Universidad de Buenos Aires
Facultad de Ingeniería

75.29 – Teoría de Algoritmos

Trabajo Práctico N° 3

2º Cuat. - 2016

Nombre	Padrón	Mail
Pablo Méndez	88908	pablo.23.32.23@gmail.com
Martín Lucero	89630	luceromartinandres89630@gmail.com
Julien Connaulte	100368	julien.connaulte@free.fr

Contenido

1. Introducción

Clases de complejidad

En teoría de la complejidad computacional, una clase de complejidad es un conjunto de problemas de decisión de complejidad relacionada.

Las clases de complejidad mas sencillas se definen teniendo en cuenta factores como:

- El tipo de problema computacional: Los problemas más comúnmente utilizados son los problemas de decisión, pero las clases de complejidad se pueden definir por otros tipos de problemas.
- El modelo de cómputo: El modelo de cómputo más común es la Máquina de Turing Determinista, pero muchas clases de complejidad se basan en Máquinas de Turing no Deterministas , Máquinas de Turing Cuánticas, etc.
- El recurso o recursos que están siendo acotados o las cotas: Estas dos propiedades usualmente se utilizan juntas; por ejemplo, “tiempo polinomial”, “espacio logarítmico”, “profundidad constante”, etc.

De entre las clases de complejidad existentes se pueden mencionar los siguientes ejemplos:

- Clase P: La clase P contiene a aquellos problemas que son solubles en tiempo polinómico por una máquina de Turing determinista.
- La clase NP: Esta clase de decisión contiene muchos problemas de búsqueda y optimización para los que se desea saber si existe cierta solución o una mejor solución que las conocidas.
- Otras clases son: DTIME, EXPTIME, NTIME, NEXPTIME, DSPACE, L, PSPACE, EXSPACE, NSPACE, NL, NPSPACE y NEXPSPACE.

Algoritmos de aproximación

Bajo el supuesto de que P es distinto de NP, los problemas NP-hard no tienen algoritmos polinomiales, por lo que un algoritmo que los resuelva en forma exacta puede tardar un tiempo prohibitivo. De manera que se deben utilizar algoritmos polinomiales que den soluciones aproximadas. Existen dos categorías de tales algoritmos: algoritmos de aproximación y algoritmos heuristicos. Un algoritmo de aproximación es un algoritmo que entrega una solución con una garantía teórica de cercanía al optimo. Mas aun, es frecuente que estos algoritmos posean un

desempeño practico muy superior a su garantía teórica. En las ultimas décadas, los algoritmos de aproximación han sido (y continúan siendo) un tema central de investigación en computación teórica y su aplicabilidad es cada vez mas evidente.

Los problemas NP-hard varían mucho en su aproximación; algunos, tales como el problema de la mochila, pueden ser aproximados mediante cualquier factor superior a 1 (tal familia de algoritmos de aproximación se conoce como esquema de aproximación de tiempo polinomial o PTAS). Otros, como el problema de la clique, son imposibles de aproximar dentro de cualquier constante, o incluso factor polinomiales, a menos que P = NP.

El presente trabajo se centrará en estudiar estos dos temas; analizando en un primer lugar, una serie de problemas en los cuales se intentará encontrar un algoritmo de orden polinomial que lo resuelva o en caso contrario demostrar que son NP-Completos y; en una segunda instancia, se abordara el problema de la aproximación presentando y analizando algoritmos que resuelvan; en forma aproximada, los problemas NP-Hard de la Mochila y del Viajante.

2. Clases de complejidad

A continuación se estudiarán una serie de problemas a los cuales se les intentará encontrar un algoritmo que los resuelva en tiempo pilinómico presentando; en cuyo caso, el pseudocódigo del mismo y en caso contrario se expondrá una demostración que demuestre su NP-Completitud.

2.1. Problema 1

Enunciado: Se tiene un grafo dirigido y pesado G, cuyas aristas tienen pesos que pueden ser negativos. Se pide devolver si el grafo tiene algún ciclo con peso negativo.

Resolución: Para la resolución de este problema se propone la utilización del Algoritmo de Floyd Warshall para; en una primera instancia, encontrar el camino mínimo entre todos los vértices del grafo y luego utilizar la siguiente condición para detectar el ciclo negativo: Si los números de la diagonal de la matriz de caminos resultante son negativos, es condición necesaria y suficiente para que este vértice pertenezca a un ciclo negativo. Este algoritmo corre en tiempo $O(V^3)$, siendo V la cantidad de vértices del grafo.

Ok

Bellman-Ford también sirve, con menor complejidad asintótica.

Inicio

Armar la matriz de adyacencia F, teniendo en cuenta que F(i,j)=0 si i = j (diagonal principal es 0). Además dónde no exista camino se debe indicar con infinito.

```
Para k desde 1 hasta n
    Para i desde 1 hasta n
        Para j desde 1 hasta n
            F[i,j]=min(F[i,j], F[i,k] + F[k,j])
        Fin para j
    Fin para i
Fin para k
Para i desde 1 hasta n
    Si F[i,i] < 0 entonces
        Imprimir "Existe ciclo negativo"
        break
```

Ver: "Exponentially large numbers in the Floyd-Warshall algorithm" en Hougardy (2010): The Floyd–Warshall algorithm on graphs with negative cycles

Fin si

Fin para i

Fin

2.2. Problema 2

Enunciado: Se tiene un grafo dirigido y pesado G, cuyas aristas tienen pesos que pueden ser negativos. Se pide devolver si el grafo tiene algún ciclo con peso exactamente igual a cero.

Resolución: Este problema es NP-Completo.

Versión de decisión:

ZeroWeightCycle: Dado un grafo G(E,V) dirigido y pesado cuyas aristas pueden tener ciclos negativos, existe algún ciclo en el grafo cuyo peso sea = 0?

Dem:

- Dada una instancia de un grafo dirigido y pesado con un ciclo simple, es posible construir un algoritmo que corra en tiempo polinomial que recorra el ciclo determine que la suma de los pesos de las aristas que lo componen es igual a cero. Por lo tanto el problema $\in$ NP.
- Para demostrar que este problema es NP-Completo se realizará una reducción a partir del SubsetSumProblem; el cual se define de la siguiente manera: Dado un conjunto de enteros determinar si existe un subconjunto no vacío de números cuya suma sea igual a cero.
- SubsetSumProblem $\leq_p$ ZeroWeightCycle: Dado un conjunto no vacío de enteros $S=\{a_1,...,a_n\}$, construir un grafo pesado G con 2n vértices tales que, por cada número a_i existen 2 vértices en el grafo, u_i y v_i. Para cada v_i agregar una arista de v_i a u_i con peso a_i y agregar una arista de peso cero a este último desde cada u_j. ~~Para cada u_i, agregar aristas desde este vértice a cada u_j con peso cero.~~ *Incorrecto; complica el gadget innecesariamente. (Revisen su fuente, les aseguro que no era esto lo que decía.)*
- $\Rightarrow$) Si tenemos un subset $S' \subseteq S$ los cuales suman cero, entonces construyendo un grafo como indica la reducción, produce un ciclo de peso cero que contiene a las aristas cuyos pesos son los valores a_i de S'. ✓
- $\Leftarrow$) Si tenemos una instancia de un grafo con un ciclo de peso cero, entonces la suma todos los pesos de las aristas entre v_i y u_i es igual a cero, por lo tanto la suma del subset de números a_i que son representados por las aristas del grafo es igual a cero.
- $\therefore$ ZeroWeightCycle $\in$ NP-Completo.

2.3. Problema 3

Enunciado: Se tiene un conjunto de n tareas, cada una con un tiempo de ejecución ti en R+, una fecha límite de finalización di en R+ y una ganancia vi en R+ que será otorgada si se finaliza antes que su tiempo límite. Se pide devolver si existe alguna planificación que obtenga una ganancia total mayor o igual a k en R+ sabiendo que no se pueden ejecutar dos tareas a la vez.

Resolución: Este problema es NP-Completo.

Versión de decisión:

SchedulingWithProfitsAndDeadlines:

Dados:

- p[1..n]: Beneficio obtenido por la realización de la tarea i.
- d[1..n]: Tiempo de finalización para la tarea i.
- t[1..n]: Tiempo de ejecución para la tarea i.
- una ganancia k en R+

Existe una programación de tareas que otorgue una ganancia de al menos k sin ejecutar dos tareas en simultaneo?

Dem:

- Dada una instancia del problema x=(p[1..n], d[1..n], t[1..n]) y una programación y=(s[1..n]); donde, la tarea i es programada en el tiempo s[i], se puede chequear en tiempo polinomial recorriendo la lista de tareas si las mismas no se solapan en tiempo y la suma de sus beneficios producen una ganancia de al menos k. Por lo tanto el problema $\in$ NP.
- Para demostrar que este problema es NP-Completo se realizará una reducción a partir del SubsetSumProblem, el cual se define de la siguiente manera: Dado un conjunto de valores números naturales y una constante k, determinar si existe un subconjunto no vacío de números cuya suma sea ~~de al menos una cantidad~~ k.
- SubsetSumProblem $\leq_p$ SchedulingWithProfitsAndDeadlines: Dada una instancia de SubsetSumProblem X = (x[1..n], t); donde, x[1..n] es el conjunto de números y t es el resultado de la suma del subset, es posible construir en tiempo polinomial una instancia de SchedulingWithProfitsAndDeadlines de la siguiente manera Y = (x[1..n], d[1..n], x[1..n], t), $\forall$i en d[i] = t.
- $\Rightarrow$) Dado un subconjunto de elementos X cuya suma sea igual a t, entonces es posible; con la reducción planteada, armar una programación Y de manera de obtener una ganancia t que equivale al valor total de la suma de subconjuntos. ?

- ⇐) Dada una programación Y, entonces para el subset S de tareas que cumplan con los plazos; esto es, $\Sigma_{i \text{ en } S} x_i \leq t$ (las tareas en S cumplen con el beneficio) y $\Sigma_{i \text{ en } S} x_i \geq t$ (restricción de beneficio), entonces $\Sigma_{i \text{ en } S} x_i = t$.

- ∴ SchedulingWithProfitsAndDeadlines ∈ NP-Completo.

2.4. Problema 4

Enunciado: Se tiene un conjunto de n tareas, cada una con un tiempo de ejecución igual a 1, una fecha límite de finalización di en N y una ganancia vi en R+ que será otorgada si se finaliza antes que su tiempo límite. Se pide devolver si existe alguna planificación que obtenga una ganancia total mayor o igual a k en R+ sabiendo que no se pueden ejecutar dos tareas a la vez.

Ver consigna...

Resolución: Para resolver el presente problema se propondrá un algoritmo greedy el cual corre en tiempo $O(n^2)$. *¿no O(n log n)?*

Sean S una programación que asigna cada slot de tiempo unitario 1, 2, …, n a un trabajo, d tal que S(t) representa al trabajo ejecutado en el instante de tiempo t y V(t) el arreglo de beneficio para la tarea t.

Ordenar los trabajos de mayor a menor según su beneficio v_i

Para t: 1 ..n

    S(t) <- 0 {Inicializar el array S(1), S(2), …, S(n) en cero}

Fin para

Para i: i ..n

    Para el trabajo i (el siguiente de mayor beneficio), asignarle el último slot libre posible sin

    que sobrepase su propio deadline.

    Si no existe tal slot, entonces no programar la tarea.

Fin para

Retornar Suma(V(S(i)))

3. Algoritmos de aproximación: El problema de la mochila

Para reducir el tiempo de ejecucion de la resolucion del problema de la mochila, podemos utilisar un algoritmo de aproximacion. La nueva relacion de recurrencia es :

$$P(i,v) = \begin{cases} \text{si } v > \sum_{i=1}^{i-1} v_i : & w_i + P(i-1, v-v_i) \\ \text{si no :} & \min \begin{bmatrix} P(i-1,v) \\ w_i + P(i-1, max\{0, v-v_i\}) \end{bmatrix} \end{cases}$$

Gracias a este algoritmo podemos encontrar una solucion aproximada con una error insignificante. El algoritmo fue probado con los archivos del TP2. Para cada problema calculamos la diferencia **d** entre el valor de solucion $\mathbf{v_s}$ y el valor de la aproximacion $\mathbf{v_a}$. Despuès calculamos la diferencia relativa

$$\mathbf{r = d / v_s}$$

Siguen las diferencias relativas encontradas para las 6 instancias dificiles del archivo, con n = 50.

N.º instancia	1	2	3	4	5	6
Diferencia relativa	0.006	0.0009	0.0002	0.0002	0.0001	0.0002

¿Con qué valor de epsilon?

Este algoritmo se ejecuta en O(n * v_{max}) con n el numero de objectos y v_{max} la suma d las valores de todos los objectos. Esta complexidad puede parecer peor que la del algoritmo original, que era O(n * w_{max}), pero el algoritmo de aproximacion ofrece la posibilidad de reducir el orden de las valores. Por ejemplo, si todas las valores tienen un tamano de 10^9 podemos cortarlas y considerarlas con un tamano de 10^3. La diferencia de resultado que vamos a obtener sera muy pequena y el programa sera mucho mas eficiente.

4. Algoritmos de aproximación: El problema del viajante de comercio

El segundo ejercicio para el cual utilizaremos un algoritmo de aproximación es el problema del viajante de comercio. A diferencia del tp de programación dinámica, esta vez deben cumplirse 2 condiciones (que igual se cumplían en los archivos tsp de los casos de prueba):

- Distancias simétricas
- Desigualdad triangular entre cada par de ciudades

Estas condiciones se cumplen en nuestros casos de prueba ya que fueron generados utilizando la distancia euclidiana como función de costo.

El problema del viajante es un problema NP-completo, por más que se cumpla la condición de la desigualdad triangular. No existe un algoritmo que busque la solución óptima en tiempo polinomial. Sin embargo existen algoritmos de **tiempo polinomial** que encuentras soluciones aproximadas.

4.1. Presentación del algoritmo

El algoritmo de aproximación utilizado es el propuesto en el libro "*Introduction to Algorithms*" de Thomas Cormen (y otros). Cormen explica que **si se cumple la condición de la desigualdad triangular, entonces el costo total calculado del tour no puede ser mayor a 2 veces el peso del MST, y a su vez el peso del MST es una cota inferior del costo óptimo del tour**.

"We shall first compute a structure—a minimum spanning tree—whose weight gives a lower bound on the length of an optimal traveling-salesman tour. We shall then use the minimum spanning tree to create a tour whose cost is no more than twice that of the minimum spanning tree's weight, as long as the cost function satisfies the triangle inequality."

APPROX-TSP-TOUR(G, c)

1 select a vertex $r \in G.V$ to be a "root" vertex
2 compute a minimum spanning tree T for G from root r
using MST-PRIM(G, c, r)
3 let H be a list of vertices, ordered according to when they are first visited
in a preorder tree walk of T
4 **return** the hamiltonian cycle H

Utilizaremos el grafo implementado en el TP 1, modificándolo levemente para hacerlo no dirigido y agregándole una primitiva para generar el MST con el algoritmo de Prim, y parte del código del TP 2.

4.2. Resultados y tiempos de ejecución

Utilizando los resultados del TP 2 calculados por programación dinámica, se comparará la efectividad de ambos algoritmos. (Aclaración: comparando el tp2 con otros grupos, ellos obtuvieron tiempos significativamente menores en el tsp de 15 ciudades. Desconocemos si había algo mal implementado en nuestro algoritmo de DP o si los otros grupos aplicaron optimizaciones)

Número de ciudades	Resultado DP	Resultado AlgoAprox	Sobre-costo (%)	Tiempo de ejecución DP	Tiempo de ejecución AlgoAprox
15	291	362	24	9113 seg	1 ms
26	937	1168	25	→∞	1 ms
42	699	868	24	→∞	2 ms

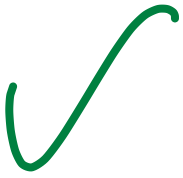
Las pruebas fueron realizadas con un Intel(R) Xeon(R) CPU E3-1220 V2 @ 3.10GHz y 24 GB RAM.

Como se puede observar, podemos obtener un resultado aproximado, con un costo del 24-25% por sobre el valor óptimo del tour, en tiempos ínfimos (para estos tsp de pocas ciudades y con una pc de escritorio estándar). Pensando el tsp a mayor escala, donde podemos hablar de miles de ciudades y utilizando supercomputadoras, creemos que tendríamos los mismos resultados: tiempos inmanejables para calcular la solución óptima vs tiempos manejables para resultados aproximados.

Quitando el parseo del tsp y el armado del grafo, tenemos:

- MST-Prim utilizando binary heap y lista de adyacencia: O(|E|log |V|)
- DFS del MST: O(|V| + |E|) con |E| = |V| - 1 (por prop árbol) => O(|V|)

Entonces tenemos que la complejidad del algoritmo es O(|E|log |V| + |V|)



5. Código fuente

5.1 El problema de la mochila

approximationKnapsack.java

```
public class approximationKnapsack{

	private int maxWeight;
	private int numberOfItem;
	private int[] values;
	private int[] weights;

	public int resultValue;
	public int resultWeight;

	public approximationKnapsack(int NumberOfItem, int MaxWeight, int[]
Values, int[] Weights){
		this.numberOfItem = NumberOfItem;
		this.maxWeight = MaxWeight;
		this.values = Values;
		this.weights = Weights;
		this.resultValue = runAlgorithm();
	}

	private int runAlgorithm(){
		int totalValueSum = 0;
		for (int i = 0; i < this.numberOfItem; i++)
			totalValueSum += this.values[i];
		int[][] M = new int[this.numberOfItem + 1][totalValueSum +
1];
		for (int i = 0; i <= this.numberOfItem; i++){
			M[i][0] = 0;
		}

		int partialValueSum = 0;
		for (int i = 1; i <= this.numberOfItem; i++){
			partialValueSum += this.values[i - 1];
			for (int v = 1; v <= totalValueSum; v++){
				if (v > partialValueSum - this.values[i - 1])
					M[i][v] = this.weights[i - 1] + M[i -
1][v];
				else
```

```
                                M[i][v] = Math.min(M[i - 1][v],
this.weights[i - 1] + M[i  - 1][Math.max(0, v - this.values[i - 1])]);
                    }
            }
            int res = 0;
            int v = 1;
            while (v <= totalValueSum && M[this.numberOfItem][v] <=
this.maxWeight){
                    this.resultWeight = M[this.numberOfItem][v];
                    res = v;
                    v++;
            }
            return res;
      }
}
```

5.2 El problema del viajante de comercio

MainClass.java

```
package ar.fiuba.tda.TspApproximation;


import java.util.concurrent.TimeUnit;
import ar.fiuba.tda.TspApproximation.ProblemaViajante.Resultado;


public class MainClass {


 public static void main(String[] args) {
   if (args.length == 0) {
       System.err.println("Falta argumento(s) con el/los archivo(s)
.tsp");
       return;
   }
   for (String file : args) {
       try {
       ProblemaViajante viajante =
TspBuilder.crearProblemaViajante(file);
       long starTime = System.nanoTime();
       Resultado res = viajante.recorrer();
       long endTime = System.nanoTime();
       System.out.println("Tsp con " +
      String.valueOf(viajante.getCantidadCiudades())
```

```
          + " ciudades - Resultado: " + res.toString() + " - Tiempo: "
          + String.valueOf(TimeUnit.NANOSECONDS.toMillis(endTime -
starTime)) + " ms");
       } catch (Exception e) {
      e.printStackTrace();
       }
   }
 }
}
```

TspBuilder.java

```
package ar.fiuba.tda.TspApproximation;


import java.io.BufferedReader;
import java.io.FileReader;
import java.util.Scanner;


public class TspBuilder {
 public static ProblemaViajante crearProblemaViajante(String tspFile)
throws Exception {
   TspParser parser = new TspParser(tspFile);
   return new ProblemaViajante(parser.getGrafo(), 1,
parser.getDistancias());
 }


 private static class TspParser {
   private int dimension;
   private int[][] distancias;
   private Grafo g;
   Boolean isFullMatrix;


   public TspParser(String filename) throws Exception {
       FileReader input = new FileReader(filename);
       BufferedReader bufRead = new BufferedReader(input);


       String myLine;
       while ((myLine = bufRead.readLine()) != null) {
      if (myLine.startsWith("DIMENSION")) {
        dimension = Integer.parseInt(myLine.replaceAll("[\\D]", ""));
        distancias = new int[dimension][dimension];
        g = new Grafo(dimension);
      } else if (myLine.startsWith("EDGE_WEIGHT_FORMAT")) {
          isFullMatrix = myLine.contains("FULL_MATRIX");
      } else if (myLine.startsWith("EDGE_WEIGHT_SECTION")) {
          int i = 0, j = 0;
```

```
        if (isFullMatrix) {
         while ((myLine = bufRead.readLine()) != null) {
           @SuppressWarnings("resource")
           Scanner in = new Scanner(myLine);
           while (in.hasNextInt()) {
             int costo = in.nextInt();
             distancias[i][j] = costo;
             if (j < i) {
               g.add_edge(i, j, costo);
             }
             if ((j = (j + 1) % dimension) == 0) {
               if (++i == dimension) {
                 in.close();
                 bufRead.close();
                 return;
               }
             }
           }
         }
        } else {
         while ((myLine = bufRead.readLine()) != null) {
           @SuppressWarnings("resource")
           Scanner in = new Scanner(myLine);
           while (in.hasNextInt()) {
             int distancia = in.nextInt();
             distancias[i][j] = distancias[j][i] = distancia;
             if (j == i) {
               j = 0;
               if (++i == dimension) {
            in.close();
            bufRead.close();
            return;
               }
             } else {
            g.add_edge(i, j, distancia);
            ++j;
             }
           }
        }
         }
      }
     }
    bufRead.close();
    throw new Exception("Mal parseo");
}


public Grafo getGrafo() {
    return g;
}


public int[][] getDistancias() {
    return distancias;
```

```
   }
 }
}
```

Arista.java

```
package ar.fiuba.tda.TspApproximation;


public class Arista {
 private int src, dst, weight;


 public Arista(int origen, int destino, int peso) {
   assert (src >= 0 && dst >= 0 && weight >= 0);


   src = origen;
   dst = destino;
   weight = peso;
 }


 public int getOrigen() {
   return src;
 }


 public int getDestino() {
   return dst;
 }


 public int getPeso() {
   return weight;
 }
}
```

Grafo.java

```
package ar.fiuba.tda.TspApproximation;


import java.util.ArrayList;
import java.util.Iterator;
import java.util.PriorityQueue;


public class Grafo {
 private ArrayList<Arista>[] adjacents;
 private int E;
```

```
/* Construye un grafo sin aristas de V vértices */
 @SuppressWarnings("unchecked")
 public Grafo(int V) {
   assert (V > 0);
   adjacents = (ArrayList<Arista>[]) (new ArrayList[V]);
   for (int i = 0; i < adjacents.length; i++) {
       adjacents[i] = new ArrayList<Arista>();
   }
   this.E = 0;
 }


/* Número de vértices en el grafo */
 public int V() {
   return adjacents.length;
 }


/* Número de aristas en el grafo */
 public int E() {
   return E;
 }


/* Itera sobre los aristas incidentes _desde_ v */
 public Arista[] adj_e(int v) {
   assert (v >= 0 && v < adjacents.length);
   Arista[] aristas = new Arista[adjacents[v].size()];
   int i = 0;
   Iterator<Arista> it = adjacents[v].iterator();
   while (it.hasNext()) {
       aristas[i] = it.next();
       ++i;
   }
   return aristas;
 }


/* Itera sobre los vértices adyacentes a 'v' */
 public int[] adj(int v) {
   assert (v >= 0 && v < adjacents.length);
   int[] adjList = new int[adjacents[v].size()];
   for (int i = 0; i < adjacents[v].size(); i++) {
       adjList[i] = adjacents[v].get(i).getDestino();
   }
   return adjList;
 }


/* Añade una arista al grafo */
 public void add_edge(int u, int v, int peso) {
   adjacents[u].add(new Arista(u, v, peso));
   adjacents[v].add(new Arista(v, u, peso));
   E++;
```

```
 }


/* Genera arbol de tendido minimo */
 public Grafo generarArbolDeTendido(int inicio) {
   PriorityQueue<PrVertex> cola = new PriorityQueue<PrVertex>(V());


   /* Estructuras auxiliares con info de la cola para rapida consulta */
   boolean[] estaEnCola = new boolean[V()];
   int[] padre = new int[V()];
   int[] costo = new int[V()];


   /* Init */
   for (int v = 0; v < V(); v++) {
       estaEnCola[v] = true;
       padre[v] = -1;
       if (v != inicio) {
      costo[v] = Integer.MAX_VALUE;
       } else {
      costo[v] = 0;
       }
       cola.offer(new PrVertex(v, costo[v]));
   }


   while (!cola.isEmpty()) {
       PrVertex vConPeso = cola.poll();
       int v = vConPeso.vertice;
       estaEnCola[v] = false;


       for (Arista e : adj_e(v)) {
       int u = e.getDestino();
       if (estaEnCola[u] && costo[u] > e.getPeso()) {
          padre[u] = v;
          costo[u] = e.getPeso();
          // No se usa el costo para el remove
          // No tengo metodo update
          cola.remove(new PrVertex(u, 0));
          cola.offer(new PrVertex(u, costo[u]));
       }
       }
   }


   Grafo arbol = new Grafo(V());
   for (int v = 0; v < V(); ++v) {
       if (v != inicio) {
          arbol.add_edge(padre[v], v, costo[v]);
       }
   }
   return arbol;
```

```
 }


 private class PrVertex implements Comparable<PrVertex> {
   public int vertice, costo;


   public PrVertex(int v, int costo) {
       this.vertice = v;
       this.costo = costo;
   }


   @Override
   public int compareTo(PrVertex o) {
       return Integer.compare(this.costo, o.costo);
   }


   @Override
   public boolean equals(Object o) {
       PrVertex other = (PrVertex) o;
       return vertice == other.vertice;
   }
 }
}
```

ProblemaViajante.java

```
package ar.fiuba.tda.TspApproximation;


import java.util.Stack;


public class ProblemaViajante {
 private Grafo gOriginal;
 private int[][] distancias;
 private int origen;


 // Las ciudades se numeran de 1..N
 public ProblemaViajante(Grafo grafo, int ciudadOrigen, int[][]
distancias) {
   assert (ciudadOrigen > 0 && ciudadOrigen <= grafo.V());
   gOriginal = grafo;
   this.distancias = distancias;
   origen = ciudadOrigen - 1; // Transf idCiudad a indice
 }


 public int getCantidadCiudades() {
   return gOriginal.V();
```

```
}


public Resultado recorrer() {
  Grafo arbolTendido = gOriginal.generarArbolDeTendido(origen);


  /* DFS del arbol */
  int camino[] = new int[arbolTendido.V() + 1];
  int index = 0;
  boolean etiquetado[] = new boolean[arbolTendido.V()];
  for (int i = 0; i < etiquetado.length; ++i) {
      etiquetado[i] = false;
  }


  Stack<Integer> pila = new Stack<Integer>();
  pila.push(origen);


  while (!pila.empty()) {
    int v = pila.pop();
    if (!etiquetado[v]) {
      etiquetado[v] = true;
      camino[index++] = v + 1; // Indice a idCiudad


      for (int u : arbolTendido.adj(v)) {
     pila.push(u);
      }
    }
  }
  camino[arbolTendido.V()] = origen + 1;


  /* Calculo costo total */
  int costo = 0;
  for (int i = 0; i < camino.length - 1; ++i) {
      // Resto uno a los vertices xq son id y no indices
      costo += distancias[camino[i] - 1][camino[i + 1] - 1];
  }


  return new Resultado(camino, costo);
}


public class Resultado {
  public int camino[];
  public int costo;


  Resultado(int camino[], int costo) {
      this.camino = camino;
```

```
        this.costo = costo;
    }


    @Override
    public String toString() {
      StringBuilder b = new StringBuilder("Camino: [");
      for (int v : camino) {
       b.append(String.valueOf(v) + ',');
      }
      b.setCharAt(b.length() - 1, ']');
      b.append(" - Costo: " + String.valueOf(costo));
      return b.toString();
    }
  }
}
```