ADA - Práctica 1: Complejidad

El objetivo de esta práctica es que el alumno encuentre la complejidad de varias implementaciones del algoritmo de Dijkstra que busca los caminos mínimos en un grafo. El cálculo de esta complejidad se realizará a partir de los tiempos de ejecución de las diversas implementaciones con grafos de distinto tamaño, tanto en número de vértices como aristas.

Algoritmo de Dijsktra

Teniendo un grafo ponderado de N nodos, sea x el nodo inicial, un vector D de tamaño N guardará al final del algoritmo las distancias desde x al resto de los nodos.

1. Inicializar todas las distancias en D con un valor infinito relativo ya que son desconocidas al principio, exceptuando la de x que se debe colocar en 0 debido a que la distancia de x a x sería 0.
2. Sea a = x (tomamos a como nodo actual).
3. Recorremos todos los nodos adyacentes de a, excepto los nodos marcados, llamaremos a estos nodos no marcados vi.
4. Para el nodo actual, calculamos la distancia tentativa desde dicho nodo a sus vecinos con la siguiente fórmula: dt(vi) = Da + d(a,vi). Es decir, la distancia tentativa del nodo 'vi' es la distancia que actualmente tiene el nodo en el vector D más la distancia desde dicho el nodo 'a' (el actual) al nodo vi. Si la distancia tentativa es menor que la distancia almacenada en el vector, actualizamos el vector con esta distancia tentativa. Es decir: Si dt(vi) < Dvi → Dvi = dt(vi)
5. Marcamos como completo el nodo a.
6. Tomamos como próximo nodo actual el de menor valor en D que no se haya visitado y volvemos al paso 3 mientras existan nodos no marcados.

Una vez terminado al algoritmo, D estará completamente lleno con las distancias de x a cualquier otro nodo del grafo.

(Extraído de https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra)

Representación del Grafo

El grafo con el que trabajaremos será un grafo cuyos vértices son puntos del plano real. El peso de una arista que conecta dos vértices será la distancia euclídea entre los dos vértices.

Clases

Los puntos del plano se representan con la clase CPoint:

```
public class CPoint {
	public double m_X,m_Y;
	public CPoint(double x,double y);
	public CPoint Neg();
	public CPoint Add(CPoint b);
	public CPoint Sub(CPoint b);
```

```
        public double Product(CPoint b);
        public CPoint Div(double b);
        public double Module();
}
```

Esta clase implementa puntos del plano con coordenadas m_X y m_Y. Además implementa las operaciones aritméticas y módulo entre puntos.

La clase CVertex representa un vértice del grafo. Esta contiene el punto del plano donde se encuentra el vértice y una lista de los vértices vecinos. Además tiene los campos m_Distance y m_Visit necesarios para implementar el algoritmo de Dijkstra.

```
public class CVertex {
        public CPoint m_Point;
        public LinkedList<CVertex> m_Neighbords;
        public double m_Distance;
        public boolean m_Visit;
        public CVertex(double x, double y);
}
```

La clase CGraph representa todo el grafo como un array de vértices (m_Vertices) y nos da todas las funcionalidades necesarias para trabajar con el grafo.

```
public class CGraph {
        public ArrayList<CVertex> m_Vertices;
        public CGraph();
        public void Clear();
        // Vertices ------------------------------------------------------
        public CVertex FindVertex(double x,double y);
        public CVertex GetVertex(double x,double y) throws Exception;
        public CVertex GetVertex(CPoint p) throws Exception;
        public boolean MemberP(CVertex v);
        // Edges ---------------------------------------------------------
        public void Add(double x1, double y1, double x2, double y2);
        // Draw ----------------------------------------------------------
        public void Draw(Graphics g,double esc, Color c);
        public void AddRectHull(CPoint min, CPoint max);
        // Files ---------------------------------------------------------
        public void Write(String filename) throws Exception;
        public void Read(String filename) throws Exception;
        // Dijkstra ------------------------------------------------------
        public void Dijkstra1(CVertex start) throws Exception;
        public void Dijkstra2(CVertex start) throws Exception;
}
```

Para crear un grafo se utiliza el constructor y la función Add nos permite añadir dos puntos que forman una arista del grafo. Esta función ya se encarga de crear automáticamente los vértices y actualizar sus listas de vecinos.

Finalmente la clase CGraphView implementa una ventana donde se pueden visualizar los resultados. Esta ventana puede visualizar objetos CGraph. Es especialmente útil para ver los resultados del algoritmo que implementareis.

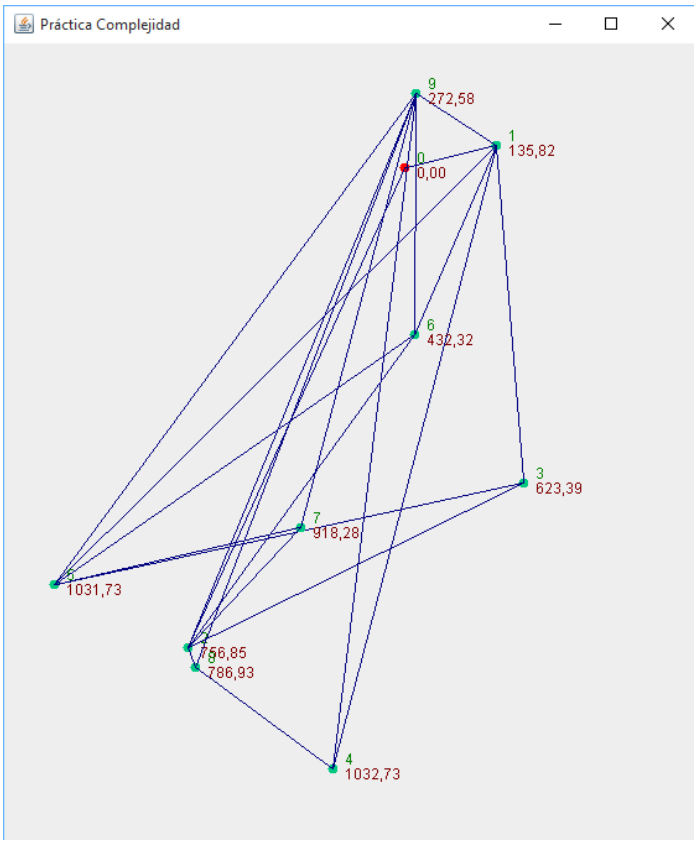
```
public class CGraphView extends JFrame {
```

```
    private static final long serialVersionUID = 1L;
    private static final int HEIGHT = 400;
    private static final int WIDTH = 700;
    private ArrayList<Object> m_Elements;
    private ArrayList<Color> m_Colors;
    public CGraphView();
    public void paint(Graphics g);
    public void ShowGraph(CGraph graph,Color color);
}
```

Material para la Práctica

En Cerbero encontrareis un fichero practica **Practica1_Complejidad.rar** que contiene un proyecto de Eclipse con todas las clases que necesitáis para hacer la práctica. El programa de este proyecto necesita uno o dos parámetros para ejecutarse. El primer parámetro es el fichero donde se guarda el grafo y el segundo parámetro indica que se ha de salir del programa si está presente. Por ejemplo, "CMain grafo1.txt" leerá el grafo1, luego visualizará el grafo y sobre el marcará las distancias calculadas. Además el programa imprimirá por la salida estándar la identificación de los alumnos, los ficheros leídos y las distancias de los vértices al primer vértice del grafo. Además tenéis en la clase CMain un generador de grafos aleatorios que puede servir para probar el programa.

Una salida posible es la siguiente donde las aristas del grafo son de color azul, los vértices de color verde excepto el inicial que es de color rojo. Los vértices está numerados a partir del 0 y además se puede ver la distancia de cada vértice al vértice 0.



La salida de texto del programa será la siguiente:

```
0000000
nombre del alumno 1
apellidos del alumno 1
0000000
nombre del alumno 2
apellidos del alumno 2
Fichero de grafo: Prob1_10_30_Graph.txt
Dijkstra1:
DISTANCES 0:0.0 1:135.82341477079717 2:756.849390565917 3:623.3857184818203
4:1032.7333407804938 5:1031.7302179596109 6:432.3162477364525
7:918.2777323705766 8:786.9326084788996 9:272.57501439581074
TIME: 0.004827005
Dijkstra2:
DISTANCES 0:0.0 1:135.82341477079717 2:756.849390565917 3:623.3857184818203
4:1032.7333407804938 5:1031.7302179596109 6:432.3162477364525
7:918.2777323705766 8:786.9326084788996 9:272.57501439581074
TIME: 7.78522E-4
```

En esta se ve la identificación del grupo, el fichero del grafo, las distancias que han calculado las dos implementaciones del algoritmo de Dijkstra y el tiempo que han tardado en segundos.

Entrega

Añadir a la clase CGraph el cuerpo de los métodos void Dijkstra1(CVertex start) y void Dijkstra2(CVertex start). Estos serán dos implementaciones del algoritmo de Dijkstra que calcula la distancia de todos los nodos del grafo al nodo start. La primera implementación realizará la búsqueda del nodo con menor distancia del paso 6 haciendo un simple bucle sobre los nodos del grafo. La segunda implementación utilizará una cola con prioridad para evitar esta búsqueda.

Una vez realizada esta implementación se puede probar con los ficheros *nombre*_Graph.txt y verificar que el resultado es correcto con las distancias guardadas en el fichero *nombre*_Result.txt. Así al ejecutar el programa se pondrá como parámetro *nombre*_Graph.txt y las distancias al primer vértice del grafo saldrán por pantalla y han de coincidir con las que hay en el fichero *nombre*_Result.txt. Si se quieren hacer más pruebas se puede utilizar la función void RandomProblem(int nVertices, int nEdges,String filename) que crea grafos aleatorios con nVertices y nEdges. Filename puede ser el nombre para que se grabe en disco el grafo y las distancias. En caso que no queremos que se creen ficheros Filename ha de ser null.

Con todo lo anterior se han de realizar un gran número de ejecuciones de las dos implementaciones del algoritmo de Dijkstra de las que se obtendrá los tiempos de ejecución. A partir de estos tiempos se tendrán que hacer gráficas donde el eje de abscisas sea número de vértices, número de aristas o una combinación y las ordenadas sean el tiempo de ejecución. Estas gráficas servirán para deducir la complejidad de las dos implementaciones respecto al tamaño del grafo. Además se tendrá que aproximar los resultados experimentales con las correspondientes funciones de complejidad que creamos que se adaptan mejor. Normalmente estas funciones serán las complejidades teóricas de los dos algoritmos.

El material proporcionado de la práctica lo encontrareis en Cerbero en un fichero Practica1_Complejidad.rar. Este contiene el proyecto para eclipse donde tendréis que añadir

el cuerpo de las funciones Dijkstra1 y Dijkstra2. Además se encontrarán los ficheros *nombre*_Graph.txt y *nombre*_Result.txt con los que podréis comprobar que vuestra práctica funciona correctamente. Es obligatorio que para cada grafo (*nombre*_Graph.txt) las distancias de los nodos del grafo calculados por las dos implementaciones del algoritmo de Dijkstra sean las que se proporcionan (*nombre*_Result.txt). Si no corresponden, **la práctica estará suspendida.**

La evaluación de la práctica la realizará el profesor con varios ficheros de grafos pensados para comprobar que el programa funciona correctamente. La salida del programa se utilizará para hacer la corrección, así que el programa no puede imprimir nada más por la salida estándar.

Cada práctica que se entregue estará identificada con los nombres y NIAs de los dos alumnos del grupo. En el fichero CMain.java hay que rellenar las siguientes variables para identificar los miembros del grupo que presentan la práctica:

```
static String NombreAlumno1="nombre del alumno 1";
static String ApellidosAlumno1="apellidos del alumno 1";
static String NIAAlumno1="0000000"; // NIA alumno1
static String NombreAlumno2="nombre del alumno 2";
static String ApellidosAlumno2="apellidos del alumno 2";
static String NIAAlumno2="0000000"; // NIA alumno2 o “” si un solo alumno
```

Esta información se utilizará para poner las notas de la práctica.

La entrega se realizará por Cerbero. La fecha máxima de entrega será el **9-10-2015**. El fichero a entregar será el fichero “Practica1_Complejidad.rar”. La estructura de directorios será la misma que hay en el fichero “Practica1_Complejidad.rar” del Cerbero que habéis utilizado para hacer la práctica. En el directorio bin se encontrarán los ficheros “.class” necesarios para ejecutar vuestra práctica y que resultan de compilarla con eclipse. Además en el directorio PracComplejidad tendréis que añadir un fichero pdf con las gráficas y las conclusiones sobre la complejidad de las dos implementaciones.

Consejos para hacer la práctica

- Para implementar el algoritmo de Dijkstra con cola jerárquica podéis utilizar la clase PriorityQueue de Java
- Para calcular los tiempos de ejecución hay que tener en cuenta el efecto del recolector de memoria de Java y el thread de interfaz de usuario.
- Los tiempos de ejecución no se ajustarán exactamente a las complejidades teóricas.
- Los grafos han de ser grandes para que los tiempos de ejecución sean realmente representativos.
- Hay que considerar la granularidad del reloj del ordenador.
- El ordenador no puede hacer nada más cuando se calculan tiempos para evitar que cualquier otro proceso nos afecte a las mediciones.
- Hay que implementar una función que genera los datos de los tiempos automáticamente. Así se podrán repetir los experimentos de forma más fácil, rápida y controlada.
- Se puede justificar la fórmula de la complejidad de las implementación por partes.