Universidad Nacional de General Sarmiento
Instituto de Industria

Trabajo practico n°3

Programación III

Comisión: 2.

Docentes: Alejandro Nelis, Fernando Torres.

Alumnos: Chamorro Franco, Montenegro Tomas, Romano Facundo, Sandoval Maximiliano

INDICE

Introducción

El siguiente informe explica la composición y organización de cómo se realizó el tercer y último trabajo practico de la materia Programación III, en el cual implementamos diferentes y nuevos elementos que tuvimos que aprender a utilizar y contemplar para poder cumplir con los requerimientos pedidos por parte del enunciado de dicho trabajo practico.

La idea principal del trabajo practico es implementar un algoritmo goloso que resuelva el problema de la clique máxima, y para resolver esto, pensamos en hacer una heurística que ordene los nodos del grafo, que ingresa el usuario, de mayor a menor peso. Una vez ordenados los mismos, vamos agarrando el nodo de mayor peso con su vecino de más peso, buscamos los vecinos que tienen en común y también agarramos el que más peso tiene (todo este procedimiento lo realiza de forma recursiva). También, se agregará otra heurística que ordene otra vez los nodos de mayor a menor grado y de las dos heurísticas que tenemos, nos quedamos con la de mayor peso.

Comenzamos fraccionando el proyecto en 3 diferentes packages, que son “Interfaz”, “Negocio” y “Test”. Esto nos permitió desde un principio ser más organizados a la hora de realizar el diagrama de clases, la cual suministramos de forma opulenta.

Package “Negocio”

El package “Negocio” creamos la clase “Arista”, “Clique”, “ComparadorPorGrado”, “ComparadorPorPeso”, “Grafo”, “Nodo”, “SolverGoloso” y “problemaCliqueMaxima”, que contienen el código que le otorga la funcionalidad al proyecto.

CLASE “SolverGolos”:

En la clase “SolverGoloso” tenemos los métodos para trabajar con los nodos que ingresa el usuario. El método “resolver”:

```
1. public Clique resolver() {
2.                 comenzarContarTiempo();
3.                 filtrar();
4.                 ordenarNodosPorPeso(this._listaNodosTienenVecinos);
```

```
5.                 for (int i = 0; i < this._listaNodosTienenVecinos.size();
    i++) {
6.
7.                         Nodo
   nodoMasPesado = this._listaNodosTienenVecinos.get(0);
8.                         this._listaNodosTienenVecinos.remove(0);
9.                         _maximaClique.agregarNodoAClique(nodoMasPesado,
   _grafo);
10.                                _grafo.ordenarVecinos(nodoMasPesado,
   _comparador);
11.                                Nodo
   vecinoMasPesado = (_grafo.obtenerPrimerVecino(nodoMasPesado));
12.                                _grafo.ordenarVecinos(vecinoMasPesado,
   _comparador);
13.                                _maximaClique.agregarNodoAClique(vecinoMasP
   esado, _grafo);
14.                                this._listaNodosTienenVecinos.remove(vecino
   MasPesado);
15.                                ArrayList<Nodo> listaEnComun = interseccion
   (_grafo.obtenerVecinos(nodoMasPesado),
16.                                                _grafo.obtenerVecinos(vecin
   oMasPesado));
17.                                BuscarClique(listaEnComun);
18.                        }
19.                        terminarContarTiempo();
20.
21.                        return _maximaClique;
22.                }
23.
```

La función de este método es capturar los nodos del grafo ingresado por el usuario y guardar los que tienen vecinos, ya que los que no tienen vecinos no nos sirven. Una vez que los agrega a la clique que vamos a resolver como solución, los recorre, agarra de cada nodo el vecino que más peso tiene y genera una lista de intersección entre el nodo y su vecino con mayor peso.

Esa lista de intersección la pasa al método “buscarClique” que realiza un algoritmo recursivo. Ese algoritmo corta cuando la lista de intersección ese vacío (que no tenga más vecinos en común).

```
1.         private void BuscarClique(ArrayList<Nodo> listaInterseccion) {
2.                 while (!listaInterseccion.isEmpty()) {
3.                        ordenarNodosPorPeso(listaInterseccion);
4.     _maximaClique.agregarNodoAClique(listaInterseccion.get(0), _grafo);
5.                         listaInterseccion.remove(0);
6.                         if (listaInterseccion.isEmpty()) {
7.                                 return;
8.                         } else {
9.                 ArrayList<Nodo> nueva = interseccion(listaInterseccion,
10.                _grafo.obtenerVecinos(listaInterseccion.get(0)));
11.                                BuscarClique(nueva);
12.                        }
13.                }
14.        }
```

□ CLASE “ProblemaCliqueMaxima”:

La clase “problemaCliqueMaxima” se encarga de instanciar dos heurísticas para resolver el problema de clique de mayor peso. Creamos 2 “SolverGoloso” donde uno usa un comparadorPorPeso y otro un comparadorPorGrado. De ello obtenemos 2 soluciones, una donde los nodos fueron ordenados de mayor peso a menor y otra donde los nodos fueron ordenados de mayor grado a menor.

```
1. public void hallarCliqueMaxima() {
2.                 _porCompararPeso = _solucionPorPeso.resolver();
3.                 _porCompararGrado = _solucionPorGrado.resolver();
4.                 _cliqueConMasPeso = _porCompararPeso.cliqueConMasPeso(_po
   rCompararGrado);
5.         }
```

El método hallarCliqueMaxima se queda con la solución (clique) que más peso tenga.

□ CLASE “Arista”:

La clase “Arista” contiene las propiedades del objeto, que para poder llevar acabo la idea principal, implementamos una arista que conecta dos nodos:

```
1. public Arista(int x1, int y1, int x2, int y2, String nombre) {
2.                 super();
3.                 this.x1 = x1;
4.                 this.y1 = y1;
5.                 this.x2 = x2;
6.                 this.y2 = y2;
7.                 this.setNombre(nombre);
8.         }
```

Dicho objeto recibe una posición en el eje cartesiano y también se le puede poner un nombre a la arista.

□ CLASE “GRAFO”:

Sobre la clase “Grafo” para representar adecuadamente los grafos, utilizamos métodos específicos para su funcionamiento. Uno de los métodos más importantes de esta clase es “ordenarVecinos”:

```
1. public void ordenarVecinos(Nodo nodo, Comparator<Nodo> comparador) {
2.                 if (tieneVecinos(nodo)) {
3.                         Collections.sort(getindiceConVecinos().get(nodo.g
   etIndiceNodo()), comparador);
4.                 }
```

En este método recibe un nodo, un comparador y ordena los vecinos usando el mismo para especificar que los queremos ordenar por peso y por grado, hicimos dos clases distintas, cada una de estas implementan un comparador y sobrescribe el método “compare”.

CLASE “Clique”:

Dentro de la clase “Calculo” vamos a representar en donde almacenamos el resultado de los notos. El método fundamental de esta clase es el método “agregarNodoClique”:

```
1. public void agregarNodoAClique(Nodo nodo, Grafo grafo) {
2.
3.                 if (!_listaNodo.contains(nodo) && esVecinodeTodos(nodo,
   grafo)) {
4.                         _listaNodo.add(nodo);
5.                         this._peso(this.getPeso() + nodo.getPeso());
6.                 }
7.         }
```

Este método lo que hace es comprobar, cuando agregamos un nodo a la clique, que los otros nodos sean vecinos del que agregamos. Este método llama a “esVecinodeTodos”:

```
1. private boolean esVecinodeTodos(Nodo nodo, Grafo grafo) {
2.                 boolean ret = true;
3.                 for (Nodo iteracion : _listaNodo) {
4.                         ret = ret && grafo.getindiceConVecinos().get(iter
   acion.getIndiceNodo()).contains(nodo);
5.                 }
6.                 return ret;
7.         }
```

Que comprobara, mediante un acumulador booleano, que el nodo que se pasa por parámetro es vecino de los que están dentro de la lista que se recorre.

CLASE “ComparadorPorPeso”:

La clase “ComparadorPorPeso” posee solo un método que implementa un comparador y sobre escribe el método “compare” en este caso lo usaremos para ordenar por peso, el cual retorna 0

si los nodos pasados por parámetro tienen igual peso, retorna -1 si es el peso es mayor y en caso de que no sean iguales o ni sea mayor, retorna 1:

```
1. public class ComparadorPorPeso implements Comparator<Nodo> {
2.
3.         @Override
4.         public int compare(Nodo uno, Nodo otro) {
5.                 if (uno.getPeso() == otro.getPeso())
6.                         return 0;
7.                 if (uno.getPeso() > otro.getPeso())
8.                         return -1;
9.                 else
10.                                 return 1;
11.             }
12.
13.     }
```

□ CLASE “ComparadorPorGrado”:

Al igual que la clase “ComparadorPorPeso”, esta clase “ComparadorPorGrado” también posee solo un método que implementa un comparador y sobre escribe el método “compare” pero en este caso lo usaremos para ordenar por grado, el cual retorna 0 si los nodos pasados por parámetro tienen igual cantidad de vecinos, retorna -1 si es mayor la cantidad y en caso de que no sean iguales o ni sea mayor, retorna 1:

```
1. public class ComparadorPorGrado implements Comparator<Nodo> {
2.
3.         @Override
4.         public int compare(Nodo uno, Nodo otro) {
5.                 if (uno.getCantidadVecinos() == otro.getCantidadVecinos()
   )
6.                         return 0;
7.                 if (uno.getCantidadVecinos() > otro.getCantidadVecinos())
8.                         return -1;
9.                 else
10.                                 return 1;
11.             }
```

□ CLASE “Nodo”:

La clase “Nodo” básicamente representara al objeto en sí y lo creara con una posición en x e y sobre un eje cartesiano, con un respectivo peso, diámetro, su índice para identificarlo y cantidad de vecinos:

```
1. public Nodo(int indice_nodo,int x, int y,int peso) {
2.                 super();
3.                 this.x = x;
4.                 this.y = y;
5.                 this._indice_nodo = indice_nodo;
6.                 this.setPeso(peso);
7.                 this._cantidadVecinos = 0;
8.         }
```

Package “Interfaz”

En el package “Interfaz” creamos las clases “Dibujo”, “Principal_TP3”, “TextoTransluciodo”, “VistaBienvenida” y “VistaGrafo” que contiene todo el código implementado para brindar la mecánica que dispone Windowsbuilder. Por un lado, trabajamos con objetos graficos, que son objetos primitivos de Java y a partir de ahí heredamos el jFrame.

□ CLASE “VistaBienvenida”:

La clase “VistaBienvenida” funciona como una ventana estilo JFrame y capturamos el input numérico del usuario en el método “inicializarTextfieldCantNodosCrear”:

```
1. private void inicializarTextfieldCantNodosCrear() {
2.                 _tFCantNodos = new JTextField();
3.                 setTextoTranslucido(new TextoTranslucido("NUMERO MAX DE
   NODOS A CREAR", _tFCantNodos));
4.                 _tFCantNodos.setBounds(347, 515, 343, 52);
5.                 _ventanaBienvenida.getContentPane().add(tFCantNodos);
6.                 _tFCantNodos.setColumns(10);
7.                 limitarInputUsuario(_tFCantNodos);
8.         }
```

Y limita al usuario para escribir hasta 7 digitos en dicho textField en el método “limitarInputUsuario” (el campo no puede estar vacio):

```
1. private void limitarInputUsuario(JTextField tf) {
2.                 tf.addKeyListener(new KeyAdapter() {
3.                         public void keyTyped(KeyEvent e) {
```

```
4.                                  if((e.getKeyChar() < '0') || (e.getKeyCha
   r() > '9') && (e.getKeyChar() != '\b')){
5.                                          e.consume();
6.                                  } else {
7.                                          _btnEnviar.setEnabled(true);
8.                                          if (tf.getText().length() >= 7)
9.                                                  e.consume();
10.                                     }
```

Cuando accionamos el botón enviar, se captura esa información y la almacenamos en una variable para pasársela a los métodos que lo necesitan. Todo esto se realiza en el método "accionBotonEnviar":

```
1. private void accionBotonEnviar() {
2.                 _btnEnviar.addActionListener(new ActionListener() {
3.                         public void actionPerformed(ActionEvent e) {
4.                                 if (!_tFCantNodos.getText().toString().eq
   uals("") && !_tFCantNodos.getText().toString().equals("1") && !_tFCantNod
   os.getText().toString().equals("0")) {
5.                                         _cantidadNodosLimite = Integer.pa
   rseInt(_tFCantNodos.getText().toString());
6.                                         cambiarVentana();
7.                                 } else
8.                                         JOptionPane.showMessageDialog(nul
   l, "INGRESE UN VALOR NUMERICO >1 PARA CONTINUAR");
9.                         }
10.                    });
11.             }
```

▯ CLASE "Dibujo":

La clase "Dibujo" está muy conectado con lo que es el método "Paint" de la clase "VistaGrafo", que trabajamos pura y exclusivamente con objetos de tipo gráficos (primitivos de java). En la clase "Dibujo" no trabajamos con componentes. Tenemos el método "pintarEnlace":

```
1. public static void pintarEnlance(Graphics g, Arista e) {
2.                 g.drawLine(e.getX1(), e.getY1(), e.getX2(), e.getY2());
3.
4.                 if (e.getX1() > e.getX2() && e.getY1() > e.getY2()) {
5.                         g.drawString(e.getNombre(),
   e.getX1() - Math.abs((e.getX1() - e.getX2()) / 2),
6.                                         e.getY1() - Math.abs((e.getY1() -
    e.getY2()) / 2));
7.                 }
8.                 if (e.getX1() < e.getX2() && e.getY1() < e.getY2()) {
```

```
9.                          g.drawString(e.getNombre(),
   e.getX2() - Math.abs((e.getX1() - e.getX2()) / 2),
10.                                                 e.getY2() - Math.abs((e.get
   Y1() - e.getY2()) / 2));
11.                         }
12.
13.                         if (e.getX1() > e.getX2() && e.getY1() < e.getY2())
    {
14.                                 g.drawString(e.getNombre(),
   e.getX1() - Math.abs((e.getX1() - e.getX2()) / 2),
15.                                                 e.getY2() - Math.abs((e.get
   Y1() - e.getY2()) / 2));
16.                         }
17.
18.                         if (e.getX1() < e.getX2() && e.getY1() > e.getY2())
    {
19.                                 g.drawString(e.getNombre(),
   e.getX2() - Math.abs((e.getX1() - e.getX2()) / 2),
20.                                                 e.getY1() - Math.abs((e.get
   Y1() - e.getY2()) / 2));
21.                         }
22.                 }
```

Me dibuja el enlace, conecta nodo A con nodo B y realiza el respectivo dibujo (realiza la línea). Estas operaciones, se realizan para que la conexión quede bien alineada.

En el método “pintarNodo” de igual forma, es un objeto de tipo gráfico y lo importante que recibimos acá es el objeto gráfico, el nodo y un límite de X y un de Y. Principalmente dibuja un circulo que representa al nodo y por otra parte guardamos el peso y el índice correspondiente a cada nodo. El límite de X y de Y se utiliza para delimitar que dichos nodos no se puedan dibujar fuera del panel.

```
1. public static void pintarNodo(Graphics g, Nodo n,Color color) {
2.                          g.setColor(color);
3.                          g.drawOval(n.getX() - Nodo.diametroCirculo / 2,
   n.getY() - Nodo.diametroCirculo / 2,
4.                          Nodo.diametroCirculo,Nodo.diametroCirculo);
5.
6.                          if (n.getPeso() == null)
7.                                  g.drawString("", n.getX(), n.getY());
8.                          else
9.                                  g.drawString("P(" + n.getPeso() + ") //
   i:(" + n.getIndiceNodo() + ")",
10.                                         n.getX() - Nodo.diametroCirculo / 3
   , n.getY());
11.                         }
```

□ CLASE "VISTAGrafo":

En la clase "VistaGrafo" estamos trabajando con objetos graficos de tipo primitivo y para que nosotros podamos dibujar, tenemos que establecer una herencia desde JFrame. Una vez que heredamos del JFrame, uno de los métodos que nosotros heredamos es el método "paint":

```
1.  @Override
2.          public void paint(Graphics g) {
3.                  super.paint(g);
4.                  for (Nodo nodo : _vectorNodos) {
5.                          Dibujo.pintarNodo(g,nodo,_colorObjetos);
6.                  }
7.                  for (Arista enlaces : _vectorAristas) {
8.                          Dibujo.pintarEnlance(g, enlaces);
9.                  }
10.                       _colorObjetos = Color.black;
11.             }
```

Todo lo que dibujamos en pantalla esta en este método (sino lo está, no se dibuja).

Desde el método "mouseClicked" estamos heredando un evento del sistema que, básicamente dependiendo el botón del mouse que se oprima realizara un evento distinto. El clic izquierdo se utiliza para ubicar y dibujar los vértices y el clic derecho para establecer conexiones con aristas entre los mismos:

```
1.          @Override
2.          public void mouseClicked(MouseEvent e) {
3.                  if (_interruptorMouse) {
4.                          if (_panelDibujar.getWidth() > e.getX() && _panel
    Dibujar.getHeight() > e.getY()) {
5.                          if (e.getButton() == MouseEvent.BUTTON1) {// nodo
6.                                          String inputPesoNodo = JOptionPan
    e.showInputDialog("INGRESE UN PESO PARA EL NODO: ") + "0";
7.                                          if (inputPesoNodo != null && sonT
    odosNumeros(inputPesoNodo)) {
8.                                                  if (_indiceAuxCrearNodo +
     1 < VistaBienvenida._cantidadNodosLimite)
9.                                                          cargarNodo(e, Int
    eger.parseInt(inputPesoNodo));
10.                                                 else
11.                                                         seExcedioLi
    miteNodos();
12.                                         } else
13.                                                 JOptionPane.showMes
    sageDialog(null, "DEBE INGRESAR UN DATO NUMERICO PARA EL NODO");
14.                                 }
15.                                 if (e.getButton() == MouseEvent.BUT
    TON3)// arista
16.                                         accionClickDerecho(e);
17.                         }
```

```
18.                                 else
19.                                         e.consume();
20.                         }
21.                 }
```

Package “Test”

En el package “Test” creamos las clases para poder realizar los respectivos tests unitarios de las clases que se encuentran dentro del package “Negocio”. No profundizaremos demasiado en el informe sobre los tests ya que los mismos nos proveen de la información necesaria para entender el código (funcionan como documentación del proyecto). Pero igualmente utilizamos expresiones lambda como complemento de los mismos.