**PRACTICA 2b: Algoritmos Voraces**

- Algorítmica y Modelos de Computación -

Óscar Chaves Navarro

3º Grado en Ingeniería Informática.

# 1.- INTRODUCCIÓN

# 2.- ESTUDIO TEÓRICO

## ALGORITMO DE KRUSKAL

|  |
| --- |
| public ArrayList<Arista> AlgoritmoDeKruskal(Grafo g){    ArrayList<Arista> ConjuntoSolucion;  PriorityQueue<Arista> QueueCandidatos;  QueueCandidatos = new PriorityQueue();  ArrayList<Arista> ConjuntoCandidatos;  ConjuntoSolucion = new ArrayList<>();  ConjuntoCandidatos = g.getAristas();  //Crear una Cola de prioridad con las aristas ordenadas por distancia    for(int i = 0; i < ConjuntoCandidatos.size(); i++){  QueueCandidatos.add(ConjuntoCandidatos.get(i));  }    int n = g.getNumelementos();    //Inicializar, creamos un conjunto de conjuntos disjuntos de vertices    ArrayList<ArrayList> ConjuntoConjuntoVertice = new ArrayList<>();    for(int z = 0 ; z < g.getVertices().size() ; z++){  ArrayList<Vertice> ConjuntoVertices = new ArrayList<>();  ConjuntoVertices.add(g.getVertices().get(z));  ConjuntoConjuntoVertice.add(ConjuntoVertices);  }      //Bucle voraz    while(ConjuntoSolucion.size() < n-1){    //Extraemos la arista mas corta eliminandola de la cola.  Arista aux = QueueCandidatos.poll();  int i = 0 , j = 0;    //Comprobamos que los vertices de la arista pertenecena distintos  //conjuntos disjuntos.    while(!ConjuntoConjuntoVertice.get(i).contains(aux.getA()))  i++;    while(!ConjuntoConjuntoVertice.get(j).contains(aux.getB()))  j++;    if(i != j){  Fusionar(ConjuntoConjuntoVertice,i,j);  ConjuntoSolucion.add(aux);  }  }    return ConjuntoSolucion;  }    public void Fusionar(ArrayList <ArrayList> CCV, int i, int j){    int x = 0;    for(x = 0; x < CCV.get(i).size(); x++)  CCV.get(j).add(CCV.get(i).get(x));    CCV.remove(i);  } |

## ANALISIS TEÓRICO

Kruskal calcula el árbol expandido mínimo.

La cardinalidad de los vértices es **n** y la cardinalidad de las aristas del grafo es **m**

Ordenar A dentro de una cola de prioridad tiene un coste de **O(m\*logm)** al utilizar montículos.

Inicializar conjuntos disjuntos, cada uno con un vértice tiene coste de O(n)

2m buscar(peor caso) y n-1 fusionar(siempre): O(mlogn)

Resto O(m) (peor caso)

T(n) = O(m log n)

En nuestro caso, las aristas totales son prácticamente n2 al estar todos los vértices conectados entre si, por lo tanto la complejidad en el peor caso es:

**T(n) = O(n2 log n)**

## ALGORITMO DE PRIM

|  |
| --- |
| //////////////////////// ALGORITMO DE PRIM /////////////////////////////////    //Como todos los puntos en el grafo estan conectados con todos, contruiremos  //el algoritmo de un modo diferente (Sin matriz de adyacencia)  public ArrayList<Arista> AlgoritmoDePrim(Grafo graf){    ArrayList<Arista> ConjuntoSolucion;  Arista AristaAux;    int n = graf.getNumelementos();    //elegir punto de partida  Random rnd = new Random();  rnd.setSeed(System.currentTimeMillis());  int indicealeatorio = Math.abs(rnd.nextInt()%graf.getNumelementos());    Vertice verticeaux = graf.getVertices().get(indicealeatorio);  //Añadimos el vertice en una cola de prioridad ordenada por distancia    ArrayList<Vertice> VerticesVisitados = new ArrayList<>();    //desplegamos los adyacentes del vertice, añadimos al arbol la distancia minima  //añadimos el vertice al arbol  //movemos el puntero al vertice apuntado por la arista con distancia minima    ConjuntoSolucion = new ArrayList<>();    //si es solucion lo añado al conjunto solucion    while(ConjuntoSolucion.size() < n-1){    AristaAux = MasProximo(graf,verticeaux,VerticesVisitados);    ConjuntoSolucion.add(AristaAux);  VerticesVisitados.add(AristaAux.getB());  verticeaux = AristaAux.getB();    }    return ConjuntoSolucion;  }    public Arista MasProximo(Grafo graf, Vertice verticeaux, ArrayList<Vertice> VerticesVisitados){    int indice = graf.getVertices().indexOf(verticeaux);  double distancia\_minima = 1000000;  int indice\_final = 0;  for(int j = 0 ; j < graf.getVertices().size() ; j++){  if((graf.getMatrizDistancias()[indice][j] < distancia\_minima) &&  (!VerticesVisitados.contains(graf.getVertices().get(j))) &&  ( j != indice )){    indice\_final = j;  distancia\_minima = graf.getMatrizDistancias()[indice][j];    }  }    return (new Arista(verticeaux,graf.getVertices().get(indice\_final),graf.getMatrizDistancias()[indice][indice\_final]));    } |

## ANALISIS TEÓRICO

La cardinalidad de los nodos (Vertices) es n; la cardinalidad de las Aristas es m

Inicialización O(n)

Bucle voraz n-1 iteraciones y por cada iteración O(n) operaciones por lo tanto la complejidad total es:

**T(n) = O(n2)**

# 3.- Estudio empírico

## Comparativa de tiempos para el Algoritmo de Kruskal (Conjuntos de vértices aleatorios)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 200 | 400 | 600 | 800 | 1000 |
| Teórico |  |  |  |  |  |
| Experimental | 3215 us | 6250 us | 11337 us | 24448 us | 34024 us |

## Gráfica

## Comparativa de tiempos para el Algoritmo de Prim (Conjuntos de vértices aleatorios)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 200 | 400 | 600 | 800 | 1000 |
| Teórico |  |  |  |  |  |
| Experimental | 308 us | 906 us | 2125 us | 4556 us | 6060 us |

## Gráfica

## Comparativa de tiempos para el Algoritmo de Kruskal (Archivos del ch130, ch150 y berlin52)

|  |  |  |  |
| --- | --- | --- | --- |
|  | 52 | 130 | 150 |
| Teorico Kruskal | 4640,073041 | 35725,6427 | 48962,0533 |
| Experimental Kruskal | 1025 | 1295 | 2108 |

## Gráfica

## Comparativa de tiempos para el Algoritmo de Prim

## (Archivos del ch130, ch150 y berlin52)

|  |  |  |  |
| --- | --- | --- | --- |
|  | 52 | 130 | 150 |
| Teorico Prim | 2704 | 16900 | 22500 |
| Experimental Prim | 28 | 277 | 443 |

## Gráfica

## 4.- CONCLUSIONES

Las gráficas y los datos teóricos coinciden asintóticamente con los datos experimentales de los ejemplos.

En la práctica realizada hemos comprobado la eficiencia de los algoritmos voraces. En nuestro caso el algoritmo de Prim da resultados diferentes al de Kruskal ya que difiere algo en la implementación al aprovechar la conexión total de los vértices del grafo.

Esto provoca que haya mínimos locales al no poder hacer Backtracking.