DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

subtitulo del trabajo

Algoritmos 3
Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Ricardo Colombo	156/08	ricardogcolombo@gmail.com.com
Federico Suarez	610/11	elgeniofederico@gmail.com
Juan Carlos Giudici	827/06	elchudi@gmail.com
Franco Negri	893/13	franconegri2004@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Plan de vuelo

1.1. Introducción

En este ejercicio se nos pide encontrar un algoritmo que encuentre una combinacion de vuelos entre la ciudad A y la ciudad B tal que encuentre la manera de llegar lo antes posible a destino.
La entrada del algoritmo será:

- Un string **A** $\rightarrow$ Representará la ciudad de partida.
- Un string **B** $\rightarrow$ Representará la ciudad de de llegada.
- Un entero **n** $\rightarrow$ Representará el numero total de vuelos entre todas las ciudades.
- **n** filas donde, para cada fila se tiene:
  - Un string **ori** $\rightarrow$ Representará la ciudad de partida.
  - Un string **des** $\rightarrow$ Representará la ciudad de de llegada.
  - Un entero **ini** $\rightarrow$ Representará el numero la hora de despegue de la ciudad ori
  - Un entero **fin** $\rightarrow$ Representará el numero la hora de arribo a la ciudad des

A esto nuestro algoritmo debe devolver:

- Un entero **fin** $\rightarrow$ Representará el horario de llegada a la ciudad B.
- Un entero **k** $\rightarrow$ la cantidad de vuelos del itinerario.
- k enteros **v_1, v_2 ..., v_k** $\rightarrow$ los vuelos tomados

1.2. Ejemplos y Soluciones

Se procede a generar una posible instancia del problema para ilustrar lo que se espera del algoritmo. Supongamos que queremos ir de la ciudad de Buenos Aires a la isla de Saba (Dato curioso: en la isla de Saba se encuentra el aeropuerto mas chico del mundo (400 m)).
Lamentablemente no existen vuelos directos, por lo que tendrémos que hacer escala para poder llegar ahí. A continuacíón se muestran los posibles vuelos que podríamos tomar para llegar a destino.

- Buenos Aires - Seúl: 10 - 20
- Buenos Aires - La isla de los pitufos: 17 - 24
- Seúl - San Fransisco: 22 - 40
- La isla de los pitufos - isla de Saba: 28 - 30
- San Fransisco - isla de Saba: 40 - 43
- Buenos Aires - San Fransisco: 13 - 20

De esta simple instancia, ya vemos que es posible combinar los vuelos de varias maneras posibles. Podría irse desde Buenos Aires a Seul, y de Seul a San Fransisco, y de allí a Saba.
O de Buenos aires a San Fransisco directo, sin pasar por Seul y de allí a Saba.
Y ya aquí puede la intuición nos indicaría que lo que a uno le gustaría hacer es minimizar es algun problema de caminos minimos. Tal vez donde los pesos de las aristas estan indicados por el tiempo que se tarda, agregando de alguna manera esta condición extra de que para poder tomar un vuelo, es necesario estar en esa ciudad dos horas antes en la ciudad de la que se parte. En el apartado siguiente profundisaremos sobre esa idea.
Sin embargo, para esta pequeña instancia, es facil resolver a mano el problema y descubrir que la combinación de vuelos optima es:

- Buenos Aires - La isla de los pitufos: 17 - 24
- La isla de los pitufos - isla de Saba: 28 - 30

Llegando a destino a la hora 30.

1.3. Desarrollo

Podemos observar del ejemplo anterior, que existen muchas maneras factibles de llegar desde Buenos Aires al objetivo. Una manera es tomar la ruta de Buenos aires - Seul, de allí ir a San Fransisco y finalmente a la isla de Saba, pero rapidamente vemos que si bien este viaje es factible, es al menos tan bueno como tomar el vuelo de Buenos aires a san fransisco y de allí continuar a Saba. De esta observación se desprende un dato interesante que utilizaremos en nuestro algoritmo, si se llegar de la manera mas barata a un grupo de ciudades v_1, v_2, ... v_s y tengo w_1, w_2,...w_i ciudades a las que puedo llegar desde el primer grupo. Si ahora tomo la ciudad w_g tal que la hora de arribo a esa ciudad desde una en v es la menor posible, sé que no es posible llegar de una manera mas barata a esa ciudad tambien.

Rapidamente esta idea nos remite al algoritmo de dijstra, el cual comparte una gran similitud en la idea de ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices hasta llegar al destino.

Luego la idea subyacente en este problema es, utilizando un algoritmo de dijstra levemente modificado, en cada paso, calcular la manera mas barata de llegar desde un conjuto de ciudades ya visitadas, a uno quue todavía no hemos visitado.

Luego el pseudocodigo para este algoritmo será algo asi:

1: Para toda ciudad, pongo su peso en infinito O(n)
2: Para toda ciudad, pongo el vuelo que tuve que tomar para llegar a ella en 0 O(n)
3: Para la ciudad de inicio, pongo su peso en 0
4: Para i entre 1 y n
5: $CostoMinimo = \infty$
6: Para j entre 1 y n
7: Si el peso de la ciudad de origen del vuelo j es distinto de ∞
8: Si la hora a la que llegue a la ciudad del vuelo j mas dos es menor a la hora a la que parte el vuelo j
9: SI $CostoMinimo$ es mayor a horario al que finaliza el vuelo $j + 2$
10: Asigno horario al que finaliza el vuelo $j + 2$ como $CostoMinimo$
11: Me guardo el vuelo j en una variable $VueloMinimo$
12: Si $CostoMinimo == \infty$
13: No puedo alcanzar ninguna otra ciudad, salgo del ciclo
14: sino asigno al peso de la ciudad de destino guardada en $VueloMinimo$, $CostoMinimo$
15: guardo que la manera de llegar a esta ciudad de destino es por $VueloMinimo$

Algorithm 1: BuscarCaminoMinimo

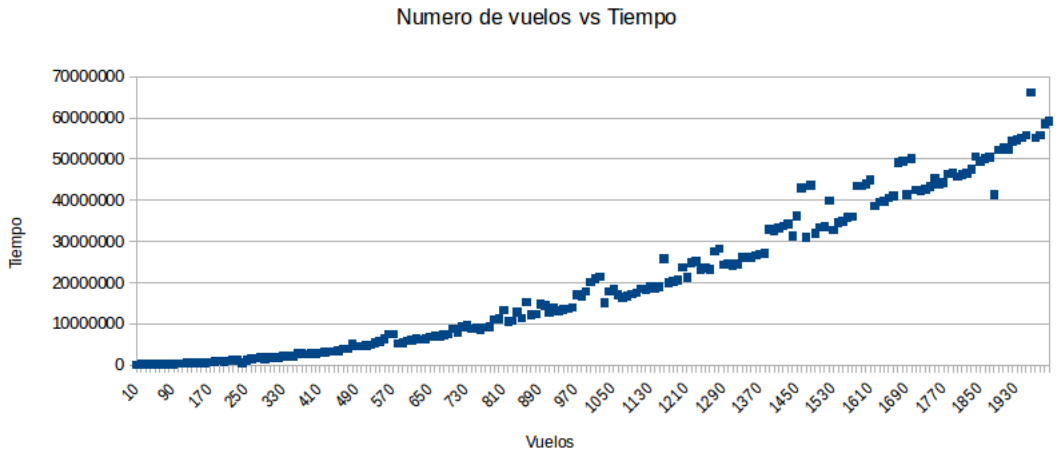
Al finalizar este algoritmo tengo todas las maneras mas rapidas de llegar de la ciudad de origen a todas las otras ciudades, siempre que exista una secuencia de vuelos que lo permitan.

1.4. Complejidad

Dado que el algoritmo consiste en iterar por dos loops anidados, cuyo tamaño es n (osea, la cantidad de vuelos), la complejidad del algoritmo será $O(n^2)$.
Ademas, este algoritmo para cada paso del loop mas externo, itera forzosamente por todo el arreglo para determinar el minimo, por lo que sabemos que, para el caso en que existe una solución $\Omega(n^2)$.

1.5. Experimentacion

Dado que ya dijimos que no existen, 'peores casos', porque nuestro algoritmo esta acotado por ambos lados por n^2, realizamos un testeo random para comprobarlo:



Aqui puede verse claramente que nuestras hipotesis eran correctas.

2. Caballos salvajes

2.1. Introducción

Para este ejercicio, se nos pide encontrar un algoritmo, que, dados k caballos repartidos por un tablero de n por n casillas, encuentre cual es la casilla donde puedo reunir a todos los caballos en la menor cantidad de saltos.
Nuestra entrada será:

- Un entero **n** $\rightarrow$ Representarán el largo y el ancho del tablero.
- Un entero **k** $\rightarrow$ Representará el numero de caballos repartidos en el.
- **k** filas donde, para cada fila se tiene:
  - f c $\rightarrow$ Representarán la fila y la columna de cada caballo.

A esto nuestro algoritmo deve devolver:

- Un entero **f c** $\rightarrow$ Representará la fila y columna a donde deven converger los caballos.
- Un entero **m** $\rightarrow$ Representará el numero total de saltos que le costará a todos los caballos llegar hasta ahí.

2.2. Ejemplos y Soluciones

Se procede ahora a realizar un ejemplo para ilustrar el problema.
Supongamos que tenemos un tablero de 4 por 4 con un caballo en la posición 1,1 y otro en la posición 4,4.
La entrada del problema luego sería:

- 4 2
- 1 1
- 4 4

Para este caso es posible encontrar una solución a mano, por ejemplo, es facil ver que en dos movimientos es posible hacer converger a amobos caballos.
En caso de querer asegurarnos de ello, podríamos hacer lo siguiente. Dibujamos en un papel dos matrices de n por n. En la primera matriz, vamos a poner cual es la cantidad minima de saltos que el primer caballo realiza para saltar a cada una de las casillas del tablero.
Para ello primero anotamos en la matriz con costo 0 la posición donde se encuentra el primer caballo. Ahora, saltamos desde esta posición a todas las posibles posiciones válidas del tablero. Todas estas tendran costo 1.
Ahora, desde todas las posiciones de costo 1 saltamos a todas las posiciones válidas del tablero que podamos. Estas van a tener costo 2. Si seguimos realizando este procedimiento, demostraremos que obtenemos la cantidad mínima de saltos que el primer caballo realiza para saltar a cada una de las casillas del tablero, que era lo que buscabamos.
Aqui puede verse la matriz obtenida:

$$A = \begin{bmatrix} 0 & 3 & 2 & 5 \\ 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 3 \\ 5 & 2 & 3 & 2 \end{bmatrix}$$

Realizamos lo mismo con el segundo caballo, marcamos la casilla donde se encuentra parado con costo 0 y empezamos a saltar a las casillas válidas.

La matriz será:

$$B = \begin{bmatrix} 2 & 3 & 2 & 5 \\ 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 3 \\ 5 & 2 & 3 & 0 \end{bmatrix}$$

Ahora sumamos ambas matrices, y lo que obtenemos es una matriz con los costos mínimos de que todos los caballos salten a cada una de las posiciones del tablero.

$$A + B = \begin{bmatrix} 2 & 6 & 4 & 10 \\ 6 & 8 & 2 & 4 \\ 4 & 2 & 8 & 6 \\ 10 & 4 & 6 & 2 \end{bmatrix}$$

Buscando los mínimos en esta matriz, obtenemos lo que queríamos.
Luego, algunas soluciones que el algoritmo podría devolver en este caso son:

- 1 1 2
- 2 3 2
- 4 4 2

2.3. Desarrollo

La idea general del algoritmo es sencilla, para cada caballo, confeccionamos una matriz con el costo minimo de saltar a cada uno de los casilleros de la matriz. Luego sumando estas k matrices, obtenemos el costo minimo de que cada caballo salte a cada uno de los casilleros.
Para asegurarnos de que en cada paso estamos tomando efectivamente la menor cantidad de saltos para que un caballo llegue a un casillero de la matriz, podemos pensar a la misma como un grafo, en el cual dos nodos estan conectados si y solo si un caballo puede saltar de uno a otro de manera valida.
Luego solo basta realizar un BFS para obtener el costo minimo de que un caballo llegue a esa casilla.
Cabe destacar, que por una cuestión de claridad, en la implementacion final, la idea de recorrer un grafo está implicita, la misma solo nos ayuda a ver que tanto la complejidad como la correctitud son las adecuadas en el problema dado.
En la implementación, simplemente creamos k matrices de enteros de n por n. Luego para cada caballo, tomamos todos los nodos de distancia j, buscamos todos los nodos válidos de distancia $j + 1$ y los seteamos. Realizamos esto hasta que no quedan nodos no seteados y allí pasamos de caballo.
Mas formalmente:

1: Generar k matrices de nxn todas seteadas en infinito
2: Creo dos colas: colaDeProfundidadJ,colaDeProfundidadJmasUno
3: Para cada caballo, tomo la casilla donde se encuentra y lo encolo en colaDeProfundidadJ
4: Creo un entero j igual a 0
5: Mientras colaDeProfundidadJ no este vacía.
6: Para toda casilla $\in$ colaDeProfundidadJ
7: En la matriz correspondiente a este caballo, asigno j, como el valor del nodo
8: Busco los vecinos, si estan seteados en infinito los encolo en colaDeProfundidadJmasUno
9: Sumo 1 a j
10: Encolo los valores de colaDeProfundidadJmasUno en colaDeProfundidadJ
11: Vacío colaDeProfundidadJmasUno
12: Sumo las k matrices
13: Busco el minimo
14: imprimo el minimo

Algorithm 2: void FuncionPrincipal()

2.4. Demostración De Correctitud

Para demostrar la correctitud de este algoritmo, primero, demostramos que dada una matriz de nxn, $n \geq 4$ desde cualquier casillero existe una sucesión de saltos para llegar a cualquier otro.
Por inducción en n, siendo n la cantidad de filas y de columnas.
Caso base:
Ya vimos en el ejemplo que desde en un tablero de 4x4 es posible llegar a cualquier casillero:

$$A = \begin{bmatrix} 0 & 3 & 2 & 5 \\ 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 3 \\ 5 & 2 & 3 & 2 \end{bmatrix}$$

Caso inductivo:

Luego, si existe el tablero de nxn tal que de cualquier casillero podemos ir a cualquier casillero, quiero demostrar el tablero de $n+1$x$n+1$ también cumple.
Tomamos el casillero hubicado en la fila $n+1$ columna 1 (que notamos $(n+1,1)$), y vemos que existe un salto al casillero ($n-1$,2), luego, existe una sucesión de saltos desde el casillero $(n+1,1)$ a cualquiera del tablero de nxn.
Para todos los casilleros en ($n+1$,i) con $i > 1$, podemos saltar al casillero ($n-1$,$i-1$), y luego desde todos estos casilleros, también podemos saltar dentro del tablero de nxn.
Ahora podemos ver que para la casilla (1,$n+1$), también existe el salto a (2,$n-1$), que hace que también cumpla.
De la misma manera, para los casilleros para (i,$n+1$) con $1 < i \geq n+1$ existe el salto a la casilla ($i-1$,$n-1$).
Con lo cual concluimos que existe un salto desde cualquier casillero de la fila $n+1$ o columna $n+1$ podemos saltar a cualquier casillero del tablero de nxn, por lo tanto, por hipótesis inductiva, existe una sucesión de saltos para llegar a cualquier casillero del tablero, en particular, puedo ir desde cualquier casillero de la fila/columna $n+1$x$n+1$ a cualquier otro que pertenezca a la misma fila/columna.
A partir de un tablero de nxn el modelado cada casillero representa un nodo, y dos nodos son adyacentes sii existe un salto de caballo en el tablero.

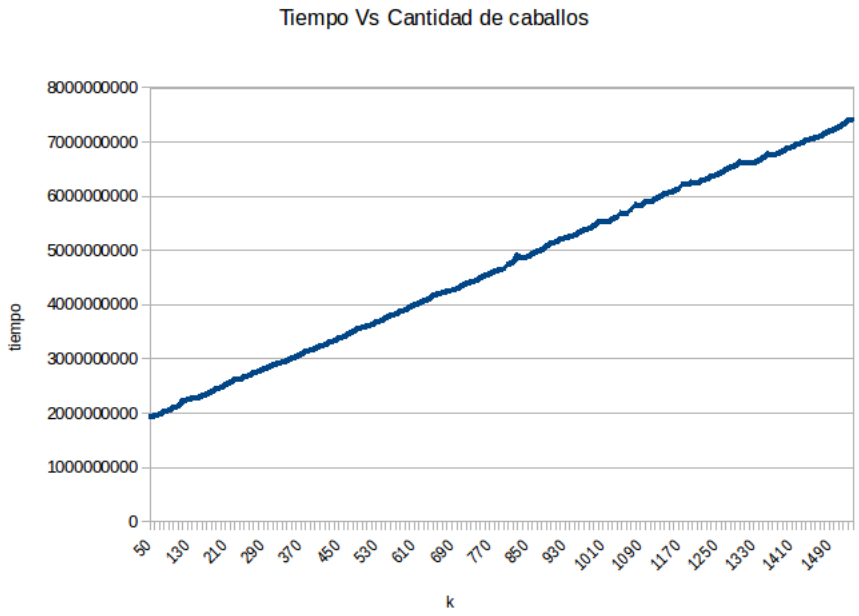
Por la demostración anterior, sabemos que desde cualquier casillero existe una sucesión de saltos para llegar a cualquier otro, traducido a nuestro modelo de grafo, quiere decir que éste es conexo.
Como el grafo es conexo y finito, realizando un bfs para cada caballo, podemos obtener la cantidad mínima de saltos desde la posición en la que se encuentra el caballo hasta todas las otras del tablero.
Por lo tanto, sumando cuantos saltos debe realizar cada caballo a cada casillero, obtengo el mínimo de saltos que tienen que dar todos los caballos para llegar a cualquier casillero.
Para cada casillero, sumanos los saltos de cada caballo y nos quedamos con el que tenga suma mínima.

2.5. Complejidad

Para cada caballo, encolamos una sola vez cada nodo y posteriormente lo procesamos.
Entonces la complejidad es la cantidad de caballos (k) por la cantidad de nodos (n^2).
En otras palabras la complejidad es $O(kn^2)$.

2.6. Experimentacion

Para probar el rendimiento de nuestro algoritmo se creó un generador de entradas que fabricará instancias al azar del problema.
Para la primera experimentación se deja el tablero fijo, y se varía el numero de caballos. Para este testeo, se tomó $n = 100$ y se hizo variar el numero de caballos entre 50 y 1500, cada uno posicionado de manera aleatoria en una casilla, y se tomaron 50 muestras de cada uno.
Los resultados obtenidos se muestran en el siguiente grafico.



Puede verse claramente que el algoritmo se comporta de manera lineal con respecto a la cantidad de caballos

3. La comunidad del anillo

3.1. Introducción

En este ejercicio se propone solucionar el problema de dado una red de existente de computadoras, con a lo sumo un enlace entre cada par de ellas y con un costo asociado al mismo, seleccionar algunas de estas para formar un backbone con topología de red tipo anillo, la cual tiene que tener como característica que conecte a todas las computadoras originales y que minimice el costo de ancho de banda de la red.

Se pide un algoritmo que genere este backbone, con un costo temporal estrictamente menor que O(n^3), este algoritmo debe detectar casos en los que no hay solucion.

3.2. Desarrollo

3.2.1. Modelado

Dada una red, la misma se puede modelar con un grafo de la siguiente manera:

1. Cada computadora se representa con un nodo.
2. Los enlaces entre cada par de computadoras se son los ejes en mi grafo, con el costo de ancho de banda como peso del mismo.

Transformamos el problema de ser uno de redes, a uno de grafos, donde encontrar un backbone con topologia de red tipo anillo que tenga costo mínimo y conecte a todas las computadoras, se transforma en encontrar subgrafo conexo cuyo costo sea mínimo y que contenga un único circuito simple, el cual se corresponde al backbone.

3.2.2. Solución, Correctitud y Complejidad

Suponiendo que el grafo resultante es conexo, ya que si no lo fuese no habria soluci'on, vamos a utilizar el algoritmo de Prim, el cual dado un grafo conexo con pesos asociados a sus ejes construye un Árbol Generador Mínimo, es decir un subgrafo conexo cuya la suma de los pesos de sus ejes es mínima. Para completar el circuito, seleccionamos el eje con costo mínimo de los no elegidos por el algoritmo de Prim, el cual nos va a generar un único circuito simple.

Este último paso se justifica por lo visto en las clases teóricas donde demostramos que dado un Árbol, si agregamos un eje entre cualquier par de nodos, se forma un único circuito simple.

El algoritmo es el siguiente:

```
1: si no Es_conexo_o_no_tiene_ejes_suficientes_para_construir_un_circuito(G)
2:     devolver no
3: agm, ejes_no_seleccionados ← Prim(G)
4: eje_minimo ← Encontrar_Eje_Minimo(ejes_no_seleccionados)
5: circuito ← Construir_Circuito(agm, eje_minimo)
6: Mostrar circuito
```

Algorithm 3: EncontrarBackBone(G(E,V))

- Nuestra implementación del algoritmo de Prim, ademas del árbol generador mínimo genera una lista de ejes no seleccionados, la cual vamos a usar para encontrar el eje mínimo y completar el circuito.
- `Es_conexo_o_no_tiene_ejes_suficientes_para_construir_un_circuito(G)` recorre el grafo mediante DFS, llevando la cuenta de los nodos visitados para decidir si es conexo una vez finalizado y para decidir si es posible armar un circuito verifica que `m >= n`.
- `Encontrar_Eje_Minimo(ejes_no_seleccionados)` busca linealmente en la cantidad de ejes (a lo sumo $m = n^2$) el eje con costo mínimo.

- `Construir_Circuito(agm, eje_minimo)` Toma como punto principio y final los nodos del eje_minimo y mediante DFS construye el circuito.

3.2.3. Complejidad

La complejidad del algoritmo `EncontrarBackBone` es la siguiente:

- `Es_conexo_o_no_tiene_ejese_suficientes_para_construir_un_circuito(G)` tiene un costo de O(n^2), esta implementado mediante una variacion del algoritmo de DFS, el cual va marcando los nodos visitados.
- La implementación de `Prim(G)` que utilizamos tiene un costo O(n^2), dado que utilizamos una matriz de adyacencias.
- `Encontrar_Eje_Minimo(ejes_no_seleccionados)` tiene un costo de O(n^2)
- `Construir_Circuito(agm, eje_minimo)` tiene un costo de O(n), dado que el agm tiene $n - 1$ aristas.

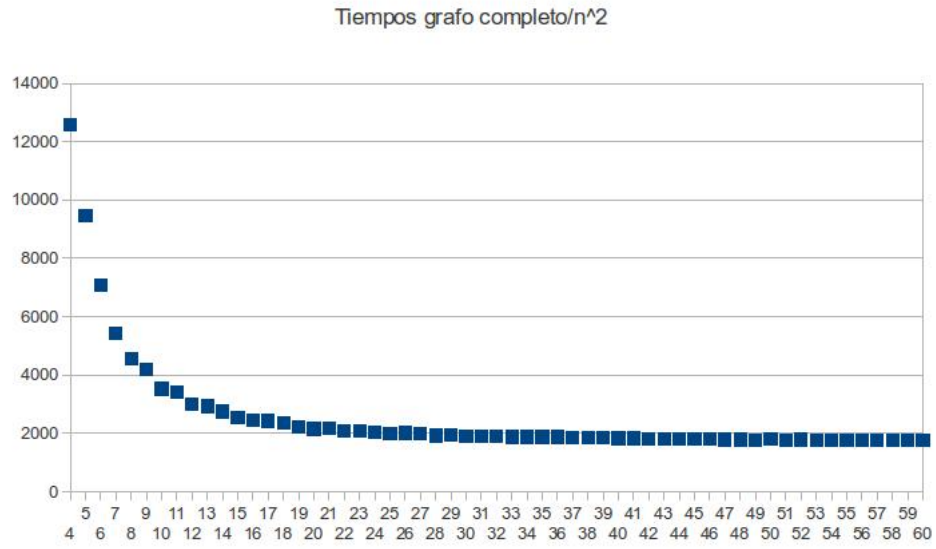
Por lo tanto, el algoritmo tiene un orden temporal de O(n^2), cumpliendo con lo pedido en el enunciado.

3.3. Experimentacion

Para probar que efectivamente nuestro algoritmo es cuadratico con respecto a n en el peor caso, creamos grafos completos con un generador de entrada, y corremos 50 tests, para reducir el posible factor de ruido que puede generar correr nuestro algoritmo de manera concurrente con otros procesos del sismtema. Los grafos completos que tomamos van desde el grafo K_1 hasta el K_60.

Los resultados pueden observarse en la siguiente grafica:

Dividimos por una función cuadratica para ver que se obtiene:



Si bien en los primeros casos, el algoritmo pareciera no ser lineal, esto puede deverse a que la instancia del problema es muy pequeña y los outliers en ese caso son mayores.
Para las instancias mas grandes se vé que el algoritmo el casi perfectamente constante. Luego, de manera empirica, podemos concluír que nuestro algoritmo es cuadratico con respecto a n en el peor caso.

4. Aclaraciones

4.1. Medicion de los tiempos

Para este tp como trabajamos bajo el lenguaje de programacion C++, decidimos calcular los tiempos utilizando 'chrono' de la libreria standard de c++ (chrono.h) que nos permite calcular el tiempo al principio del algoritmo y al final, y devolver la resta en la unidad de tiempo que deseamos.

5. Código Fuente

5.1. Ej1.cpp

```
struct vuelo
{
  int ini;
  int fin;
  int numeroDeVuelo;
  int origen;
  int destino;
};

vector<vuelo> dijstra_sin_grafo(vector<vuelo> vector_vuelos,int origen,int destino, int cantidad_de_ciudades)
{
        vector<int> costo_de_llegar_a_esta_ciudad;
        vector<vuelo> vuelo_tomado_para_llegar_a_esta_ciudad;
        vuelo_tomado_para_llegar_a_esta_ciudad.resize(cantidad_de_ciudades);

        for(int i = 0; i < cantidad_de_ciudades; i++)
        {
                costo_de_llegar_a_esta_ciudad.push_back(INT_MAX);
        }

        costo_de_llegar_a_esta_ciudad[origen] = -100;

        for(int i = 0; i < vector_vuelos.size(); i++)
        {
                int min_costo = INT_MAX;
                vuelo vuelo_de_costo_min;
                for(int j = 0; j < vector_vuelos.size(); j++)
                {
                        //si la ciudad de origen esta en el grupo de los marcados, y la de destino no
                        if(costo_de_llegar_a_esta_ciudad[vector_vuelos[j].origen] != INT_MAX &&
    costo_de_llegar_a_esta_ciudad[vector_vuelos[j].destino] == INT_MAX)
                        {
                                //si cumple la condicion de los vuelos
                                if(costo_de_llegar_a_esta_ciudad[vector_vuelos[j].origen] + 2 <= vector_vuelos[j].ini)
                                {
                                        //si el peso para esta ciudad de menor que el que ya tenia guardado.
                                        if(min_costo > vector_vuelos[j].fin);
                                        {
                                                //me guardo este vuelo
                                                min_costo = vector_vuelos[j].fin;
                                                vuelo_de_costo_min = vector_vuelos[j];
                                        }
                                }
                        }
                }
                // si el costo minimo continua siendo infinito, no existe manera de llegar desde las ciudades ya marcadas al
    resto de los vertices, detengo dijstra
                if(min_costo == INT_MAX)
                {
                        break;
```

```
                }
                //agrego la ciudad a las ciudades ya marcadas e itero.
                 costo_de_llegar_a_esta_ciudad [vuelo_de_costo_min.destino] = min_costo;
                vuelo_tomado_para_llegar_a_esta_ciudad[vuelo_de_costo_min.destino] = vuelo_de_costo_min;
        }

        if( costo_de_llegar_a_esta_ciudad [destino] == INT_MAX)
        {
                vuelo_tomado_para_llegar_a_esta_ciudad.erase(vuelo_tomado_para_llegar_a_esta_ciudad.begin(),
    vuelo_tomado_para_llegar_a_esta_ciudad.end());
        }
        return vuelo_tomado_para_llegar_a_esta_ciudad;
}

void encontrar_camino(vector<vuelo> vuelo_tomado_para_llegar_a_esta_ciudad,int destino,int origen)
{
        vector<vuelo> resultado;
        resultado.push_back(vuelo_tomado_para_llegar_a_esta_ciudad[destino]);
        bool fin = false;
        // partiendo del destino, regenero el camino para llegar a la solucion
        int ciudadAnterior = (vuelo_tomado_para_llegar_a_esta_ciudad[destino].origen);
        while(!fin)
        {
                vuelo vueloQueTomo = vuelo_tomado_para_llegar_a_esta_ciudad[ciudadAnterior];
                resultado.push_back(vueloQueTomo);
                ciudadAnterior = vueloQueTomo.origen;
                if(ciudadAnterior == origen)
                        fin = true;
        }
        reverse(resultado.begin(),resultado.end());

        cout << resultado[resultado.size() - 1].fin << "␣" << resultado.size() << "␣";
        for(int i = 0; i < resultado.size(); i++)
                cout << resultado[i].numeroDeVuelo << "␣";
        cout << endl;
}



int main(int argc, char *argv[]){

  map< string, int> dict_ciudad_vector_distancias;
  int cant_vuelos;
  string origen, destino;
  cin >> origen;
  dict_ciudad_vector_distancias [origen] = 0;
  cin >> destino;
  dict_ciudad_vector_distancias [destino] = 1;
  cin >> cant_vuelos;
  vector< vuelo > vector_vuelos;
  int cantidad_de_ciudades = 2;

  for (int i = 0; i < cant_vuelos; i++) {
    vuelo v;
    string origen_vuelo, destino_vuelo;

    cin >> origen_vuelo;
    cin >> destino_vuelo;

    //checkeo si tengo los destinos en mi vector de coordenadas
    if( dict_ciudad_vector_distancias .find(origen_vuelo) == dict_ciudad_vector_distancias.end()) {
      dict_ciudad_vector_distancias [origen_vuelo] = cantidad_de_ciudades;
      cantidad_de_ciudades++;
    }


    if( dict_ciudad_vector_distancias .find(destino_vuelo) == dict_ciudad_vector_distancias.end()) {
      dict_ciudad_vector_distancias [destino_vuelo] = cantidad_de_ciudades;
```

```
        cantidad_de_ciudades++;
      }

      // cada ciudad es representada con un numero que a su vez se guarda en un diccionario.
      v.origen = dict_ciudad_vector_distancias[origen_vuelo];
      v.destino = dict_ciudad_vector_distancias[destino_vuelo];

      cin >> v.ini;
      cin >> v.fin;
      v.numeroDeVuelo = i+1;
      vector_vuelos.push_back(v);
    }

    vector<vuelo> vuelo_tomado_para_llegar_a_esta_ciudad = dijstra_sin_grafo(vector_vuelos, dict_ciudad_vector_distancias[origen
        ], dict_ciudad_vector_distancias[destino],cantidad_de_ciudades);

    //linea para imprimir el resultado, descomentar para entregar
    if(vuelo_tomado_para_llegar_a_esta_ciudad.size() != 0)
          encontrar_camino(vuelo_tomado_para_llegar_a_esta_ciudad, dict_ciudad_vector_distancias[destino],
        dict_ciudad_vector_distancias[origen]);
    else
          cout << "no" << endl;
    return 0;
}
```

5.2. Ej2.cpp

```
#define NODO_NO_MARCADO INT_MAX
#define NODO_MARCADO INT_MAX−1

struct coordenada
{
  int x;
  int y;
};

struct tablero
{
  vector< vector< int> > casillas;
  int n;
};

void agregar_nodos_de_profunidad_k_mas_uno(coordenada nodo, queue<coordenada> *nodos_de_altura_k_mas_uno, tablero &
    unTablero)
{
    //chequeo el rango, si el caballo salta a una posicion valida y no salte previamente a esta posicion, lo agrego a la
    cola
    //me aseguro de no meter dos veces en la cola el mismo nodo 'maracandolo', asi, si de dos nodos de altura k puedo
    saltar al mismo nodo de altura k+1, solo lo agrego la primera vez

    //muy cabeza


    if (nodo.x − 2 >= 0)
    {
        if (nodo.y − 1 >= 0)
        {
        if (unTablero.casillas [nodo.x − 2][nodo.y − 1] == NODO_NO_MARCADO)
            {

                unTablero.casillas [nodo.x − 2][nodo.y − 1] = NODO_MARCADO;
                (*nodos_de_altura_k_mas_uno).push(crear_coordenada(nodo.x − 2, nodo.y − 1));
            }
        }
        if (nodo.y + 1 < unTablero.n)
        {
            if (unTablero.casillas [nodo.x − 2][nodo.y + 1] == NODO_NO_MARCADO)
            {

                unTablero.casillas [nodo.x − 2][nodo.y + 1] = NODO_MARCADO;
                (*nodos_de_altura_k_mas_uno).push(crear_coordenada(nodo.x − 2, nodo.y + 1));
            }
        }
    }

    if (nodo.x + 2 < unTablero.n)
    {
        if (nodo.y − 1 >= 0)
        {
            if (unTablero.casillas [nodo.x + 2][nodo.y − 1] == NODO_NO_MARCADO)
            {
                unTablero.casillas [nodo.x + 2][nodo.y − 1] = NODO_MARCADO;
                (*nodos_de_altura_k_mas_uno).push(crear_coordenada(nodo.x + 2, nodo.y − 1));
            }
        }
        if (nodo.y + 1 < unTablero.n)
        {
            if (unTablero.casillas [nodo.x + 2][nodo.y + 1] == NODO_NO_MARCADO)
            {
                unTablero.casillas [nodo.x + 2][nodo.y + 1] = NODO_MARCADO;
                (*nodos_de_altura_k_mas_uno).push(crear_coordenada(nodo.x + 2, nodo.y + 1));
            }
        }
```

```
        }
        if (nodo.x − 1 >= 0)
        {
                if (nodo.y − 2 >= 0)
                {
                        if (unTablero. casillas [nodo.x − 1][nodo.y − 2] == NODO_NO_MARCADO)
                        {
                                unTablero. casillas [nodo.x − 1][nodo.y − 2] = NODO_MARCADO;
                                (∗nodos_de_altura_k_mas_uno).push(crear_coordenada(nodo.x − 1, nodo.y − 2));
                        }
                }
                if (nodo.y + 2 < unTablero.n)
                {
                        if (unTablero. casillas [nodo.x − 1][nodo.y + 2] == NODO_NO_MARCADO)
                        {
                                unTablero. casillas [nodo.x − 1][nodo.y + 2] = NODO_MARCADO;
                                (∗nodos_de_altura_k_mas_uno).push(crear_coordenada(nodo.x − 1, nodo.y + 2));
                        }
                }
        }

        if (nodo.x + 1 < unTablero.n)
        {
                if (nodo.y − 2 >= 0)
                {
                        if (unTablero. casillas [nodo.x + 1][nodo.y − 2] == NODO_NO_MARCADO)
                        {
                                unTablero. casillas [nodo.x + 1][nodo.y − 2] = NODO_MARCADO;
                                (∗nodos_de_altura_k_mas_uno).push(crear_coordenada(nodo.x + 1, nodo.y − 2));
                        }
                }
                if (nodo.y + 2 < unTablero.n)
                {
                        if (unTablero. casillas [nodo.x + 1][nodo.y + 2] == NODO_NO_MARCADO)
                        {
                                unTablero. casillas [nodo.x + 1][nodo.y + 2] = NODO_MARCADO;
                                (∗nodos_de_altura_k_mas_uno).push(crear_coordenada(nodo.x + 1, nodo.y + 2));
                        }
                }
        }
}



int main()
{
        int n;
        int cantidad_de_caballos;
        vector<coordenada> lista_caballos;
        cin >> n;
        cin >> cantidad_de_caballos;


        for(int i = 0; i < cantidad_de_caballos; i++)
        {
                coordenada nuevoCaballo;
                cin >> nuevoCaballo.x;
                cin >> nuevoCaballo.y;
                //para que me quede congruente con las matrices les resto 1
                nuevoCaballo.x−−;
                nuevoCaballo.y−−;
                 lista_caballos .push_back(nuevoCaballo);
        }

        // en este tablero voy guardando cuanto le cuesta al caballo i, llegar a la posision (x,y) del tablero
        vector<tablero> tablero_para_caballo_i(cantidad_de_caballos, crear_tablero (n));

        for(int caballo_i = 0; caballo_i < cantidad_de_caballos; caballo_i ++)
```

```
{
        queue<coordenada> *nodos_de_altura_k = new queue<coordenada>;
        queue<coordenada> *nodos_de_altura_k_mas_uno = new queue<coordenada>;
        int k = 0;

        //meto el primer nodo de todos (donde esta el caballo inicialmente)
        (*nodos_de_altura_k).push( lista_caballos [ caballo_i ]) ;

        while(! (*nodos_de_altura_k).empty())
        {
            //mientras haya casillas de altura k, las recorro y agrego los nodos validos de altura k+1
            while(! (*nodos_de_altura_k).empty())
            {

                coordenada nodo = (*nodos_de_altura_k).front();

                (*nodos_de_altura_k).pop();

                //tengo asegurado que esto esto solo se va a asignar una vez, no necesito ifs
                  tablero_para_caballo_i [ caballo_i ]. casillas [nodo.x][nodo.y] = k;

                  agregar_nodos_de_profunidad_k_mas_uno(nodo, nodos_de_altura_k_mas_uno,
tablero_para_caballo_i[caballo_i]);
            }
            //ok, ya no hay nodos de altura k, ahora paso a k+1, (que va a ser el nuevo k), y borro lo que habia
en k+1

            delete nodos_de_altura_k;

            nodos_de_altura_k = nodos_de_altura_k_mas_uno;

            nodos_de_altura_k_mas_uno = new queue<coordenada>();

            k++;
        }
}

int suma_min = INT_MAX;
coordenada nodo_minimo;

for(int j = 0; j < n; j++)
{
        for(int k = 0; k < n; k++)
        {
            //para cada casilla voy sumando cuantos saltos le cuesta llegar a todos los caballos ahi
            int acum = 0;
            for(int i = 0; i < cantidad_de_caballos;i++)
            {
                if( tablero_para_caballo_i [i ]. casillas [j][k] < INT_MAX-1)
                {
                    acum += tablero_para_caballo_i[i]. casillas [j][k];
                }
                else
                {
                    acum = INT_MAX;
                    break;
                }
            }
            // si es menor, actualizo, ahora tengo una manera de llegar que cuesta menos saltos
            if (suma_min > acum)
            {
                suma_min = acum;
                nodo_minimo.x = j;
                nodo_minimo.y = k;
            }
        }
}
```

```
        // si el minimo es NODO_NO_MARCADO, no hay manera de que todos los caballos lleguen a la misma pocicion
        if (suma_min >= NODO_MARCADO)
                cout << "no" << endl;
        else
                cout << nodo_minimo.x + 1 << " " << nodo_minimo.y + 1 << " " << suma_min << endl;
        return 0;
}
```

5.3. Ej3.cpp

```
struct arista
{
        int e1;
        int e2;
        int costo;
};

struct distancia
{
        int nodoOrigen ;
        int costo;
};
struct resultado
{
        int costoTotal;
         list <arista> anillo;
         list <arista> resto;
        bool conexo;

};

resultado solucion(vector<arista> enlaces,int cantEnlaces,int cantEquipos ){
        //matriz de adyacencias
        vector< vector<int> > adyacencias (cantEquipos);
        vector< vector<int> > agm (cantEquipos);
        vector< vector<int> > aux (cantEquipos);
         list <arista> anillo;
         list <arista> resto;
        vector< distancia > distancias (cantEquipos);
        vector<bool> estan(cantEquipos);
         list  <arista> restoEnlaces;
        int iteraciones =0;
        int costoTotal = 0;
        int posActual=0;

        //pongo la matriz de adyacencias en -1 que indica que no hay conexion
        for(int i=0;i<cantEquipos;i++){
                adyacencias[i].resize(cantEquipos);
                agm[i].resize(cantEquipos);
                for(int j=0;j<cantEquipos;j++){
                        adyacencias[i][j] = -1;
                        agm[i][j] = -1;

                }
        }

        //Completo con las adyacencias del vector que viene como entrada
        for(int i=0;i<cantEnlaces;i++){
                adyacencias[enlaces[i].e1][enlaces[i].e2] = enlaces[i].costo;
                adyacencias[enlaces[i].e2][enlaces[i].e1] = enlaces[i].costo;

        }

        aux = adyacencias;
        stack<int> esConexo;
        esConexo.push(0);
        int contador = 1;

        for(int i = 0;i<cantEquipos;i++){
                estan[i]=false;
        }
        estan[0]=true;

        while (!esConexo.empty()){
                int topAnt = esConexo.top();
                for (int i = 0; i<cantEquipos; i++){
```

```
                if (aux[topAnt][i]!=-1 && !estan[i]){
                        esConexo.push(i);
                        contador++;
                        estan[i]=true;
                        aux[topAnt][i] = -1;
                        aux[i][topAnt] = -1;
                        break;
                }
        }
        if (topAnt == esConexo.top()){
                esConexo.pop();
        }
}

if (contador != cantEquipos || cantEnlaces< cantEquipos){
        resultado res;
        res.conexo = false;
        return res;
}

// inicializo el arreglo de distancias
for(int i = 0;i<cantEquipos;i++){
        if(adyacencias[0][i] != -1){
                distancias[i].costo = adyacencias[0][i];
                distancias[i].nodoOrigen = 0;
        }else{
                distancias[i].costo = -1; //si no son adyacentes seteo -1
                distancias[i].nodoOrigen = -1;
        }
}

for(int i = 0;i<cantEquipos;i++){
        estan[i]=false;
}
estan[0]=true;

while(iteraciones < cantEquipos -1){

        int min=-1;
        int nodoMin;
        //busco el minimo en el arreglo de distancias
        for(int i =0;i<cantEquipos;i++){
                if(min == -1 || ( 0 < distancias[i].costo && distancias[i].costo< min && !estan[i])){
                        min = distancias[i].costo;
                        nodoMin = i;
                }
        }
        distancias[nodoMin].costo = -1;

        //Actualizamos la matriz de adyacencias de agm
        agm[distancias[nodoMin].nodoOrigen][nodoMin] = min;
        agm[nodoMin][distancias[nodoMin].nodoOrigen] = min;
        adyacencias[nodoMin][distancias[nodoMin].nodoOrigen] = -1;
        adyacencias[distancias[nodoMin].nodoOrigen][nodoMin] = -1;

        costoTotal += min;
        estan[nodoMin]=true;

        // actualizo mi arreglo de distancias
        for(int i=0;i<cantEquipos;i++){
                if(adyacencias[nodoMin][i]>=0)
                {
                        if(!estan[i] &&
                                (distancias[i].costo == -1 ||
                                        adyacencias[nodoMin][i]< distancias[i].costo
                                )){

                                distancias[i].costo = adyacencias[nodoMin][i];
```

```
                                distancias[i].nodoOrigen =nodoMin;

                        }

                }
        }
        iteraciones++;
}
//Busco del resto de aristas q no pertecen al agm la minima para generar un circuito

 arista aristaMin = {-1,-1,-1};


for(int i=0;i<cantEquipos;i++){
        for(int j=i+1;j<cantEquipos;j++){
                if(     aristaMin.costo == -1 ||
                        (adyacencias[i][j]!=-1 && adyacencias[i][j] < aristaMin.costo)
                  )
                {
                        aristaMin.costo = adyacencias[i][j];
                        aristaMin.e1 = i;
                        aristaMin.e2 = j;
                }
        }
}



costoTotal += aristaMin.costo;
//BUSCO EL CAMINO
stack<int> caminoActual;
caminoActual.push(aristaMin.e1);

aux = agm;
while (caminoActual.top() != aristaMin.e2){
        int topAnt = caminoActual.top();
        for(int i = 0; i<cantEquipos; i++){
                if(aux[topAnt][i]!=-1){
                        caminoActual.push(i);
                        aux[topAnt][i] = -1;
                        aux[i][topAnt] = -1;
                        break;
                }
        }
        if(topAnt == caminoActual.top()){
                caminoActual.pop();
        }
}

// CONSTRUYO EL CIRCUITO

while(!caminoActual.empty()){
        int topAnt = caminoActual.top();
        caminoActual.pop();
        if(caminoActual.empty()) break;
        arista nueva = {topAnt,caminoActual.top(),0};
        anillo.push_back(nueva);
        //limpio en la matriz del agm aquellas aristas que pertenecen al anillo
        agm[topAnt][caminoActual.top()] = -1;
        agm[caminoActual.top()][topAnt] = -1;
}
//agrego el arista que cierra el circuito
anillo.push_back(aristaMin);

//CONSTRUYO LA LISTA DE LAS ARISTAS QUE QUEDAN FUERA DEL ANILLO
for(int i =0;i<cantEquipos;i++){
        for(int j=i+1;j<cantEquipos;j++){
                if(agm[i][j]!=-1){
```

```
                                arista nueva = {i,j,0};
                                resto.push_back(nueva);
                            }
                        }
                }

                resultado res = {costoTotal, anillo, resto, true};
                return res;
}

int main(int argc, char *argv[])
{
        // para detectar cuando estoy leyendo la primer linea
        bool primerLinea = true;
        string line;
        //e1 y e2 representan el numero de los equipos
    int cantEquipos,e1,e2,costo;
    int cantEnlaces = 0;
        vector<arista> enlaces;
        int cantLeidas = -1;
    //Comienzo la lectura del archivo
    while ( getline (cin, line) ){
                vector<string> entradaSplit;
                int fromIndex = 0;//inicio string
                int length = 0; //longitud string


            if(primerLinea){
                    for(int i = 0; i<line.length(); i++ ){
                            if(i== line.length()-1){
                              length++;
                              entradaSplit.push_back(line.substr(fromIndex, length));
                            }else if(line[i]== ' '){
                              entradaSplit.push_back(line.substr(fromIndex, length));
                              fromIndex = i+1;
                              length = 0;
                            }else{
                              length++;
                            }
                        }

                    cantEquipos = atoi(entradaSplit[0].c_str());
                    cantEnlaces = atoi(entradaSplit[1].c_str());
                    primerLinea=false;
                    continue;
            }


                for(int i = 0; i<line.length(); i++ ){
                if(i== line.length()-1){
                  length++;
                  entradaSplit.push_back(line.substr(fromIndex, length));
                }else if(line[i]== ' '){
                  entradaSplit.push_back(line.substr(fromIndex, length));
                  fromIndex = i+1;
                  length = 0;
                }else{
                  length++;
                }
            }

            e1 = atoi(entradaSplit[0].c_str());
                        e2 = atoi(entradaSplit[1].c_str());
                        //SOLO PARA EMPROLIJAR INDICES RESTO 1
                        e1--;
                        e2--;

            costo = atoi(entradaSplit[2].c_str());
```

```
                    arista nueva = {e1,e2,costo};
                    enlaces.push_back(nueva);
    }

    resultado res = solucion(enlaces,cantEnlaces,cantEquipos);

    //IMPRIMO LA SALIDA
    if(res.conexo){
        cout << res.costoTotal << ' ' << res.anillo.size();
            cout << ' ' << res.resto.size() << endl;

            for (list<arista>::iterator it = res.anillo.begin(); it!=res.anillo.end(); it++){
                int e1 = (*it).e1 +1;
                int e2 = (*it).e2 +1;
                        cout << e1 << ' ' << e2 << endl;
                }
                for (list<arista>::iterator it = res.resto.begin(); it!=res.resto.end(); it++){
                        int e1 = (*it).e1 +1;
                int e2 = (*it).e2 +1;
                        cout << e1 << ' ' << e2 << endl;
                }
    }else{
        cout << "no" << endl;

    }

        return 0;
}
```