) , podemos observar cómo podemos obtener el código del grafo para generar el grafo de la figura [2]

```
1 digraph Grafo{
2  1->8 [label="2.3" style="setlinewidth(2.3)"];
3  1->9 [label="1.72" style="setlinewidth(1.72)"];
4  8->9 [label="4.002" style="setlinewidth(4.002)"];
5  8->4 [label="0.4" style="setlinewidth(0.4)"];
6 }
```

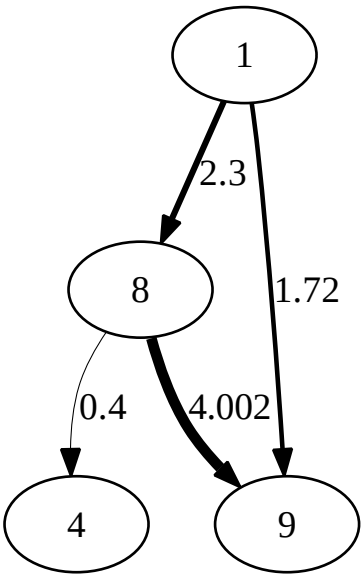

Figura 2.
Ejemplo 1

para realizar la exportación a Ucinet, debemos utilizar el método de la línea 18 (**System.out.println(g.toUcinet(null));**) , y obtendríamos:

```
1 dl n = 4 format = edgelist1
2 data:
3 1 2 2.3
4 1 3 1.72
5 2 3 4.002
6 2 4 0.4
```

cabe destacar que el grafo que genera para ucinet requiere de un vector con los identificadores asociados a los nodos que se insertaron en orden. Esto es $1 \rightarrow 1, 2 \rightarrow 8, 3 \rightarrow 9, 4 \rightarrow 4$. En el siguiente código se muestra la creación del vector de etiquetas *v* y la generación de código para ucinet.

```
1 Vector<String> labels = new Vector<String>();
2
3 labels.add("1");
4 labels.add("8");
5 labels.add("9");
6 labels.add("4");
7
8 System.out.println("");
9 System.out.println(g.toUcinet(labels));
```

la salida por pantalla sería la siguiente:

```
1 dl n = 4 format = edgelist1
2 labels:
3 1,8,9,4
4 data:
5 1 2 2.3
6 1 3 1.72
```

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

y la representación en Ucinet de este código es la mostrada a continuación en la figura [3]:

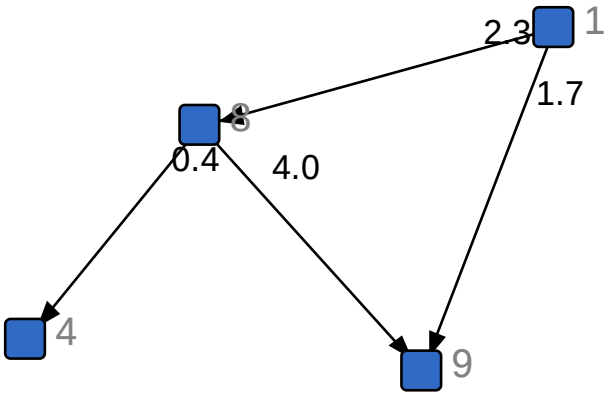

Figura 3.
Red generada para ucinet

6. Ejemplo completo

En el ejemplo propuesto, se ha programado un algoritmo de *backtracking* [Knuth, 1968] que irá mostrando todas las rutas posibles entre dos nodos dados del grafo. Se muestra el código, el grafo generado de manera visual por Graphivz y la salida de la ejecución.

```
1
2  import java.util.Vector;
3  import sngraph.node.Sngraph;
4
5  /*
6   *  Copyright 2009-2010
7   *  Consejo Superior de Investigaciones Cientificas
8   *  Centro de Ciencias Humanas y Sociales
9   *  Unidad de sistemas de informacion geografica
10  *  Attribution-Noncommercial 3.0 Unported - http://creativecommons.org/
        licenses/by-nc/3.0/
11  */
12 /**
13  *
14  * @author Roberto Maestre Martinez <roberto.maestre AT cchs.csic.es>
15  */
16 public class Uso1 {
17
18
19     //Indica si un nodo esta en el vector de visitados
20     private static boolean isVisitado(int i, Vector<Integer> visitados) {
21         boolean cent=false;
22         int c=0;
```

```
23          while (c<visitados.size() && !cent){
24              if (i==visitados.get(c)) cent=true;
25              c++;
26          }
27          return cent;
28      }
29
30      //Procedimiento para calcular el camino mas corto entre dos nodos
31      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){
32          //Compruebo mejor solucion
33          if (actual==hasta && total<mejortotal.get(0)){
34                      System.out.println("mejor encontrado, conste="+total);
35                      System.out.println(camino);
36                      //Copio el mejor total y el mejor camino
37                      mejorcamino = new Vector<Integer>();
38                      for (int x=0;x<camino.size();x++)
39                          mejorcamino.add(camino.get(x));
40                      mejortotal.add(0, total);
41          } else {
42              //Calculo los nodos a los que puedo ir desde el nodo actual
43              Vector<Integer> a=new Vector<Integer>();
44              for (int i=0;i<N;i++){
45                  if ((g.getWeightBetweenNodes(actual, g.getNodeList().get(i))
                        !=g.getInfiniteValue()) && !isVisitado(g.getNodeList().
                        get(i),visitados))
46                      a.add(g.getNodeList().get(i));
47              }
48              //Si hay nodos a los que pueda ir
49              if (a.size()>0){
50                  //Los recorro
51                  for (int i=0;i<a.size();i++){
52                      //Anoto
53                      total=total+g.getWeightBetweenNodes(actual, a.get(i));
54                      visitados.add(actual);
55                      camino.add(a.get(i));
56                      //Llamada
57                      backtracking(g, visitados, camino, mejorcamino, a.get(i)
                            , hasta, N, total, mejortotal);
58                      //Desanoto
59                      total=total-g.getWeightBetweenNodes(actual, a.get(i));
60                      visitados.remove(visitados.size()-1);
61                      camino.remove(camino.size()-1);
62                  }
63              }
64          }
65      }
66
67
68      public static void main (String args[]) throws Exception{
69          Sngraph g = new Sngraph();
70          g.insertNode(1); g.insertNode(8); g.insertNode(9);g.insertNode(12)
                ;g.insertNode(16);g.insertNode(24);g.insertNode(50);
71          g.insertNode(69);g.insertNode(70);g.insertNode(76);g.insertNode
                (79); g.insertNode(80); g.insertNode(89);g.insertNode(91);
```

```
72          g.insertNode(93);g.insertNode(94);g.insertNode(95);g.insertNode
                (100);g.insertNode(101);g.insertNode(103);g.insertNode(106);
73          g.insertNode(110); g.insertNode(115);g.insertNode(122);g.
                insertNode(129);g.insertNode(135);g.insertNode(140);g.
                insertNode(169);
74          g.insertNode(170);g.insertNode(176);
75
76          g.insertWeightBetweenNodes(1, 8, 2.3, true);
77          g.insertWeightBetweenNodes(1, 16, 1.72, true);
78          g.insertWeightBetweenNodes(9, 12, 1.0, true);
79          g.insertWeightBetweenNodes(16, 24, 3.2, true);
80          g.insertWeightBetweenNodes(24, 50, 1.05, true);
81          g.insertWeightBetweenNodes(50, 9, 3.2, true);
82          g.insertWeightBetweenNodes(9, 8, 2.05, true);
83          g.insertWeightBetweenNodes(12, 16, 2.72, true);
84          g.insertWeightBetweenNodes(94, 122, 1.72, true);
85          g.insertWeightBetweenNodes(95, 100, 1.02, true);
86          g.insertWeightBetweenNodes(94, 95, 2, true);
87          g.insertWeightBetweenNodes(100, 1, 3.56, true);
88          g.insertWeightBetweenNodes(69, 8, 1.26, true);
89          g.insertWeightBetweenNodes(170, 176, 0.26, true);
90          g.insertWeightBetweenNodes(110, 129, 1.64, true);
91          g.insertWeightBetweenNodes(129, 12, 2.26, true);
92          g.insertWeightBetweenNodes(176, 50, 0.76, true);
93          g.insertWeightBetweenNodes(89, 176, 0.06, true);
94          g.insertWeightBetweenNodes(16, 122, 0.32, true);
95          g.insertWeightBetweenNodes(140, 169, 1.32, true);
96          g.insertWeightBetweenNodes(129, 135, 2.32, true);
97          g.insertWeightBetweenNodes(140, 140, 0.12, true);
98          g.insertWeightBetweenNodes(140, 94, 0.16, true);
99          g.insertWeightBetweenNodes(94, 140, 0.16, true);
100         g.insertWeightBetweenNodes(95, 100, 0.16, true);
101         g.insertWeightBetweenNodes(101, 176, 0.16, true);
102         g.insertWeightBetweenNodes(24, 95, 0.16, true);
103         g.insertWeightBetweenNodes(100, 110, 1.16, true);
104         g.insertWeightBetweenNodes(89, 101, 0.26, true);
105         g.insertWeightBetweenNodes(89, 170, 1.3, true);
106         g.insertWeightBetweenNodes(16, 89, 0.4, true);
107         g.insertWeightBetweenNodes(110, 169, 0.54, true);
108         g.insertWeightBetweenNodes(169, 69, 2.264, true);
109
110         System.out.println(g.toGraphviz());
111         System.out.println("");
112         Vector<String> labels = new Vector<String>();
113         labels.add("1"); labels.add("8"); labels.add("9");labels.add("12")
                ;labels.add("16");labels.add("24");labels.add("50");
114         labels.add("69");labels.add("70");labels.add("76");labels.add("79"
                ); labels.add("80"); labels.add("89");labels.add("91");
115         labels.add("93");labels.add("94");labels.add("95");labels.add("100
                ");labels.add("101");labels.add("103");labels.add("106");
116         labels.add("110"); labels.add("115");labels.add("122");labels.add(
                "129");labels.add("135");labels.add("140");labels.add("169");
117         labels.add("170");labels.add("176");
118         System.out.println(g.toUcinet(labels));
119         System.out.println("");
120         g.print();
```

```
121             System.out.println("");
122
123             Vector<Double> mejortotal=new Vector<Double>();
124             mejortotal.add(Double.MAX_VALUE);
125             Vector<Integer> visitados=new Vector<Integer>(),camino=new Vector<
                    Integer>(),mejorcamino=new Vector<Integer>();
126             double total=0.0;
127             backtracking(g, visitados, camino, mejorcamino, 1,50, g.
                    getNodeList().size(), total,mejortotal);
128             System.out.println(mejorcamino);
129
130         }
131
132
133
134
135 }
```

la salida de la ejecución del código anteriormente consta de varias partes:

1. Número de veces que se redistribuye el índice de la tabla hash
2. Salida para Graphivz
3. Salida para Ucinet
4. Representación de la estructura interna
5. Camino mínimos obtenidos por el *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]
```

el grafo generado para Graphivz puede verse en la figura [5] y para ucinet en la figura [6].

7. Trabajo futuro

1. Generación de capas para SIG con redes georeferenciadas.

8. Copyrigth

9. Como citar

... se construyó [Maestre-Martinez, 2010] la red de ejemplo utilizando los pesos de parentesco y relaciones comerciales ...

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

10. Anexo Figuras

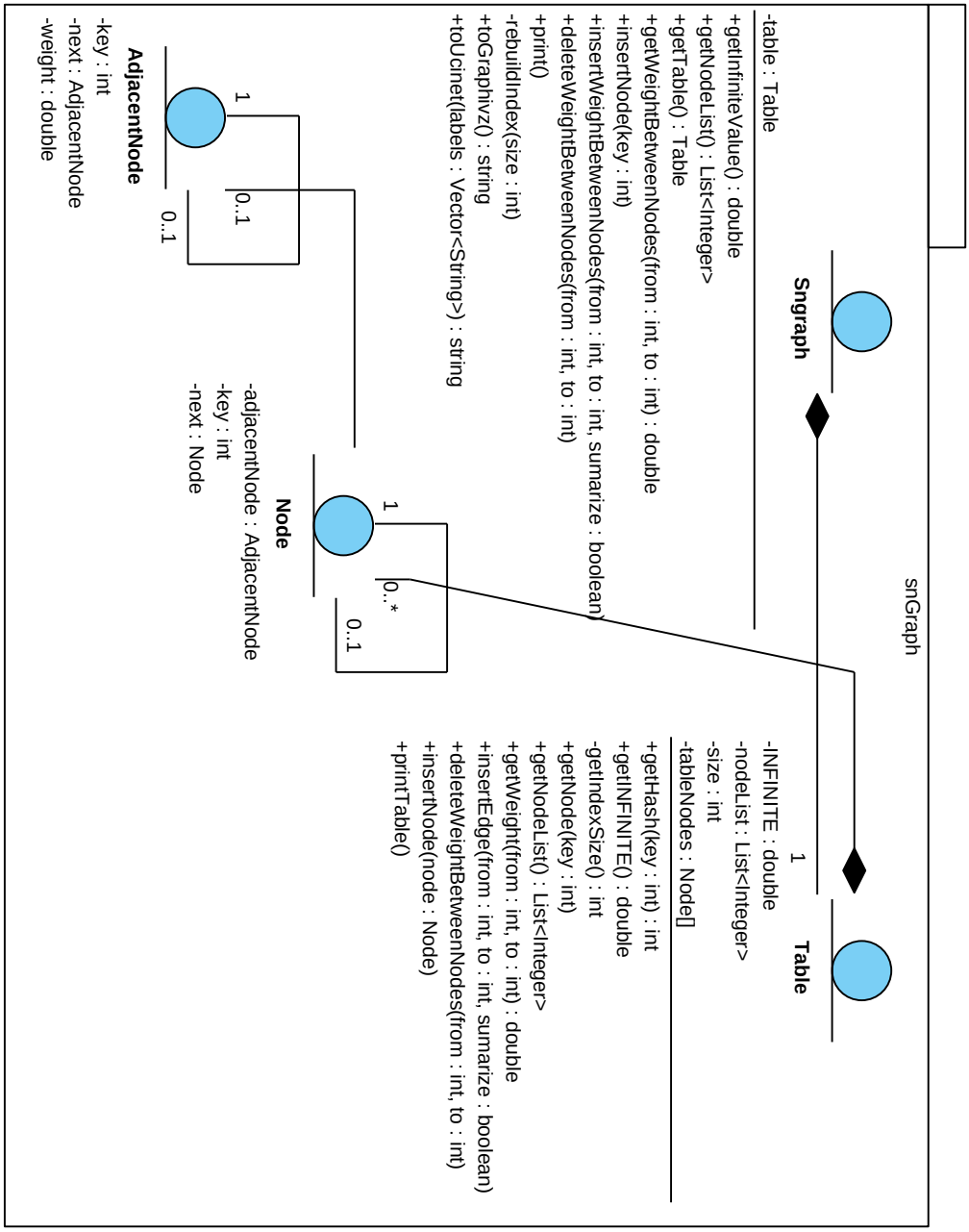

Figura 4.
Diagrama de clases

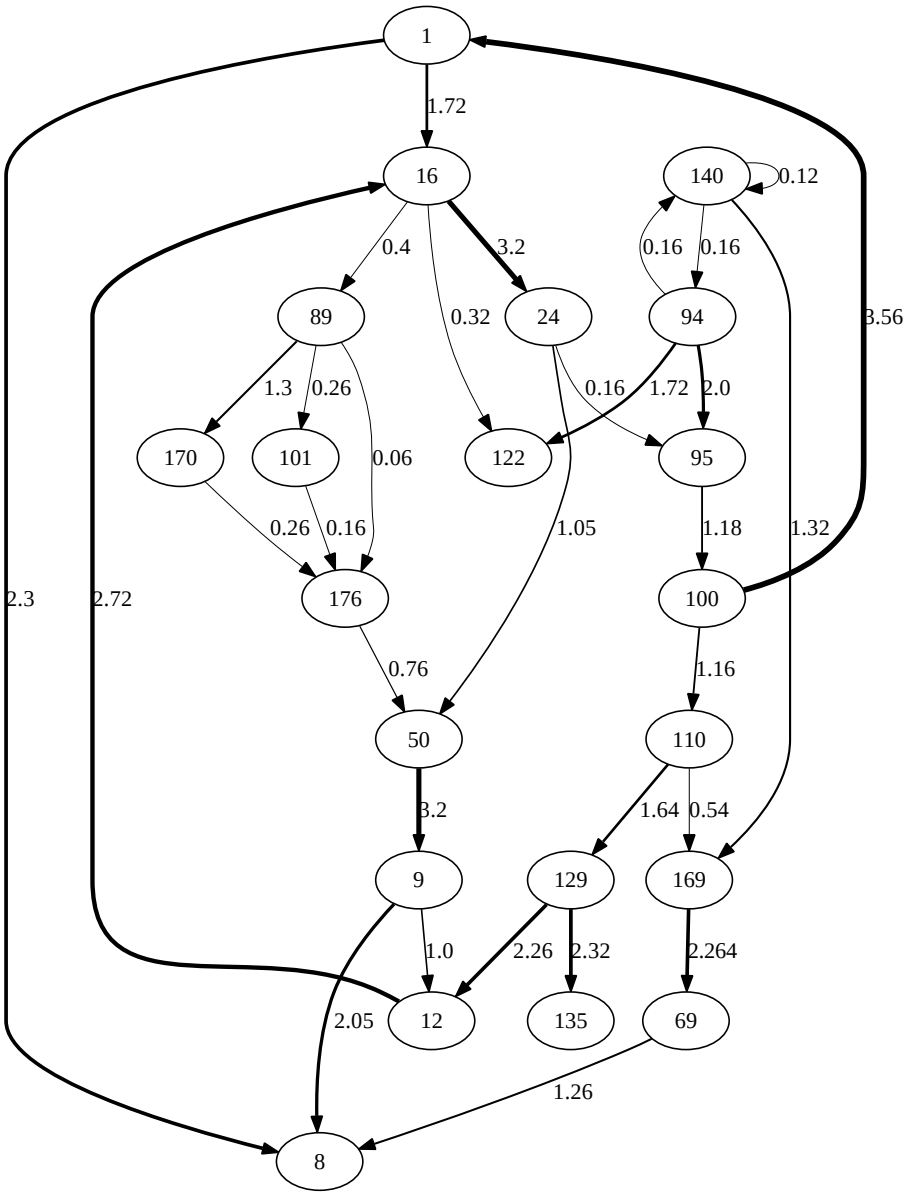

Figura 5.
Ejemplo grafo Graphivz

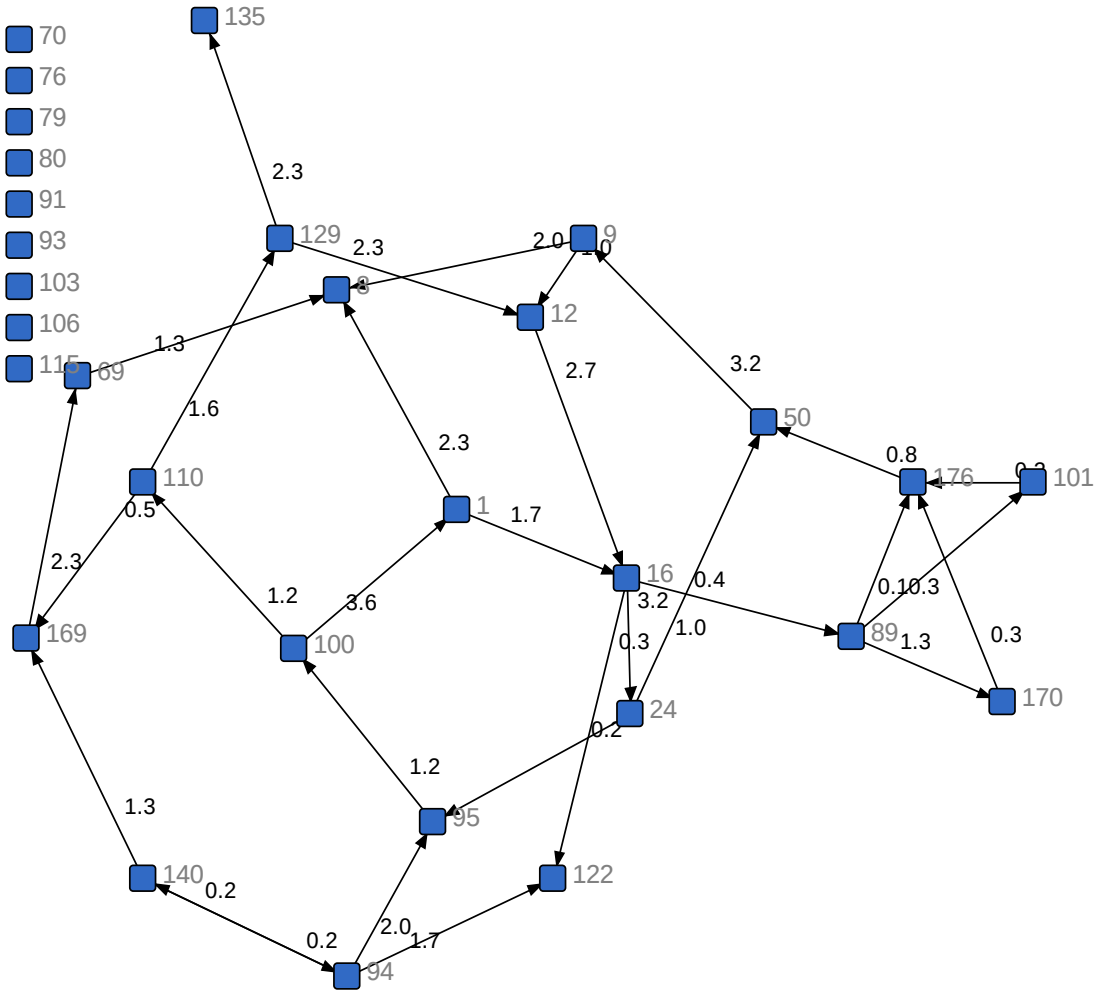

Figura 6.
Ejemplo grafo ucinet

Referencias

[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.