DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nro. 2

20 de noviembre de 2014

Algoritmos y Estructuras de Datos III
Trabajo Práctico Nro. 2

Integrante	LU	Correo electrónico
Pablo Gomez	156/13	mago-1986@hotmail.com
Lucia Parral	162/13	luciaparral@gmail.com
Emanuel Lamela	21/13	emanuel93_13@hotmail.com
Petr Romachov	412/13	promachov@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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) 4576-3300
http://www.exactas.uba.ar

Índice

1. Ejercicio 1

1.1. Introducción

1.1.1. Contexto

Estamos en el desarrollo de un sitio web de compra de pasajes aéreos, conocido como aterrizar.com. Nuestro sitio ofrece, entre sus otras opciones de búsqueda, una en particular que es innovadora, perfecta para impacientes/viajeros apurados, la cual consiste en que detallando el punto de origen y el punto de destino, el sistema indicará al usuario el itinerario que llega lo antes posible al destino seleccionado; todo esto el sistema lo hará siempre y cuando exista un posible itinerario que conecte ambos puntos. La solución provista puede utilizar la cantidad de tramos que sean necesarios, ya que su único objetivo es optimizar la fecha y hora de llegada a destino.

1.1.2. El problema a resolver

Nos encontramos entonces frente al desafío de desarrollar un algoritmo que se encargue de dicho itinerario, es decir, que dados los puntos de origen y destino, A y B respectivamente, y una lista de vuelos disponibles, debemos devolver un itinerario tal que vaya desde A hasta B, de la manera rápida posible. Para eso podemos combinar todos los vuelos que hagan falta pasando por cualquier ciudad intermedia en el camino. De cada vuelo de la lista de disponibles conocemos su origen y destino y las fechas y horas de partida y llegada. El algoritmo debe devolver un itinerario factible, y para ello, deben cumplirse las siguientes condiciones:

- El primer vuelo debe salir de A y el último vuelo debe terminar en B.
- Si el itinerario usa más de un vuelo, la ciudad de llegada de cada vuelo debe coincidir con la ciudad de salida del vuelo siguiente.

Además, debe haber al menos 2 horas de diferencia entre la llegada de un vuelo y la partida del vuelo siguiente para los tiempos de embarque pertinentes entre un vuelo y otro.

1.1.3. Ejemplos

1. Ciudad de origen: Buenos Aires
   Ciudad de destino: Jujuy
   Cantidad de vuelos: 10

Ciudad De Origen	Ciudad Destino	Hora Salida	Hora Llegada
Buenos Aires	La Pampa	10	12
Buenos Aires	Entre Ríos	9	10
Buenos Aires	Formosa	11	13
Formosa	Córdoba	16	17
Entre Ríos	Jujuy	12	20
Entre Ríos	La Rioja	13	15
La Rioja	Tucuman	20	21
Santiago del Estero	Jujuy	20	22
Santiago del Estero	Catamarca	23	24
La Rioja	Santiago del Estero	17	18

   En este ejemplo hay dos itinerarios que llevan de la ciudad de origen, Buenos Aires a la de destino, Jujuy, cumpliendo que entre cada vuelo haya un mínimo de dos horas de diferencia entre la llegada de uno y el despege del siguiente:

   - (Buenos Aires - Entre Ríos) - (Entre Ríos - La Rioja) - (La Rioja - Santiago del Estero) - (Santiago del Estero - Jujuy) - **Horario de llegada:** 22hs
   - (Buenos Aires - Entre Ríos) - (Entre Ríos - Jujuy) **Horario de llegada:** 20hs

   Por lo tanto la solución es el segundo itinerario, ya que tiene un horario de llegada menor.

2. Ciudad de origen: Buenos Aires
   Ciudad de destino: Jujuy
   Cantidad de vuelos: 10

Ciudad De Origen	Ciudad Destino	Hora Salida	Hora Llegada
Buenos Aires	La Pampa	10	12
Buenos Aires	Entre Ríos	9	10
Buenos Aires	Formosa	11	13
Formosa	Córdoba	16	17
Entre Ríos	Jujuy	11	20
Entre Ríos	La Rioja	13	15
La Rioja	Tucuman	20	21
Santiago del Estero	Jujuy	20	22
Santiago del Estero	Catamarca	23	24
La Rioja	Santiago del Estero	17	18

   En este ejemplo, se pierde el itinerario óptimo encontrado en el ejemplo anterior, ya que entre el primer vuelo (Buenos Aires - Entre Ríos) y el segundo (Entre Ríos - Jujuy) ya no hay un mínimo de 2 horas entre el horario de llegada de uno y el horario de partida del otro, por lo que nos queda una única solución, que es también óptima:

   (Buenos Aires - Entre Ríos) - (Entre Ríos - La Rioja) - (La Rioja - Santiago del Estero) - (Santiago del Estero - Jujuy) - **Horario de llegada:** 22hs

3. Ciudad de origen: Buenos Aires
   Ciudad de destino: Jujuy
   Cantidad de vuelos: 10

Ciudad De Origen	Ciudad Destino	Hora Salida	Hora Llegada
Buenos Aires	La Pampa	10	12
Buenos Aires	Entre Ríos	9	10
Buenos Aires	Formosa	11	13
Misiones	Córdoba	16	17
Corrientes	Jujuy	12	20
Corrientes	La Rioja	13	15
La Rioja	Tucuman	20	21
Santiago del Estero	Jujuy	20	22
Santiago del Estero	Catamarca	23	24
La Rioja	Santiago del Estero	17	18

   En este ejemplo, no hay ninguna silución, ya que si bien hay vuelos que salen desde la ciudad de origen Buenos Aires, no es posible realizar a partir del primer vuelo ninguna conexión para poder llegar a la ciudad de destino.

4. Ciudad de origen: Buenos Aires
   Ciudad de destino: Jujuy
   Cantidad de vuelos: 10

Ciudad De Origen	Ciudad Destino	Hora Salida	Hora Llegada
Buenos Aires	La Pampa	10	12
Buenos Aires	Entre Ríos	9	10
Buenos Aires	Formosa	11	13
Formosa	Córdoba	16	17
Entre Ríos	Jujuy	12	20
Entre Ríos	La Rioja	11	12
La Pampa	Tucuman	14	15
Tucumán	Jujuy	17	20
Santiago del Estero	Catamarca	23	24
La Rioja	Santiago del Estero	17	18

En este ejemplo hay dos itinerarios que son solución óptima:

- (Buenos Aires - Entre Ríos) - (Entre Ríos - La Rioja) **Horario de llegada:** 20hs
- (Buenos Aires - La Pampa) - (La Pampa - Tucumán) - (Tucumán - Jujuy) **Horario de llegada:** 20hs

5. Ciudad de origen: Buenos Aires
   Ciudad de destino: Jujuy
   Cantidad de vuelos: 10

Ciudad De Origen	Ciudad Destino	Hora Salida	Hora Llegada
Chubut	La Pampa	10	12
Río Negro	Entre Ríos	9	10
Neuquén	Formosa	11	13
Formosa	Córdoba	16	17
Entre Ríos	Jujuy	12	20
Entre Ríos	La Rioja	11	12
La Pampa	Tucuman	14	15
Tucumán	Jujuy	17	20
Santiago del Estero	Catamarca	23	24
La Rioja	Santiago del Estero	17	18

Ningún vuelo sale de la ciudad de Origen, por lo tanto, no hay solución.

1.2. Desarrollo

Para resolver el problema planteado, utilizamos la técnica de programación dinámica y un algoritmo top down, ya que presentaba las características necesarias para usarla, en tanto tiene subestructuras óptimas, es decir era posible dividir el problema en subproblemas más pequeños.

Por ejemplo, si quiero viajar de la ciudad A a la B de manera óptima, si el camino óptimo incluye ir de la ciudad X a B, y desde A es posible llegar a B, entonces el camino óptimo es el camino óptimo de A a X y de X a B.
. Finalmente, al utilizar todas estas subsoluciones óptimas de viajes de cualquier ciudad a la ciudad de destino, se puede formar la solución al problema. Para esto, almacenaremos las soluciones ya calculadas para no tener que recalcularlas al momento de volver a necesitarlas.

Teniendo esto en mente, el algoritmo ideado para resolver el problema se basa principalmente en dos etapas:

Preparación de los datos y almacenamiento en estructuras de datos:

- A: Ciudad de origen
- B: Ciudad objetivo
- n: Cantidad de vuelos
- Io: Ciudad origen del vuelo I
- Id: Ciudad destino del vuelo I
- Ip: Hora de partida del vuelo I
- If: Hora de llegada del vuelo I
- intCityMapping: diccionario<string,int>
- salidas: vector<vector<Vuelo>>

```
1   salidas.reservarLugarPosiciones(n*2 + 2)
2   Para i = 0 hasta n-1 hacer
3       Si Io no esta definida en intCityMapping        O(log n)
4         agregar Io a intCityMapping                   O(log n)
5         agrandar en 1 el tamano del vector salidas    O(1) (reserved memory)
6
7       Si Id no esta definida en intCityMapping        O(log n)
8         agregar Id a intCityMapping                   O(log n)
9         agrandar en 1 el tamano del vector salidas    O(1) (reserved memory)
10
11      IDciudadOrigen <- intCityMapping[Io]            O(log n)
12      IDciudadDestino <- intCityMapping[Id]           O(log n)
13
14      Vuelo vuelo <- (IDciudadOrigen,IDciudadDestino,Ip,If,i+1)   O(1)
15      salidas[IDciudadOrigen].agregarAtras(vuelo)     O(1) (reserved memory)
16  Fin Para
17
18  Para j = 0 hasta salidas.size()-1 hacer
19      ordenar(salidas[j])   O(x log x) (x = cantidad de vuelos que salen de j)
20  Fin Para
```

Se mapean los vuelos pasados en el input, de modo de formar una matriz en la que para cada ciudad están los vuelos que salen de ella.
Luego, cada uno de los vuelos que parten de cada ciudad, los ordenaremos según su horario de salida.
A partir de este momento, los datos están almacenados en la forma necesaria para poder aplicar el algoritmo de mejor camino con la complejidad requerida.

Cálculo del mejor itinerario:

```
1  vector<bool>  de cantidadDeCiudades posiciones, con todas inicializadas en
       bool
2  mejorCamino(int origen, int horaDeConsulta):
3    revisadoHasta <- Infinit
4
5    Si(origen == ID representativo de ciudad B)
6      cache[origen] <- (calculado, puedeLlegarAB, horaDeConsulta,horaDeConsulta)
7      return cache[origen]
8    Fin Si
9
10   Si(cache[origen] tiene un calculo igual a horaDeConsulta)
11     return cache[origen]
12   Sino
13     revisadoHasta <- cache[origen].horaDeCalculo
14   Fin Si
15
16   Si(cache[origen] tiene un calculo a una hora menor)
17     return Itinerario(calculado,hayOtroOptimo, Infinit, horaDeCalculo)
18   Fin Si
19
20   ////////Variables
21   itinerario <- Itinerario(calculado,noLlegahastaB,Infinit,horaDeConsulta)
22   from <- 0
23   esNecesarioSeguirRevisando <- true
24   Si(vuelosDeSalida[origen].tam > 0)
25     from <- customBinarySearch(hora, vuelosDeSalida[origen])
26
27   //Ciclo
28   para i = from  hasta  vuelosDeSalida[origen].tam-1 hacer
29     vuelo := vuelosDeSalida[origen][i]
30
31     Si(disponibles[vuelo.destino])
32       Si(best[origen].calculado && revisadoHasta+2 <= vuelo.inicio)
33           potencial = best[origen];
34           esNecesarioSeguirRevisando <- false;
35       Sino
36         disponibles[origen] <- false    //Apago la ciudad.
37         recursivo <- mejorCamino(vuelo.destino, vuelo.fin);
38       Fin Si
39
40       Si(recursivo.puedeLlegarAB && recursivo.llegada < itinerario.llegada)
41         itinerario<-(calculado,puedeLlegarAB,horaDeConsulta,recursivo.llegada)
42       Fin Si
43
44       Si(!esNecesarioSeguirRevisando)
45         break
46       Fin Si
47
48     fin si
49
50     disponibles[origen] <- true   //Activo la ciudad de nuevo
51   fin para
52
53   best[origen] = itinerario
54   return best[origen]
55 fin mejorCamino
```

Llamaremos:

- Vuelo a la clase que contiene Origen (ciudad desde donde parte), Destino (ciudad a la cual se quiere llegar), hora de salida del vuelo y hora de llegada.
- Ciuda_Origen a la ciudad desde la cual se parte en la instancia pasada.
- Ciuda_Destino a la ciudad a la cual se quiere llegar.

Primero chequeamos si el mejor vuelo para la Ciuda_Origen ya fue calculado para el horario en el que se esta haciendo el llamado recursivo. Si es así, lo devuelvemos.

Si fue calculado en un horario menor:

- si desde ese horario no se había podido llegar a la Ciuda_Destino, entonces en un horario mayor tampoco se podrá, por lo que devolvemos que no es posible llegar.
- si desde ese horario si se pudo llegar, entonces sabemos que hay otro vuelo en el que se llega a la Ciuda_Destino más rápido o igual, por lo que no es necesario seguir haciendo más llamados recursivos.

Si el origen del vuelo que estamos mirando es la Ciuda_Destino, entonces guardamos la Ciuda_Destino en un itinerario que informa que se puede llegar y que la mejor forma de llegar a la Ciuda_Destino ya fue calculada y la devolvemos.

Si estamos en cualquier otro escenario, realizaremos una búsqueda binaria customizada de los vuelos de salida de la ciudad en la que estamos para posicionarnos justo en el primer vuelo que podríamos tomar en el horario actual y recorreremos a partir de ahí linealmente hasta que lleguemos a un horario que ya fue calculado anteriormente. De no haberlo, recorreremos hasta el final.

Para evitar realizar un llamado recursivo con una ciudad de la que ya revisé sus vuelos en un llamado recursivo anterior durante la búsqueda de este camino, decidimos utilizar el vector de ciudades disponibles, en el que indicamos si en el llamado recursivo actual es válido o no revisar una ciudad. Por ejemplo:

Si tomo un vuelo desde A a C, voy a hacer el llamado recursivo con C y revisar los vuelos que salen de C. Si uno de esos vuelos vuelve a A, llamémoslo vuel_i, no tiene sentido que lo tome, ya que resultaba lo mismo no tomar el vuelo a C y esperar en A y tomar directamente desde allí el vuel_i en un horario posterior. De esta forma, evitamos que se formen estos ciclos de vuelos que van y vuelven a una misma ciudad en un mismo llamado recursivo.

De esta forma, si la ciudad está marcada como disponible, podemos utilizarla para el llamado recursivo. Como primer paso, la marcaremos como no disponible para evitar cálculos de vuelos que vuelvan hacia la ciudad que estamos mirando, y a partir de ahí, se realiza el llamado recusivo con el destino del vuelo elegido, tratando de encontrar la mejor forma de llegar hasta la Ciuda_Destino desde la ciudad donde aterrice el vuelo.

Si el vuelo que conseguimos en la recursión es mejor que el mínimo actual, lo pisamos. De esta forma, la variable *itinerario* siempre contiene el que tardó menos y después de revisaitinerarior todos los vuelos de salida, si se puede llegar a B, contiene el que en menor tiempo lo hace. Caso contrario, queda indicado que no se puede llegar.

Luego de terminar de recorrer los vuelos pertinentes se guarda en la matriz best el valor conseguido y se retorna dicha posición de la matriz *best*.

Luego, se reconstruye con la información obtenida cuales los vuelos que nos llevarán hasta B desde A de poder hacerlo.

Por lo explicado, se corrobora que este algoritmo realiza los pasos de programación dinámica, ya que:

1. Divide el problema en subproblemas más pequeños, ya que, de no encontrar la solución requerida almacenada previamente, llama recursivamente a la función con un vuelo menos.
2. Resuelve estos subproblemas de manera óptima para el algoritmo, usando este proceso recursivamente, ya que realiza llamados recursivos hasta llegar a uno de los casos bases detallados previamente.
3. Utiliza estas subsoluciones óptimas para construir una solución óptima al problema original, ya que almacena los resultados parciales por medio de la técnica de memoización.

1.3. Complejidad

El algoritmo propuesto tiene una complejidad temporal de O(n log n). Para demostrar esta complejidad, será conveniente y más prolijo dividir al algoritmo en 2 submódulos claramente independientes.

1. *Recepción de datos y almacenamiento en estructura:* ***O(n log n)***
2. *Cálculo de las subsoluciones del problema y almacenamiento en una estructura caché:* ***O(n log n)***

Lo cual da una complejidad temporal asintótica total de ***O(n log n)***.

El resto del programa no se analizará ya que no se toman en cuenta los tiempos de escritura en la salida estandar del sistema, que es la última parte del programa.

Para los 2 submódulos será interesante notar los siguientes tips:

- La cantidad de ciudades distintas que habrá en una instancia del problema es del orden de n ya que a lo sumo habrá 2 + 2*n ciudades. Dicho eso, realizar acciones de complejidad T en cada ciudad distinta tiene complejidad O(n * T).
- Para una mejor organización se definieron los ***struct Vuelo*** y ***struct Itinerario***. La complejidad de hacer una copia de alguno de estos 2 es O(1) puesto que están construidos con tipos que no son colecciones.

Las complejidades de las clases y métodos utilizados de la STL de C++ se detallan a continuación:

map - C++

- **constructor** Constante, O(1)
- **cout** O(k log k) con k cantidad de claves definidas
- **operator[ ]** O(k log k) con k cantidad de claves definidas
- **size** Constante, O(1)

vector - C++

- **constructor** Constante, O(1)
- **operator[ ]** Constante, O(1)
- **push_back** Constante, O(1), si se reserva memoria primero.
- **size** Constante, O(1)

algorithm - C++

- **stable_sort** O(k log k) con k cantidad de posiciones del arreglo a ordenar

Para el submódulo 2 se definió una búsqueda binaria customizada:
Dicha búsqueda, antes de comenzar, realiza un chequeo previo; el mismo consiste en comprobar si el elemento búscado es más grande que el último del vector, o más chico que el primero.
Si se cumple alguna de estas 2 condiciones, como el vector presenta acceso en O(1) y está ordenado, la búsqueda se resuelve en O(1).
De lo contrario, si el valor búscado b cumple que *vector[0] <= b <= vector[vector.size() - 1]*, entonces la complejidad de la búsqueda es como la de una búsqueda binaria convencional: *O(log vector.size())*

Sin más, comenzamos con el primer submódulo:

- **Submódulo 1/2: Recepción de datos y almacenamiento en estructuras: *O(n log n)***

Pseudocódigo del submódulo 1

- Io: Ciudad origen del vuelo I
- Id: Ciudad destino del vuelo I
- Ip: Hora de partida del vuelo I
- If: Hora de llegada del vuelo I
- intCityMapping: diccionario<string,int>
- salidas: vector<vector<Vuelo>>

```
1   salidas.reservarLugarPosiciones(n*2 + 2)
2   Para i = 0 hasta n-1 hacer
3       Si Io no esta definida en intCityMapping        O(log n)
4         agregar Io a intCityMapping                   O(log n)
5         agrandar en 1 el tamano del vector salidas    O(1) (reserved memory)
6
7       Si Id no esta definida en intCityMapping        O(log n)
8         agregar Id a intCityMapping                   O(log n)
9         agrandar en 1 el tamano del vector salidas    O(1) (reserved memory)
10
11      IDciudadOrigen <- intCityMapping[Io]            O(log n)
12      IDciudadDestino <- intCityMapping[Id]           O(log n)
13
14      Vuelo vuelo <- (IDciudadOrigen,IDciudadDestino,Ip,If,i+1)   O(1)
15      salidas[IDciudadOrigen].agregarAtras(vuelo)     O(1) (reserved memory)
16  Fin Para
17
18  Para j = 0 hasta salidas.size()-1 hacer
19      ordenar(salidas[j])   O(x log x) (x = cantidad de vuelos que salen de j)
20  Fin Para
```

Análisis del primer *para* (línea 2):

El peor caso de una iteración será cuando Io e Id no hayan sido previamente mapeadas. La consulta de mapeo, el mapeo en sí y estirar el vector de vuelos de salida tienen complejidad *O(2 * log n + 1) = O(log n).*
Hacer esto 2 veces es *O(2 * log n)* que nuevamente es ***O(log n)***

Luego se genera el nuevo struct vuelo con los ID de las ciudades del vuelo que se está iterando actualmente. Conseguir dichos IDs tiene costo *O(2 * log n)* = ***O(log n)***

Una vez construido el vuelo, se accedé a la posicion correspondiente a la ciudad de origen en el vector de salidas y se agrega el mismo.

- **Acceder a posicion correspondiente:** O(1)
- **Copiar vuelo:** O(1)
- **AgregarAtras:** O(1) ya que es memoria reservada

Lo cual nos deja afirmar que el peor caso de una iteración tiene un costo acotado superiormente por:
O(log n) *(mapeo)* + ***O(log n)*** *(creacion y carga del struct vuelo)* = ***O(log n)***

Observando que el *para* en análisis cicla n veces, su complejidad es de: ***O(n log n)***

Análisis del segundo *para* (línea 18):
Para el costo del segundo *para* es interesante notar una propiedad, y es que:

Para v0,v1...vi vectores, si se cumple que:

$$\sum_{j=0}^{i} v_j.size() = n$$

Entonces ordenar todos los arreglos con un algoritmo *O(l log l)* (l longitud de colección a ordenar) tiene una cota superior de ***O(n log n)***. ***(Demostrado al final del submódulo 2)***

Como el vector de salidas contiene en cada posicion conjuntos de vuelos disjuntos con los de otra posición, y como la cantidad de vuelos es igual a n, entonces, por la propiedad mostrada, ordenar todas las posiciones del vector *salidas* tiene el mismo funcionamiento que el descrito en la propiedad.
Luego, la complejidad del segundo *para* es: ***O(n log n)***

Concluyendo la cota superior del submódulo 1 (recepción de datos y almacenamiento en estructuras)
Mapear ciudades y guardar los datos en las estrucutas: ***O(n log n)***
Ordenar todos los arreglos del vector de salidas ***O(n log n)***

Complejidad total del submódulo 1: O(n log n)

- **Submódulo 2/2: Cálculo de las subsoluciones del problema y almacenamiento en una estructura caché: *O(n log n)***

Pseudocódigo del submódulo 2

```
disponibles <- vector<bool>(cantidadDeCiudadesDistintas,true)

mejorCamino(int origen, int horaDeConsulta):
  revisadoHasta <- Infinit

  Si(origen == ID representativo de ciudad B)
    cache[origen] <- (calculado, puedeLlegarAB, horaDeConsulta,horaDeConsulta)
    return cache[origen]
  Fin Si

  Si(cache[origen] tiene un calculo igual a horaDeConsulta)
    return cache[origen]
  Sino
    revisadoHasta <- cache[origen].horaDeCalculo
  Fin Si

  Si(cache[origen] tiene un calculo a una hora menor)
    return Itinerario(calculado,hayOtroOptimo, Infinit, horaDeCalculo)
  Fin Si

  ////////Variables
  itinerario <- Itinerario(calculado,noLlegahastaB,Infinit,horaDeConsulta)
  from <- 0
  esNecesarioSeguirRevisando <- true
  Si(vuelosDeSalida[origen].tam > 0)
    from <- customBinarySearch(hora, vuelosDeSalida[origen])

  //Ciclo
  para i = from  hasta  vuelosDeSalida[origen].tam-1 hacer
    vuelo := vuelosDeSalida[origen][i]

    Si(disponibles[vuelo.destino])
      Si(best[origen].calculado && revisadoHasta+2 <= vuelo.inicio)
          potencial = best[origen];
          esNecesarioSeguirRevisando <- false;
      Sino
        disponibles[origen] <- false    //Apago la ciudad.
        recursivo <- mejorCamino(vuelo.destino, vuelo.fin);
      Fin Si

      Si(recursivo.puedeLlegarAB && recursivo.llegada < itinerario.llegada)
        itinerario<-(calculado,puedeLlegarAB,horaDeConsulta,recursivo.llegada)
      Fin Si

      Si(!esNecesarioSeguirRevisando)
        break
      Fin Si

    fin si

    disponibles[origen] <- true   //Activo la ciudad de nuevo
  fin para

  best[origen] = itinerario
  return best[origen]
fin mejorCamino
```

Podemos dividir al submódulo en 2 claros casos. Los casos base y los casos que posiblemente lleven a una recursividad. Analizaremos la complejidad particular de cada uno, y luego, en conjunto.

Casos base: [Línea 3 - Línea 24]
Caso recursivos: [Línea 26 - Línea 45]

Casos base
Tenemos 3 casos base, que son

- Querer la mejor manera de llegar a B, y eso ya fue calculado a una hora menor o igual a la hora de consulta
- Querer la mejor manera de llegar a B, desde B.
- Querer llegar hasta B, desde una ciudad que no posee vuelos de partida

Para el primero caso (líneas 3-18), por lo dicho al principio de la sección, el costo de copiar un *Itinerario* es O(1). Dicho eso y como la matriz *caché* es un vector, su acceso y lectura también son en O(1), por lo que las preguntas realizadas para el primer caso base tienen costo O(1).

En cuanto al segundo caso base (líneas 20-24), nuevamente, por acceso O(1) a la matriz *caché*, la guarda, la asignacion y el return tienen costo O(1) cada uno, por lo que todo el segundo caso base se resuelve en O(1).

El tercer caso basé es el caso en el que se intenta acceder al for pero no se tiene exito ya que la cantidad de vuelos que parten de la ciudad dada es 0. Por ende, este caso base realiza dicha verificación, una asignación a variable (línea 26), una asignación a la memoria *caché* para dejar guardado el calculo, y un return. Nuevamente todos se resuelven en O(1) por lo que este caso base también tiene costo O(1).

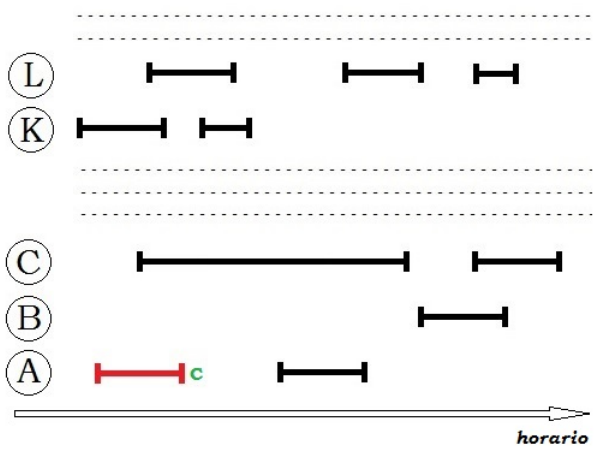
Habiendo calculado los 3 casos bases por separado, la complejidad total de peor caso, si se cae en algún caso base, es: O(1).

Casos recursivos

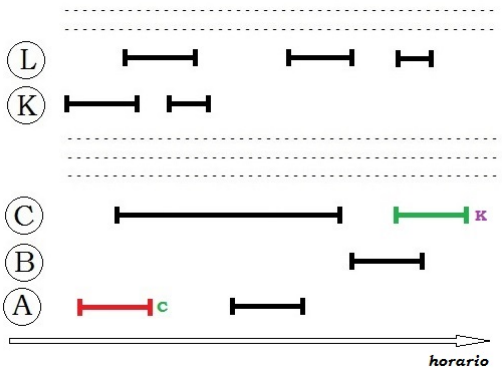
Este caso tiene un *para* en el que se hacen consultas y asignaciones de tiempo constante. Existe un llamado recursivo que será analizado más adelante, pero para el resto de las instrucciones, todas se realizan en una complejidad O(1), lo cual hace que el cuerpo del *para* tenga complejidad O(1).
Como el *para* itera sobre los vuelos que salen de una ciudad dada (origen), a lo sumo realizará n ciclos.

Para el caso de los llamados recursivos, debido a que nunca se hacen llamados recursivos sobre vuelos que vuelven a ciudades por las que estamos calculando, la cantidad de vuelos que salen de una ciudad a la que llegamos por un llamado recursivo es <= (n - vuelos que salen de la ciudad origen antes del llamado recursivo).
Dicho eso y debido también a que nunca se consulta por el horario de un vuelo 2 veces (ya que la búsqueda binaria impide eso), la complejidad total es, en peor caso O(n log n). Viendolo con el siguientes ejemplo, que gráfica vuelos:

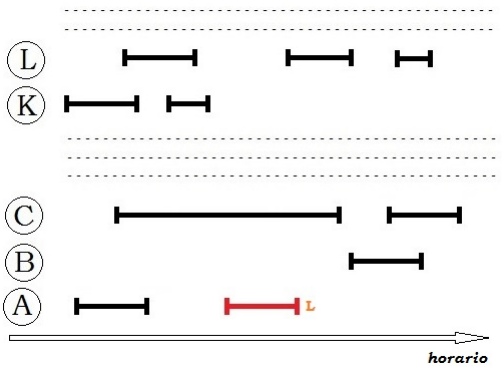
Realizando la búsqueda binaria de vuelos de salida de A para posicionarnos en el primero que podríamos tomar, consultamos que sea de una ciudad disponible, y como no estamos calculando nada en C (destino del vuelo), es una ciudad disponible y llamamos recursivamente:



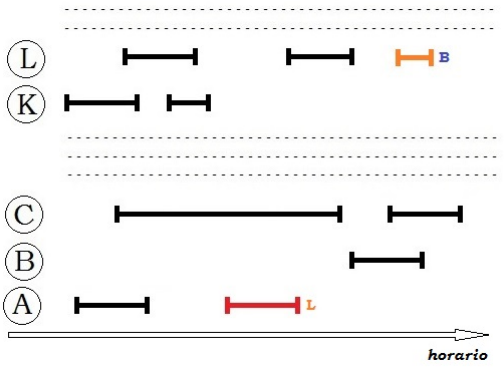
Ahora, posicionados en C, realizar la búsqueda binaria de vuelos de salida de C hará que no consultemos vuelos que obviamente no podemos tomar (como el primer vuelo largo de C), haciendo que nos concentremos en el siguiente.



Luego, dicho vuelos llevará nuestra atención a la ciudad K
Como nuestra busqueda binaria tiene 2 verificadores previos (explicado al inicio de la sección de complejidad), para el caso de ejemplo la búsqueda de vuelo se resuelve en O(1), viendo que no hay vuelos disponibles a la hora en la que llegamos a K.
Eso hace que volvamos a C para buscar otro vuelo, y a falta de otro para verificar, concluímos que a la hora de llegada a C (horario de llegada del primer vuelo de A) no es posible llegar hasta B (nuestro objetivo), por lo que guardamos en caché y procedemos a buscar otro vuelo en A.



Nuevamente, recursivamente llamamos a la ciudad J (ciudad disponible), y con búsqueda binaria, tomamos el primer vuelo posible.



Al tener ese vuelo como destino a B, guardamos en caché['L'] que desde L es posible llegar a B al horario de llegada del segundo vuelo de A, para no recalacularlo, y volvemos a A buscando otro vuelo que talvés nos termine llevando a B más temprano; como no lo hay, finaliza.

Si para el segundo vuelo de A, también hubieramos llegado a C, debido a que en cada ciudad se revisan los vuelos solo desde los que se pueden tomar hasta el ultimo sin revisar, **no hubieramos re-calculado el vuelo que se estaría tentado de calcular**, ya que la matriz caché indica que hay un calculo ya hecho a esa hora y simplemente basta con leerlo.
Ahora, si el segundo vuelo de A hubiera llegado lo suficientemente temprano como para tomar otro posible potencial vuelo de C, caché['C'] *pisaría* su valor para indicar la nueva franja horaria desde la que ya se conocen cálculos.

Dicho eso, el peor de los casos, debido a sucesivos llamados recursivos podría acotarse superiormente por n búsquedas binarias, es decir, el peor caso del segundo submódulo tendría una cota superior ***O(n log n)***.

Notar que: la complejidad recién señalada, sugiere que se realizarían n búsquedas binarias (una por cada vuelo) de costo *O(log n)* cada una, lo cual solamente pasaría si en todas las ciudades donde un vuelo llegara hubiera n vuelos de salida, lo cual es *absurdo* ya que hay n vuelos en total y no hay vuelos que salgan y lleguen a la misma ciudad.
El único caso en el que se realizarían n búsquedas binarias sería que cada ciudad tenga solamente un vuelo de salida, pero, de darse ese caso, las mencionadas n búsquedas binarias se resolverían en O(1) cada una ya que el vector de vuelos de salida de cada ciudad tendría longitud = 1, resultando una complejidad *O(n * 1) = O(n).*
Por lo que ***O(n log n)*** es una buena cota superior para el submódulo 2 de todo el programa.

Análisis de mejor caso del submódulo 2
Como el algoritmo se resuelve con la forma top-down, al comenzar mirando los vuelos de salida de A, esto hace que el mejor caso de este submódulo sea que **no salgan vuelos desde A**, teniendo una cota superior (de mejor caso) ***O(1)***.

Habiendo demostrado la complejidad de los 2 submódulos, la complejidad TOTAL del algoritmo es:

Complejidad submódulo 1 + Complejidad submódulo 2 =

O(n log n) + O(n log n) =

O(n log n)

Demostración propiedad: *Complejidad Ordenar subarreglos = Complejidad Ordenar arreglo entero*

Sean *X, Y* y *Z* arreglos tal que X.size() + Y.size() + Z.size() = n.
Sea *tamI* el tamaño de un arreglo I. (Ej: tamX = X.size())
Sea sort() un algoritmo que ordena colecciónes tal que la complejidad de ordenamiento es O(coleccion.size() * log coleccion.size()) cuando el acceso a una posición de la colección se realiza en O(1).

Entonces, ordenar X, Y y Z con *sort()* toma:

O(tamX * log tamX) + O(tamY * log tamY) + O(tamZ * log tamZ)

Como tamX + tamY + tamZ = n, entonces:

(tamX * log tamX) <= (n * log n)
(tamY * log tamY) <= (n * log n)
(tamZ * log tamZ) <= (n * log n)

Luego, podemos acotar la complejidad mencionada anteriormente:

O(n log n) + O(n log n) + O(n log n) = *3**O(n log n) = O(n log n)*

Quedando así demostrado que ordenar k subarreglos tal que la suma de sus longitudes es n, tiene una complejidad **O(n log n)**.

1.4. Experimentación

Para el proceso de experimentación del problema se plantearon distintas pruebas para corroborar que el algoritmo propuesto funcionara correctamente, y que la cota de complejidad encontrada y justificada en la sección anterior, se cumpliera en la práctica.

Dado que el CPU de la computadora utilizada para tomar los tiempos no está atendiendo únicamente a nuestro proceso, realizar una sola vez cada prueba podría darnos valores que no son cercanos a los reales. Por lo tanto, para minimizar este margen de error, a cada prueba se la hizo ejecutar un total de 100 veces, y se tomó el mejor valor, es decir, el menor tiempo de ejecución obtenido. Notar que, tomar el mejor valor no es una mala decisión, ya que cuanto más chico sea el valor, más cerca estamos del valor real de tiempo que toma el algoritmo para una instancia dada.

En cada prueba, se tomaron métricas para la posterior evaluación del algoritmo en la práctica. Vale aclarar que la medición no contempla tiempos de salida de datos, sino que contempla:

1. La preparación de los datos de entrada para su procesamiento
2. El algoritmo que determina el mejor itinerario posible.

Para el testeo, se diseñó un generador de instancias aleatorias que toma dos parámetros:

1. **Cantidad de ciudades:** determina cuántas ciudades se tomarán aleatoriamente que puedan ser origen o destino de los vuelos, seteando por lo menos un vuelo a cada una o desde cada una para que sean ciudades con sentido.
2. **La cantidad de vuelos presentes en la instancia**.

Con este programa pudimos evaluar cuánto tiempo de ejecución toma nuestro algoritmo para distintas instancias aleatorias del problema.

Para todos los casos, se eligió una precisión de hasta 0,0001 ms (milisegundos). De ser menor, la tomamos como 0.

También desarrollamos un programa similar al explicado anteriormente, pero que arroja únicamente instancias de mejor caso.

Finalmente, el proceso de testing es:

1. Generación de instancia aleatoria, según parámetros prefijados.
2. Ejecución de dicha instancia 100 veces, tomando el mejor tiempo obtenido.
3. Repetición de los items 1 y 2 otras 99 veces y obtención del tiempo promedio.

Con esta metodología de experimentación, realizamos pruebas en dos tipos de escenarios:

- Escenarios de casos aleatorios
- Escenarios de mejor caso

A continuación, describiremos las particularidades de cada escenario.

1.4.1. Escenarios de casos aleatorios

En estos escenarios, decidimos evaluar casos aleatorios, generados por el generador de instancias aleatorias descripto anteriormente.

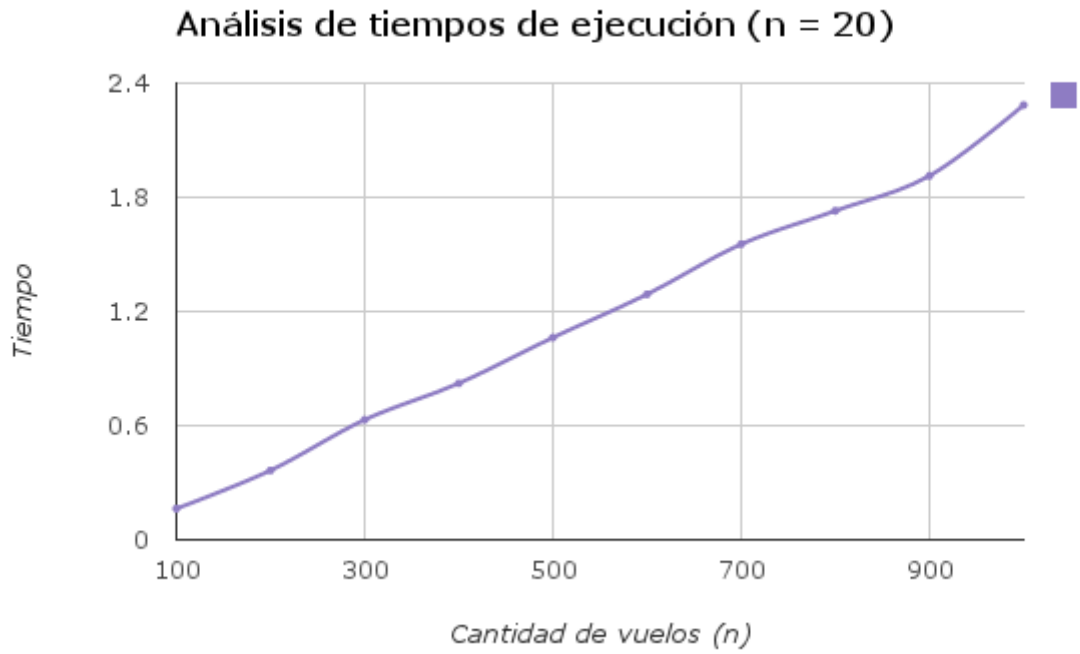
La idea fue tomar muestras del algoritmo haciendo variar la cantidad de vuelos, generando instancias que den origen y destino aleatorio a los mismos.

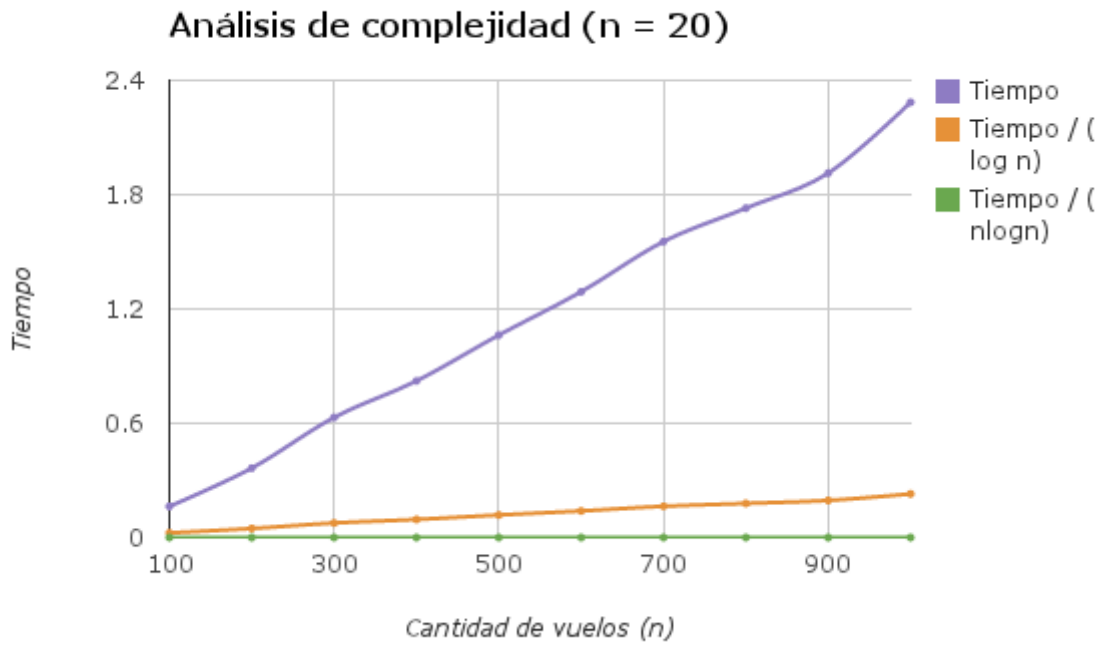
Para poder diferenciar bien los casos y poder analizar mejor, decidimos que cada escenario de las pruebas de caso aleatorio tenga una cantidad de ciudades constante. Esta capacidad de ciudades fue prefijada de manera que, si establecemos una cantidad m de ciudades, haya por lo menos un vuelo que salga o llegue a cada una, para que en la instancia haya por lo menos m ciudades activas.

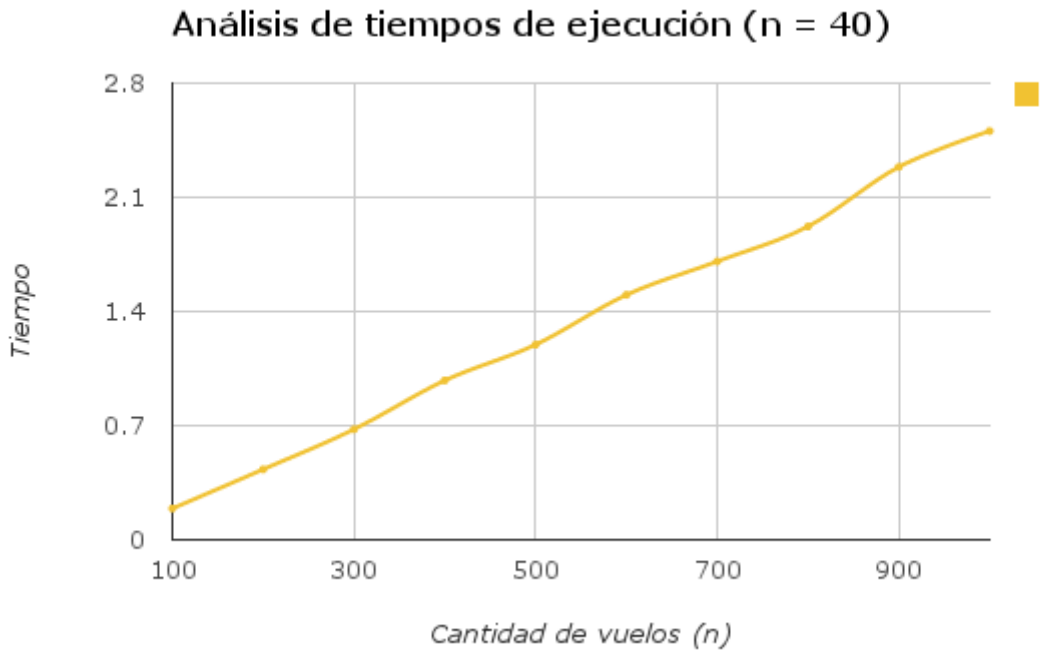
A continuación, los gráficos resultantes de la experimentación en estos escenarios. Para los mismos, variamos la cantidad de vuelos aumentándolos de a 100 y elegimos probar con 20, 40 y 80 ciudades.

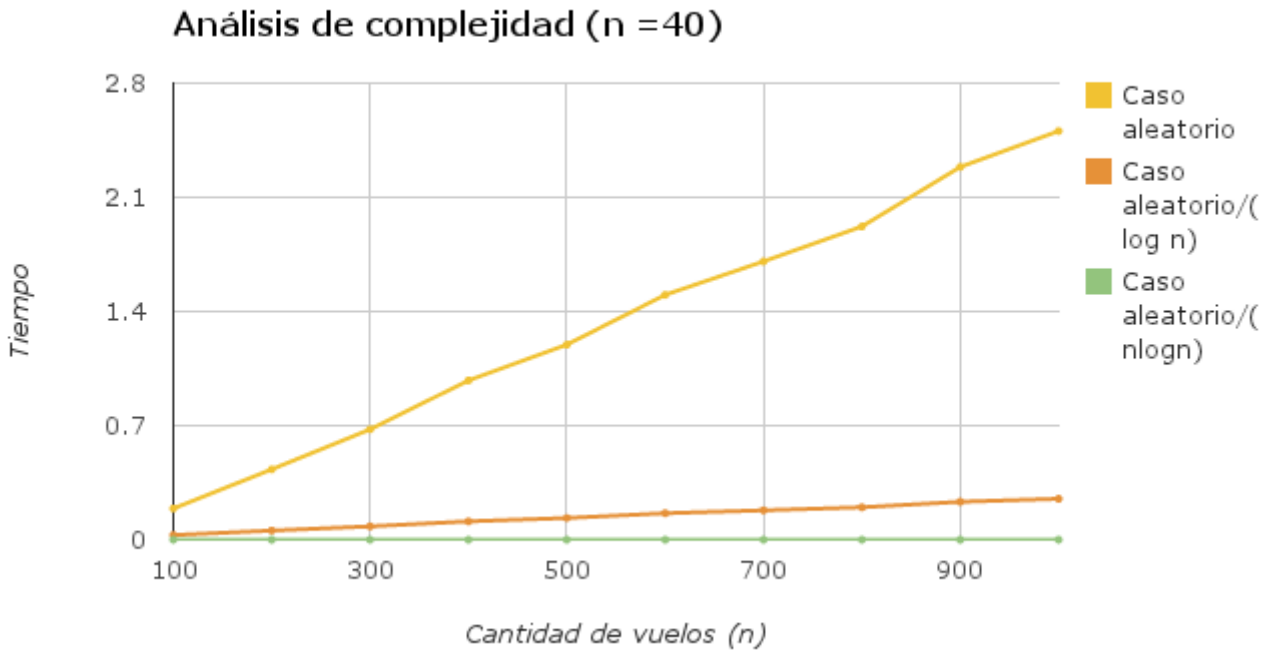
Para cada una de las pruebas, mostraremos la tabla con los valores obtenidos, los análisis de tiempos de ejecución y un análisis gráfico para estimar la complejidad del algoritmo.

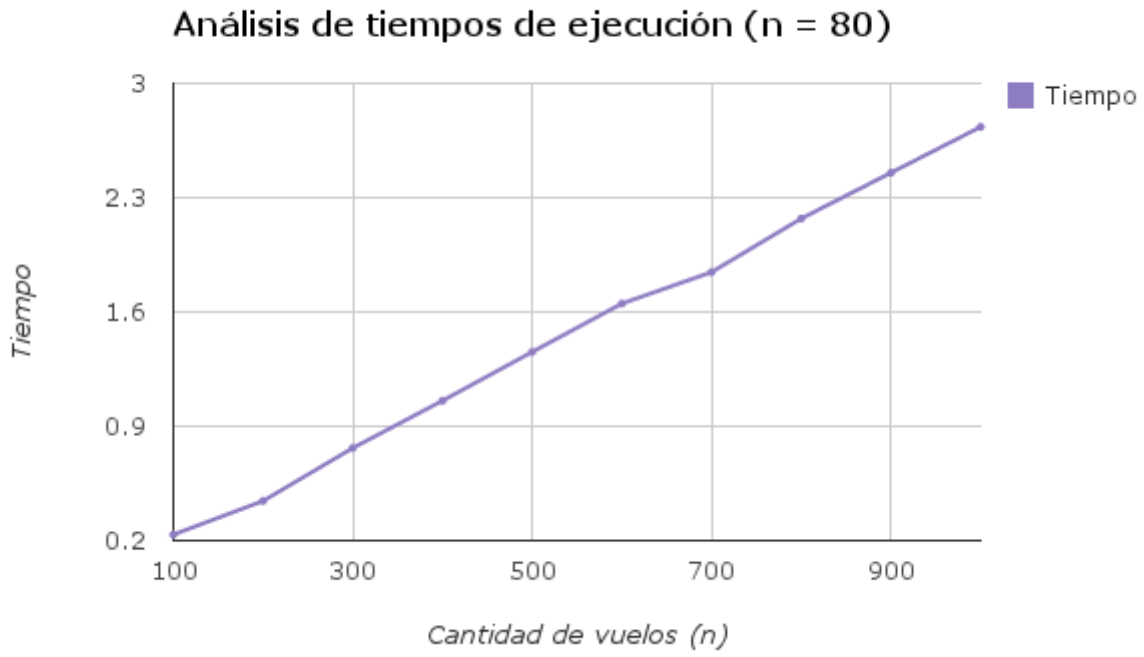
Aclaración: Por simplicidad, nos referiremos de ahora en más con ***n*** *a la* ***Cantidad de vuelos****.*

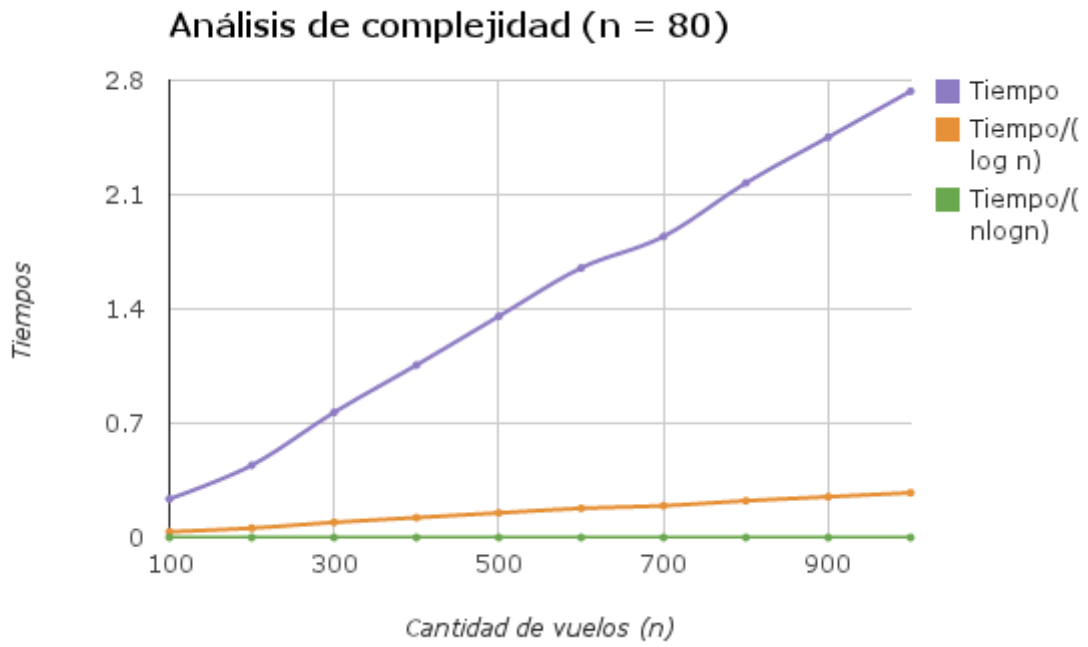



Análisis de complejidad (n = 20)
2.4
1.8
1.2
0.6
0
Tiempo
100
300
500
700
900
Cantidad de vuelos (n)
Tiempo
Tiempo / (log n)
Tiempo / (nlogn)


Análisis de tiempos de ejecución (n = 40)
2.8
2.1
1.4
0.7
0
Tiempo
100
300
500
700
900
Cantidad de vuelos (n)

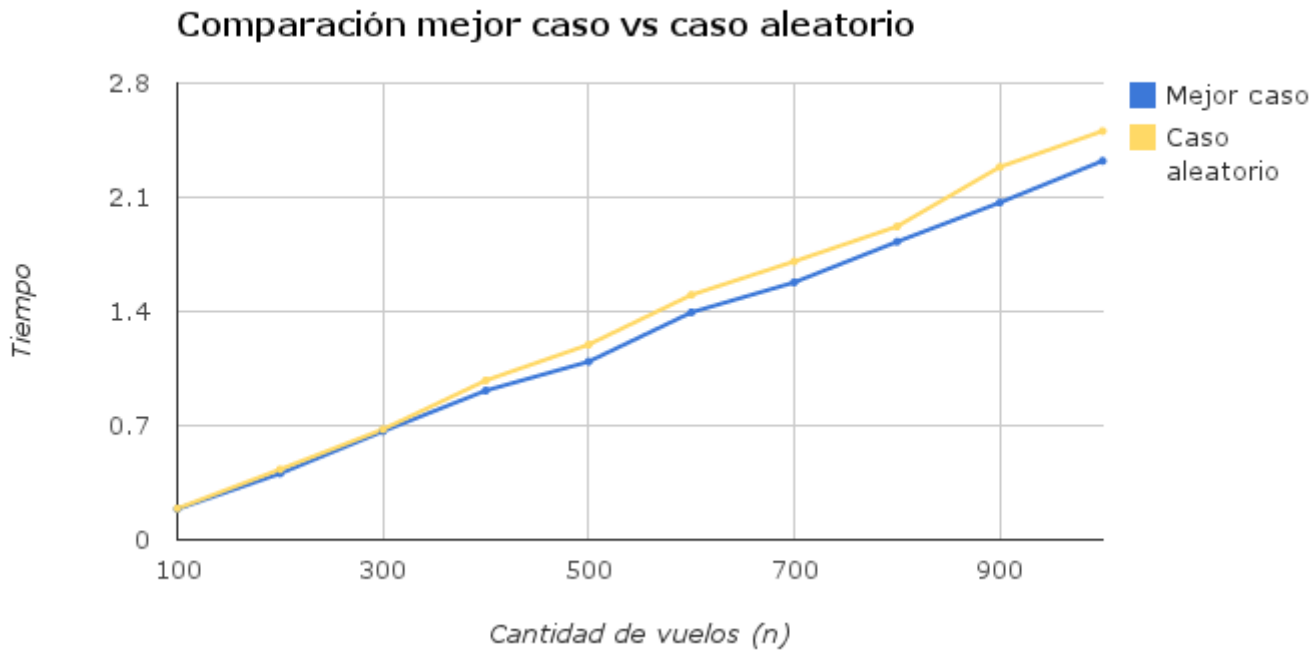

Análisis de complejidad (n =40)
2.8
2.1
1.4
0.7
0
Tiempo
100
300
500
700
900
Cantidad de vuelos (n)
Caso aleatorio
Caso aleatorio/(log n)
Caso aleatorio/(nlogn)

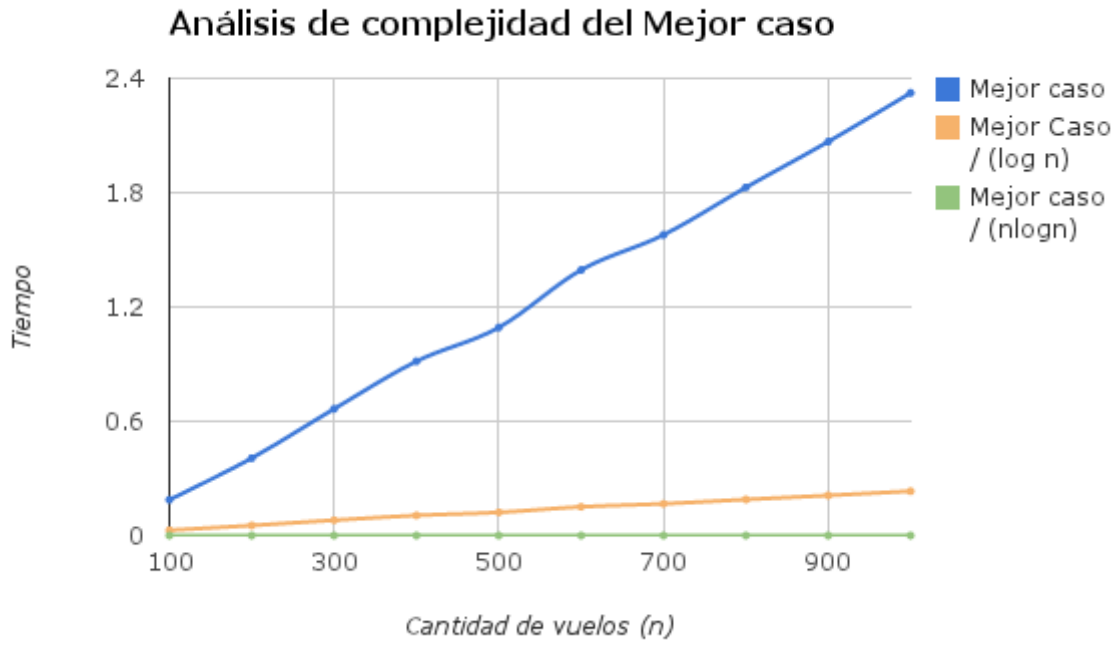

Análisis de tiempos de ejecución (n = 80)
3
2.3
1.6
0.9
0.2
Tiempo
100
300
500
700
900
Cantidad de vuelos (n)
Tiempo



1.4.2. Escenario de mejor caso

En este escenario, realizamos una experimentación análoga a la detallada anteriormente: generamos instancias de mejor caso, haciendo variar la cantidad de vuelos aumentándolos de a 100, y dejando fija la cantidad de ciudades, de la misma forma en que se explicó en el escenario anterior. Sin embargo, en este caso, únicamente realizamos la experimentación considerando 40 ciudades.





1.4.3. Conclusiones

Aclaración: En esta sección continuamos refiriéndonos con ***n*** *a la* ***Cantidad de vuelos****.*

Corroboramos lo analizado analíticamente, verificando en lo empírico que la complejidad del algoritmo propuesto es O(n log n).
En los gráficos de complejidad presentados en los distintos escenarios vemos claramente que al dividir punto f(n) de la función por log n nos queda una función lineal, y al hacerlo por n log n una constante. Esto solo podria pasar si efectivamente la curva que obtenemos al graficar los tiempos medidos está acotada por O(n log n).
Comprobamos además que a partir de la comparación de los gráficos obtenidos con instancias de 20, 40 y 80 ciudades, que la medición de tiempos, arroja para cualquiera de los tres escenarios curvas acotadas por O(n log n), por lo que corroboramos lo que analizamos desde lo analítico: la complejidad temporal del algoritmo no reside en la cantidad de ciudades que se usen para realizar los vuelos, si no en la cantidad de los mismos. Esto se debe a lo ya enunciado en el apartado de complejidad, donde expusimos que la cantidad de ciudades es del orden de n, ya que a lo sumo hay 2.n + 2 ciudades.
Por otro lado, en tanto al caso propuesto como mejor, observamos que arroja para cada n una medición de tiempo mejor que para una instancia de mismo n, pero de caso aleatorio. Sin embargo, este mejor caso solo impacta el cálculo de mejor camino, y no así la preparación de los datos, por ende esta curva obtenida a partir de la experimentación con instancias de mejor caso es también acotada por O(n log n), solo que con mejores tiempos debido a constantes de multiplicación menores.
De esta forma, corroboramos lo analizado analíticamente.

2. Ejercicio 2

2.1. Introducción

Contexto

Estamos frente a un nuevo juego de mesa, que es una alternativa al ajedrez tradicional. Se usa un tablero de ajedrez especial, que puede tomar dimensiones de $n \times n$ casillas, y solo se juega con caballos blancos. El unico movimiento permitido es desplazar dichos caballos, de la forma tradicional del de ajedrez (o sea en L), y con una regla adicional que permite que varios caballos puedan compartir la misma ubicacion.

Los caballos se colocan al inicio del juego en determinadas posiciones, las cuales son arbitrarias, y se pueden llegar a dar casos donde hay mas de un caballo inicialmente en la misma ubicacion. El objetivo del juego es ir moviendo los caballos, y lograr ubicar todos los caballos en un solo lugar usando la menor cantidad de movimientos posibles.

El problema a resolver

Debemos diseñar un algoritmo, que dado el tamaño del tablero n, la cantidad de caballos k y las posiciones f c (fila y columna respectivamente) iniciales de estos dentro del tablero, calcule el mejor casillero para elegir ubicar a dichos caballos, de tal manera que minimice la cantidad de movimientos realizados.

El algoritmo debe devolver el casillero elegido y la cantidad de saltos realizados en total, en caso de no tener solucion, simplemente devolver la palabra **no**. Este algoritmo debera tener una complejidad temporal de a lo sumo $O(k \times n^2)$.

Ejemplos

1. n = 3, k = 3

   Fila1 = 1 Columna1 = 2
   Fila2 = 3 Columna2 = 1
   Fila3 = 3 Columna3 = 3

   Resultado: 1 2 2

2. n = 3, k = 3

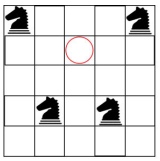
   Fila1 = 1 Columna1 = 1
   Fila2 = 2 Columna2 = 3
   Fila3 = 2 Columna3 = 2

   Resultado: no

3. n = 5, k = 4

   Fila1 = 1 Columna1 = 1
   Fila2 = 4 Columna2 = 2
   Fila3 = 1 Columna3 = 5
   Fila4 = 4 Columna4 = 4

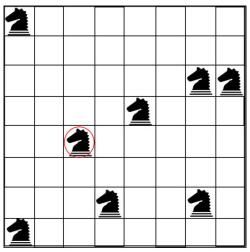
   Resultado: 2 3 4



4. n = 8, k = 8

   Fila1 = 1 Columna1 = 1
   Fila2 = 7 Columna2 = 4
   Fila3 = 3 Columna3 = 7
   Fila4 = 5 Columna4 = 3
   Fila5 = 3 Columna5 = 8
   FIla6 = 4 Columna6 = 5
   Fila7 = 8 Columna7 = 1
   Fila8 = 7 Columna8 = 7

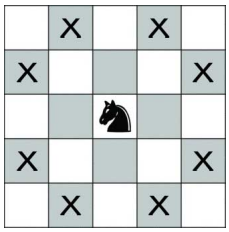
   Resultado: 5 3 14



2.2. Desarrollo

Para resolver el problema presentado, se analizaron diferentes propiedades:

Se puede interpretar al tablero como un grafo, con sus nodos las coordenadas, y sus ejes los desplazamientos que puede realizar un caballo. Dado que sabe que las opciones de desplazamiento de un caballo es como maximo 8, entonces la cantidad de ejes por nodo va a ser a lo sumo 8.



Se puede ver ademas que para el problema se pide una complejidad no superior a $O(k \cdot n^2)$, siendo k la cantidad de caballos y n^2 el tamaño del tablero, lo cual da una sospecha que para cada caballo se tengan que realizar operaciones con complejidad $O(n^2)$.

Algunas definiciones globales a partir de ahora:

1. **Tablero**, **Matriz** y **Grafo** se van a referir a lo mismo, al tablero asociado al problema.
2. **Caballo**, **Coordenada**, **Posicion**, **Nodo**, y todos sus sinonimos en grafos van a significar lo mismo.
3. **Salto**, **Desplazamiento**, **Eje**, y todos sus sinonimos en grafos van a significar lo mismo.
4. **Distancia** a la longitud del camino mas corto de una posicion a otra, es decir, su definicion en grafos.

Se propone entonces, una funcion que calcule la matriz de distancias para cada caballo, la cual en cada posicion del tablero, va a tener su distancia del caballo.

Matriz de Distancias

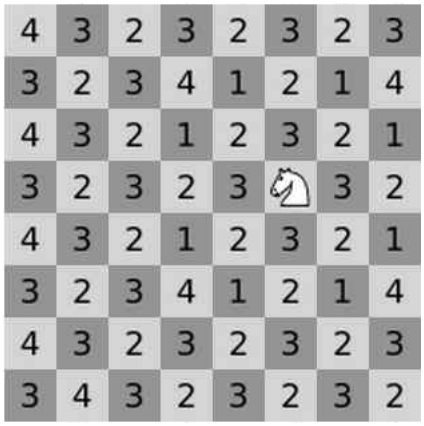
Esta función sirve para calcular el camino mínimo entre una posición del tablero inicial, y todas las que estan alrededor, sin embargo tiene funcionalidad limitada ya que en términos de este ejercicio, solo hace falta la distancia mínima y no el camino mínimo.

Se recibe una matriz con $n \times n$ casilleros inicializados con la distancia -1, para indicar que aun no se los revisaron. Le asignamos a la matriz en la posición del caballo la distancia 0, debido a que es de la cual se arranca, y se agrega dicha posición a una cola, en la cual se van a guardar todos los nodos pendientes para procesar.

Por cada nodo perteneciente a esta cola, va a desencolarlo y análizar todos sus nodos adyacentes:

- Si aun no fueron analizados, se les escribe la distancia del nodo actual mas uno, debido a que requiere un salto mas para llegar hasta ahi, y se lo guarda en la cola.
- Si ya fueron analizados, se revisa si la nueva distancia es menor que la que ya tenía, en caso positivo se la actualiza.

Como este análisis involucraría todas las posiciones, en principio va a tener una complejidad de $\Omega(n^2)$, en la sección de complejidad se verá que esto de hecho es $\Theta(n^2)$.



```
void MatrizDistancias(Tablero& tablero, Coordenada caballo)

  tablero[caballo.x][caballo.y] = 0
  Cola<Coordenada> cola
  cola.encolar(caballo)

  Mientras cola tenga elementos

    Coordenada v = cola.desencolar()

    Para toda coordenada u adyacente a v
      Si tablero[u.x][u.y] == -1
        tablero[u.x][u.y] = tablero[v.x][v.y] +1
        cola.enconlar(u)
      Fin si
    Fin Para

  Fin Mientras
Fin
```

Resolución General

Se genera para cada caballo una matriz de $n \times n$ casillas, inicializado todas sus posiciones en -1 para indicar que aun no se revisaron, y se pasa dicha matriz a una funcion que calcula la matriz de distancias asociada al caballo.

Se suman todas estas matrices en una sola, y se busca el minimo valor dentro de ella. Este minimo valor va a ser la suma de todas los saltos requeridos combinados de los caballos para llegar hasta el, que es lo que el algoritmo debe resolver.

Una manera de mejorar la complejidad espacial es mantener una matriz cache para ir guardando la suma acumulada de las sumas hasta ese momento.

Para el caso donde es con tableros con n <= 3, debido a que no se puede llegar a todos los casilleros dada una posicion de un caballo, se los revisa independientemente. Como estos casos son limitados, no aportan a la complejidad.

```
Tablero cache <- Crear matriz de tamano n x n con valor 0

Para todo c en caballos
  Tablero distancias <- Crear matriz de tamano n x n con valor -1
  MatrizDistancias(distancias, c)
  cache += distancias
Fin Para

Buscar en el tablero cache la coordenada con el minimo valor
```

El pseudocódigo puede diferir levemente del código actual, pero se lo hizo asi para que sea mas entendible y se omitieron algunos detalles innecesarios, para consultarlos, ver el código.

2.3. Correctitud

En este apartado veremos que la solución presentada para resolver el ejercicio, es correcta. Empecemos por analizar la correctitud de la funcion que calcula la matriz de distancias, dado un punto de origen, tiene que calcular la longitud minima(distancia) de salto para cada posicion en el tablero.

Matriz de Distancias

En esta parte se usa el algoritmo de BFS (Breadth-First Search), adaptado para que vaya escribiendo la distancia a cada nodo a medida que va recorriendolos. Esto usa fuertemente que el peso de todas las saltos es de 1, la idea intuitiva es que debido a que recorre a lo ancho el grafo asociado al tablero, y que el peso de las aristas es siempre 1, por cada iteración va recorriendo en forma circular el grafo, ampliando el radio cada vez que los el valor de los nodos en la cola cambia, hasta abarcar a todos los nodos (que no hayan mas nodos sin marcar). Entonces como el peso de los saltos es de uno, el anterior obligatoriamente tiene que estar en el circulo del radio actual -1.

Se puede ver facilmente en el algoritmo que la cola que se usa va a estar siempre ordenada de menor a mayor, ya que primero se va a encolar los nodos adyacentes al origen (valor 1), luego a los adyacentes sin analizar de los anteriores (valor 2), y asi sucesivamente. Llamaremos a partir de ahora entonces iteración a cada vez que se acaba determinado valor en la cola, o sea que aumenta el radio de busqueda.

Ejemplo del recorrido del algoritmo:

1. Escribe 0 en la posicion origen
2. 1era iteración: Escribe en todos los adyacentes al origen el valor 1, ya que ninguno fue recorrido aun
3. 2da iteración: Escribe en todos los adyacentes a los nodos de valor 1 que aun no fueron analizados el valor del nodo actual +1 (o sea 2)
4. 3ra iteración: Escribe en todos los adyacentes a los nodos de valor 2 que aun no fueron analizados el valor del nodo actual +1 (o sea 3)
5. k-esíma iteración: Escribe en todos los adyacentes a los nodos de valor k-1 que aun no fueron analizados el valor del nodo +1 (o sea k)

Se puede ver entonces que en cada paso, efectivamente se esta poniendo la distancia mínima en todos los nodos al nodo origen. Pero esto aun no esta formalizado.

Lema/Invariante:

Para la k-esíma iteración, para obtener la longitud minima de los nodos k al punto de origen, el predecesor directo de estos es algun nodo de la iteración k-1.

Demostremos esto por el absurdo, dado un nodo **v** en la k-esíma iteración, existe **u** nodo predecesor con distancia menor a los nodos pertenecientes a **K-1**, siendo **K-1** el conjunto de nodos analizados de la k-1esíma iteración. Esto implicaría que **u** pertenezca a **Q**, siendo **Q** el conjunto de nodos aún no analizados.

Según nuestro algoritmo:

- La unica manera de alcanzar **u** es en alguna iteracion >= k, ya que sino este pertenecería al conjunto **K-1**.
- El valor de esta iteración va a ser k + c , para algún c >= 0.
- En cada iteración, los nodos que se analizan quedan con el valor del numero de iteración.
- Por los dos puntos anteriores, valor de los nodos de la iteración k + c va a ser k + c.

Entonces queda:

$$d(u) < k \iff k + c < k \text{ ¡Absurdo!}$$

El absurdo provino de suponer que existe algún nodo con valor menor a k en los nodos aún no analizados, por lo tanto el predecesor obligatoriamente tiene que pertenecer a los nodos ya analizados de la K-1esíma iteración. Estos nodos siempre existen ya que desde la iteración 1 existe el nodo origen con distancia 0.

Entonces si se hace esto desde 0 hasta k, queda que todos los nodos de valor <= k tienen la longitud mínima al punto de origen, quedando demostrado el lema.

Teorema/Invariante y terminación:

Todos los nodos tienen la mínima longitud al punto de origen. Esto seria el invariante cuando alcanza la última iteración básicamente, cuando ya no hay mas nodos sin analizar.

Con esto queda demostrada la correctitud de la funcion que calcula la matriz de distancias.

Correctitud General

Gracias a lo probado anteriormente, se ve facilmente que sumar todas las matrices de distancia correspondientes a cada caballo, nos va a dar una suma de matrices en la cual va a estar la cantidad de saltos mínima necesaria a realizar para llegar a cada casillero. Entonces lo que se hace es buscar el mínimo en esta matriz, y devolver su posicion y valor.

Siguen existiendo casos particulares en los que puede que no haya solución aplicando esta estrategia, estos son los que tienen tamaño de matriz con n <= 3, a partir de n > 3 con cualquier caballo se puede llegar a cualquier posición.

- n == 3, se revisa que si tiene caballos en el medio, que esten todos ahí, sino no tiene solución.
- n == 2, se revisa que si tiene caballos en alguna posición, que esten todos ahí, sino no tiene solución.
- n == 1, caso trivial, siempre tiene solucion y es única.

Entonces quedan abarcados y demostrados todos los casos que se pueden presentar, quedando demostrada la correctitud del algoritmo.

2.4. Complejidad

Para analizar la complejidad del algoritmo, se va a analizar primero la funcion Matriz de Distancias, para luego poder ver la complejidad del algoritmo completo.

Matriz de Distancias

Se recibe una matriz con n^2 posiciones, todas con el valor -1. Se le asigna 0 a la posicion original y se lo agrega a una cola. Luego se analiza cada posicion en la cola: se revisa todas las adyacencias a dicha posicion, que en el peor caso son 8, ya que un caballo como maximo tiene 8 posibles posiciones para saltar, quedando que este analisis cuesta $\Theta(1)$.

Dicho analisis lo hace una y solo una vez para cada posicion en el tablero (n^2), esto es debido a que las posiciones ya revisadas no se vuelven a repetir: va eligiendo las posiciones que estan marcadas con -1 para encolarlas y desmarcarlas, luego no vuelven a valer mas -1. La cola que se usa es una cola lineal, por lo tanto tiene complejidad constante para el encolado y desencolado.

Entonces este algoritmo analiza n^2 posiciones, y como dicho analisis cuesta $\Theta(1)$, la complejidad temporal de la funcion que calcula la matriz de distancias queda $\Theta(n^2)$

Complejidad General

Realicemos un analisis de cada paso:

- Crear un tablero cache de n^2 posiciones con el valor 0: $\Theta(n^2)$
- Para cada caballo, se crea un tablero de n^2 posiciones con el valor -1, se calcula su matriz de distancias asociada, y luego se la suma al tablero cache: $\Theta(k \times n^2)$
- Buscar al minimo en el tablero cache con todas las sumas: $\Theta(n^2)$

Sumando todas las complejidades, queda entonces que este algoritmo tiene una complejidad temporal final de $\Theta(k \times n^2)$

Para el caso en el que el tamaño del tablero sea menor o igual a 3 × 3, queda absorbido por la complejidad general ya que lo resuelve en complejidad temporal lineal respecto de k.

2.5. Experimentación

Para el proceso de experimentación del problema se plantearon distintos escenarios de test para corroborar que el algoritmo propuesto funcionara correctamente y que la cota de complejidad encontrada y justificada en la sección anterior, en la práctica, se cumpliera.

En cuanto a qué casos testear, nuestro algoritmo no presenta casos "border". Es decir, no tiene un peor/mejor caso, sino que para cualquier instancia cargada, realizará el mismo procedimiento, ya que las posiciones de los caballos no afecta el tiempo de computo del algoritmo, tomando $\Theta(k \times n^2)$.

En cada prueba se tomaron métricas para la posterior evaluación del algoritmo en la práctica. Notar que la medición no contempla tiempos de entrada/salida de datos, sino que contempla solamente el núcleo del algoritmo.

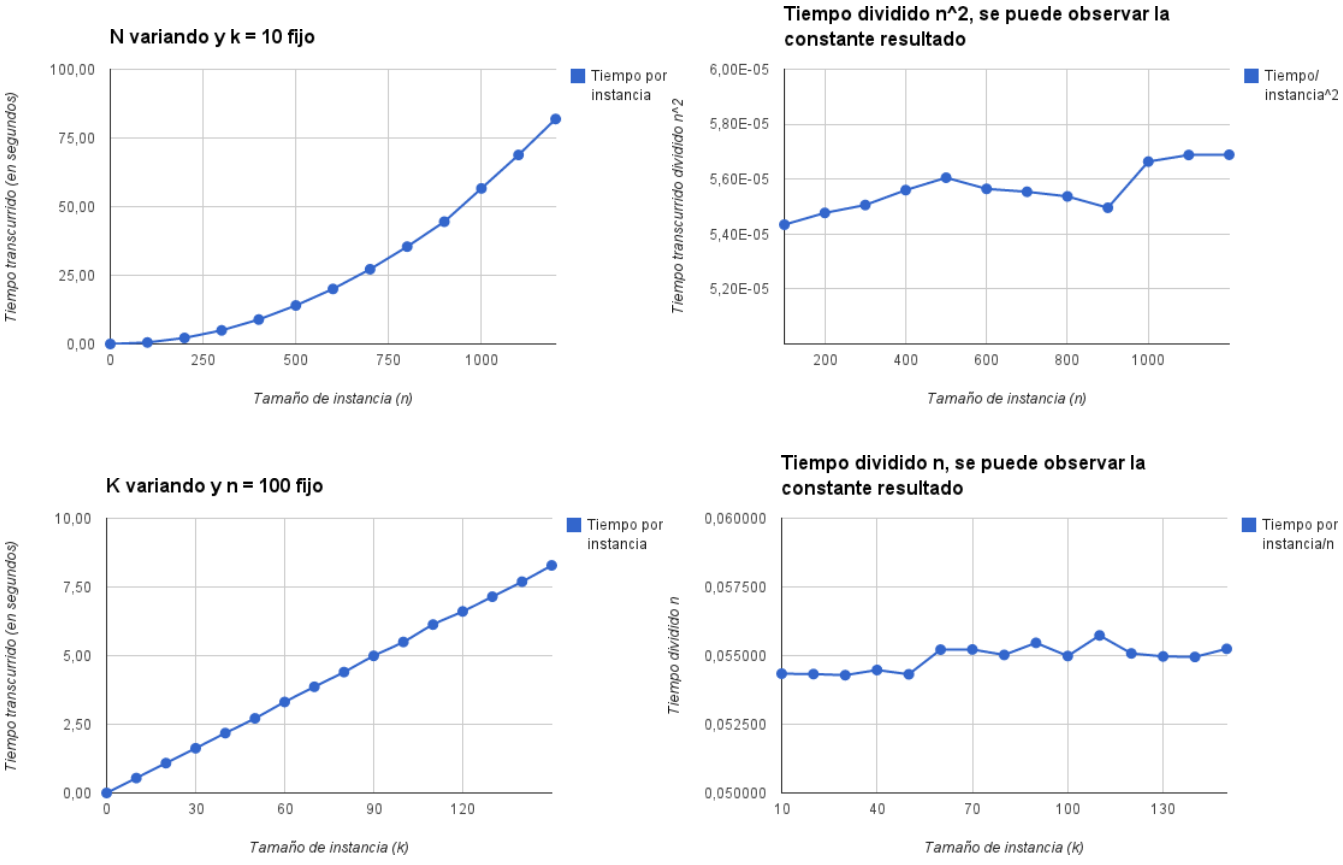
Se diseñó un programa que dado un numero i de instancias, n de edificios y k de caballos, genera i instancias aleatorias con tableros de tamaño n x n, y k caballos aleatorios para probar el algoritmo.

Para cada instancia experimentada, se la hizo ejecutar 100 veces y se tomo el valor mínimo, esto es debido a que dado que el CPU de la computadora utilizada para tomar los tiempos no está atendiendo únicamente a nuestro proceso, realizar una sola vez cada prueba podría darnos valores que no son cercanos a los reales, entonces para minimizar el márgen de error, se realiza este procedimiento. Notar que tomar el mejor mínimo no es una mala decisión, ya que mientras más chico sea, más cerca estamos del valor real de tiempo que toma el algoritmo para una instancia dada.

Explicado esto, para cada tamaño de entrada se realizaron 100 pruebas con instancias aleatorias distintas, las cuales devuelven el mejor valor, de las cuales se calcula el promedio de estos 100 valores.

Se realizaron experimentos primero fijando el tamaño del k y variando el n, y viceversa, para poder detectar la dependencia temporal del algoritmo sobre cada variable

En todos los casos se pudo comprobar que la práctica refleja lo expuesto en incisos anteriores. A continuación presentamos los gráficos 2D que reflejan las pruebas realizadas.



Los gráficos de la izquierda son las mediciones que se hicieron, y los de la derecha son los de la izquierda divididos por n^2 y k, respectivamente, formando una oscilacion de una recta. Se puede entonces ver, que los experimentos reflejan lo demostrado en la teoria.

3. Ejercicio 3

3.1. Introducción

Contexto

Una empresa dedicada a redes, llamada *AlgoNET*, nos contrató para armar un algoritmo que, dada una red ya montada, nos diga cuál sería la forma menos costosa de manejar estas conexiones existentes. Mantener una conexión tiene su costo debido a la gran demanda de ancho de banda que se necesita. El sistema debe contar con una particularidad, ciertas equipos deberían formar un servidor con forma de 'anillo'. La idea de este servidor es que la mayoría del traspaso y distribución de datos dentro de esta red la realicen ellos. Para ello, todas las máquinas deben tener al una conexión directa o indirecta (a través de otros equipos) al anillo. Diseñar la red con esta idea busca disminuir el costo de la red y que la probabilidad de que no se puedan mandar datos de un equipo a otro, en el caso de que uno o más equipos (dependiendo cuáles sean) estén dañados o malfuncionando, se imposibilite sea baja. Cabe aclarar que, al no poder agregar conexiones nuevas, hay varios escenarios en los cuales no se puede llevar a cabo esto: cuando la red no posee anillos y/o no todos los equipos poseen un camino de paso de datos a todos los pertenecientes a la red.

El problema a resolver

Dada la información que nos brinda la empresa sobre: la cantidad de equipos, los enlaces establecidos (por ende los disponibles) y el costo que representa cada uno de ellos; nuestro trabajo consta de proveer un algoritmo que describa una red que:

- Todo equipo en la red presentada debe aparecer en la descripta por el algoritmo.
- Debe contar con, al menos, 1 servidor, de 3 o más computadoras.
- Cada equipo debe estar enlazado con el servidor, ya sea directamente con uno de los componentes o a través de otros.
- Las conexiones a utilizar son las dadas, no se pueden generar conexiones nuevas.
- El costo total de las conexiones debe ser mínima. No debería poder organizar las conexiones, con la topología pedida, de forma tal que el costo sea estrictamente menor.

Para ello debe explicitar el costo total de la solución, la cantidad de enlaces del 'anillo', la cantidad de enlaces al servidor, qué enlaces se usaron para el anillo, y, finalmente, qué enlaces se utilizaron fuera del anillo, de orden estrictamente mejor que $O(n^3)$, siendo n la cantidad de equipos.

Ejemplos

1. 

2. 

3. 

3.2. Desarrollo

Preliminares

Primero que todo, para poder resolver el problema, tenemos que abstraernos de lo que es una red y encontrar una estructura donde podamos almacenar los datos de una manera relevante para facilitarnos la resolución. La estructura elegida es un ***grafo no dirigido***, representado por una ***matriz de adyacencia pesada***. Si la matriz de adyacencia A representa al grafo $G = (V, X)$, entonces la posición $A_{v1,v2} = A_{v1,v2} = x_{v1,v2} = x_{v1,v2}$, con $v1, v2 \in V$ y, $x_{v1,v2} \in X$ y es el peso de la arista que une a v1 y v2. En caso de que no exista arista que una ciertos 2 vértices ($p, q \in V$) del grafo, adoptamos que $A_{p,q} = (-1)$, como es el caso de $A_{p,p}$. Habiendo dicho esto, ***cada vértice representa un equipo*** que sea parte de la red, y ***las aristas reemplazan al costo de mantener una conexión entre 2 equipos***.

Idea

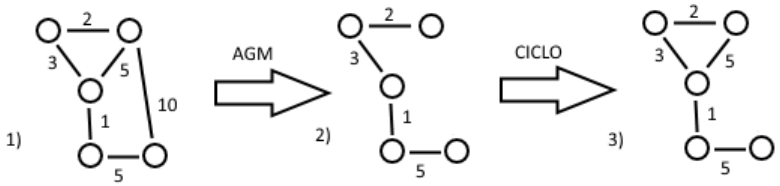
Volviendo a la resolución del trabajo, teniendo un grafo podemos aprovechar las útiles propiedades que posee, por ende de ahora en más no vamos a hablar más de red, servidor, equipos y costos sino de grafo, ciclo, vértices y peso de aristas respectivamente. Desde este punto de vista, el grafo que hay que armar es un ***subgrafo generador***, ***conexo*** con ***un sólo ciclo*** de ***costo mínimo***. Si uno quita la condición de que tenga al menos un ciclo, lo que nos queda es un grafo generador, conexo y sin ciclos de costo mínimo, en otras palabras un ***árbol generador mínimo***. Esto no resuelve el problema, pero está muy cerca ya que si obtenemos el AGM nos garantizamos un grafo sin ciclos de peso mínimo. Por ser un árbol, sabemos que con agregarle una arista pierde su propiedad árborea, ya que pasaría a tener exactamente un ciclo. Sin embargo, la elección de esa conexión a mantener no puede ser arbitraria para poder llevar a cabo nuestra tarea, ***debe ser una de las aristas de peso mínimo que no se encuentre colocada en el AGM que ya tenemos***. Si no se elige la menos costosa, siempre se puede armar una red casi equivalente, pero que posea esa conexión más barata, para formar el servidor, que no es la que elejimos. Por ende, el resultado final consta de un grafo generador y conexo con un sólo ciclo que respeta la condición de costo total mínimo. En la sección de correctitud se demostrará como este proceso genera efectivamente una solución óptima.

Ilustración

Para concretar la idea, lo que se realiza, en rasgos generales, es lo siguiente:

1. Toma de datos de la red para armar el grafo subyacente.
2. Si se puede armar el AGM en base a dicho grafo, se lo arma y se continúa al siguiente paso. Si no, indicamos que no es posible resolver la tarea.
3. Si hay al menos un enlace disponible que no haya sido utilizado en el paso 2), se busca y coloca la conexión de menor peso que no esté presente en el AGM obtenido en el paso 2).

Para ilustrar el proceso:



Algoritmos usados y detalles importantes

Dentro de los algoritmos usados para la resolución, hay 2 de importancia: el algoritmo de prim, arma un AGM posible para un grafo input; una versión modificada al algoritmo de BFS que, dado un AGM y 2 nodos pertenecientes a él, devuelve la información del camino entre ambos para obtener los equipos pertenecientes al ciclo. Esto se requiere para dar formato correcto al output. Tanto los pseudo-códigos de ambos como de las sub-rutinas utilizadas se encuentran en la sección de complejidad para facilitar el análisis de ella.

3.3. Correctitud

Objetivo

Lo que queremos demostrar es que una manera de encontrar ***una solución óptima*** es ***construyendo un AGM subyacente*** y luego ***colocarle la arista más liviana de las quue no fueron colocadas en el AGM***, efectivamente formando un ciclo.

Características de una solución óptima

Sabemos que una solución, tal vez no óptima, debe tener forma de ***sub-grafo generador con uno o más ciclos***. Llamando a la red G, supongamos que llamamos a la solución S. ***Llamamos S_i a sub-grafo de S tal que le quitamos i aristas que forman ciclos***, de tal forma que S_i sigue siendo conexo, donde $0 \leq$ i $\leq$ (cantidad de aristas que forman ciclos en S). Para simplificar las cosas, $k =$ (cantidad de aristas que forman ciclos en S) $= m-(n-1)$. Necesariamente $m > (n-1)$ debido a que un sub-grafo generador debe tener al menos $(n-1)$ aristas y además tiene 1 o más ciclos. Como estamos quitando conexiones, sabemos que $p(S_0) \leq p(S_1) \leq ... \leq p(S_k)$, donde la función $p(R)$ nos indica el peso total para un grafo cualquiera R. En otras palabras, la solución es igual o mejor si tiene menor cantidad de ciclos. La razón por la cuál la relación de desigualdades no es estricta, es porque puede haber aristas de costo 0 que formen ciclos, y si las quitamos no altera el costo del grafo. A cualquier solución se le puede hacer esta reducción de aristas, ya que sabemos que tienen 1 o más ciclos, de forma tal que abarcamos todo el espectro de soluciones. La solución minimal que podríamos armar en cada caso es un S_{k-1}. ***Lo que queremos probar es que para toda solución S, con $S_{k-1} = H$, se cumple que, dada nuestra solución propuesta T, $p(T) \leq p(H)$***. En términos menos formales, que la resolución planteada genera una red óptima.

Demostración

Supongamos que no es cierta la relación anterior: ***existe alguna solución S tal que su H, que es el resultado de sacar $k-1$ aristas que forman ciclos de S, tal que $p(T) > p(H)$***. Queremos llegar a algo absurdo que pruebe que lo postulado es es incorrecto, probando que lo contrario es cierto. Tomamos e, la arista de mayor peso en el ciclo de T con $T' = T - e'$, además T' es AGM ya que le quitamos a nuestra solución la única arista que formaba ciclos, lo que nos queda es el AGM armado. Teniendo en cuenta esto, separamos en 2 casos:

I. El ciclo de T y el ciclo de H tienen alguna arista en común.

II. Sus ciclos no tienen aristas en común.

Caso I: Tomamos e' de H, una de las que tienen en común. Tenemos que $H' = H - e'$ que es un Àrbol Generador. Sabemos que $p(e) \leq p(e')$, sino tomaríamos $T* = T' - e + e'$ que sería un árbol generador con menor peso que T', pero T' es AGM y eso es absurdo. Por ende, $p(e) + p(T) \leq p(e') + p(H') \iff p(T) \leq p(H)$, pero habíamos asumido lo contrario. Llegamos a un absurdo.
Caso II: Tomamos $e*$ cualquier arista del ciclo de H y la quito, entonces tengo $H* = H - e*$. De nuevo, $H*$ es un árbol generador entonces $p(T') \leq p(H*) \iff p(T') + p(e) \leq p(H*) + p(e*) \iff p(T) \leq p(H)$, pero habíamos asumido lo contrario. Llegamos a un absurdo.

Como son 2 casos disjuntos y nuestra solución S es genérica, probamos que es cierto para todo caso. Con lo cuál probamos lo que habíamos postulado.

Comentarios Adicionales

- Somos conscientes que a la demostración le falta formalidad. Estuvimos ajustados de tiempo ya que no se nos ocurriá como hacerla.
- La función p está sobrecargada, se ajusta a lo que le pases como parámetro devolviendo el peso total de un grafo o de una arista dependiendo el caso.

3.4. Complejidad

Aclaraciones pertinentes

1. Algunas definiciones de tipo: peso es int, nodo es int, grafo es vector<vector<peso>>, pesoPredecesor es pair<peso,nodo>, vectorPesoPredecesor es vector<pesoPredecesor>, parNodo es pair<nodo,nodo>, tuplaSolución es pair<grafo, bool>, tuplaCiclo es pair<parNodo, bool>, predecesores es vector<nodo>, list<nodo> es camino. Las estructuras utilizadas son las provistas por la STL y los tipos de datos involucrados son los estándares de C++.
2. Los componentes del pseudo-código que estén descriptos en palabras, son operaciones que tienen complejidad temporal O(1), excepto que se explicíte lo contrario en algún/os caso/s particular/es.
3. No se tendrán en cuenta los costos de entrada de datos (input) y salida (output). Cabe aclarar que, si bien el algoritmo de BFS se encuentra únicamente para dar formato pedido a la salida, este sí va a ser sometido a un análisis intrínseco debido a que lo realizado por el mismo no es trivial.
4. Ciertas subrutinas no van a ser plasmadas en pseudo-código ya que consideramos que son simples de entender (iterar una matriz, iterar una lista, etc) y puede haber un remplazo de su correspondiente pseudo-código con un párrafo o algunas frases que expliquen lo que harían de manera que esté claro lo que llevan a cabo. De la misma forma se va a mencionar su costo temporal.

Pseudo-códigos y cotas de complejidad

Comenzando por el algoritmo de prim, el siguiente es su pseudo-código:

```
1  tuplaSolucion prim(grafo& input, int& costoTotal)
2
3  int n = input.size()
4
5  grafo res(n,n)
6  vectorPesoPredecesor diccionario(n)
7
8  //Indica, para cada nodo, si fue agregado al AGM
9  vector<bool> unidos(n)
10 unidos[0] = true
11
12 //Variables principales
13 int cn = 0
14 nodo nodoActual = 0
15 bool sigo = true
16
17 //Variables Auxiliares
18 pesoPredecesor conexionMinFueraAGM;
19 nodo nodoAConectar;
20
21 mientras (cn < n && sigo)
22   para todo nodo i desde 0 hasta n-1
23     si (i no esta conectado) && input[nodoActual,j]} >= 0 &&
24         input[nodoActual,i] < diccionario[i].peso)
25
26       diccionario[i] = <input[i,j],nodoActual>
27     fin si
28   fin para
29
30   conexionMinFueraAGM = diccionario[0]
31   nodoAConectar = 0
32
33   //Recorro las aristas almacenadas en diccionario y tomo la menos pesada
        que conecte un nodo que ya este en el AGM con otro que no
34
35
36
37   para todo nodo i desde 0 hasta n-1
38     si (i no esta conectado) && (diccionario[i].peso >= 0)
```

```
39        si diccionario[i].peso < conexionMinFueraAGM.peso
40          conexionMinFueraAGM = diccionario[i]
41          nodoAConectar = i
42        fin si
43      fin si
44    fin para
45    Si (hay encontre alguna arista para unir)
46      unir(res,nodoAConectar, conexionMinFueraAGM.nodo,conexionMinFueraAGM.peso)
47      unidos[nodoAConectar] = true
48      nodoActual = nodoAConectar
49      costoTotal += conexionMinFueraAGM.peso
50      cn++
51    Si no
52      sigo = false
53      cn++
54    fin si
55  fin mientras
56
57  return <res, (sigo) || (cn == n))>
58  fin
```

- Crear el grafo res toma $\Theta(n^2)$. Las declaraciones de 'diccionario' y 'unidos' cuesta $\Theta(n)$.
- El while (lineas 22 - 58) itera n veces, por ende su costo esta caracterizado por $n * O(cuerpoWhile)$. No es Θ ya que si no hay un AGM subyacente al grafo 'input', la rutina no repite el cuerpoWhile tantas veces como nodos haya.
- El for (lineas 24 - 30) realiza n iteraciones de complejidad $O(1)$, por ende su costo total es $O(n)$.
- El for (lineas 38 - 45) realiza n iteraciones con costo temporal $O(1)$. Su costo total es $O(n)$.
- Como el resto de las operaciones tienen costo temporal constante, se ve que el costo del cuerpoWhile $= O(2n) = O(n)$. Entonces, el while mencionado anteriormente es $O(n^2)$, por propiedades de la función O.
- Se devuelve el grafo 'res' por copia y un booleano, dejando un costo de $\Theta(n^2)$ debido al tamaño de la matriz.

Finalmente, si $f(n)$ es la función que describe la complejidad del algoritmo de prim propuesto, sabemos que $f(n) \in 2\Theta(n^2) + \Theta(n) + O(n^2)$. Utilizando propiedades de las funciones de orden y que si $g(n) \in \Theta(h(n)) \Rightarrow g(n) \in O(h(n))$, obtenemos que $f(n) \in O(n^2)$.

Una vez obtenido el AGM, debemos encontrar la arista menos costosa que pertenezca a la red que no se encuentre en el AGM:

```
1  tuplaCiclo encontrarConexionMinParaAnillo(grafo& input, grafo& AGMInput)
2
3  int n = AGMInput.size()
4  parNodo conexionCostoMin = <0,0>
5  bool existe = false;
6
7  para todo nodo i desde 1 hasta n-1
8    para todo nodo j desde 0 hasta i
9      si (hay arista entre i,j en input pero no en AGM)
10       si (input[i][j] < input[conexionCostoMin.nodo1][conexionCostoMin.nodo2])
11       conexionCostoMin = <i,j>
12       existe = true
13       fin si
14     fin si
15   fin para
16 fin para
17 return <conexionCostoMin, existe>
18 fin
```

Si llamamos $g(n)$ a la función que indica el orden de complejidad de dicho algoritmo, podemos decir que: $g(n) \in \Theta(\sum_{i=1}^{n/2} i) = \Theta(\frac{(n/2) * ((n/2) - 1))}{2}) = \Theta((n^2/2) - n) = \Theta(n^2)$.

En particular, por propiedad de la función Θ, $g(n) \in O(n^2)$.

Resta ver el pseudo-código del BFS Modificado y dar su función de complejidad:

```
1  predecesores BFSMod(grafo& AGM, nodo& inicio, nodo& destino)
2
3  int n = AGM.size()
4  predecesores predecesores(n)
5  vector<bool> visitados(n)
6  cola<nodo> noVisitados
7
8  visitados[inicio] = true
9  noVisitados.push(inicio)
10
11 nodoActual = inicio
12
13 mientras (noVisitados no este vacia) && (nodoActual != destino)
14   noVisitados.pop()
15
16   para todo nodo i desde 0 hasta n-1
17     si (i no fue visitado) && (hay arista entre nodoActual e i en AGM)
18       predecesores[i] = nodoActual
19       visitados[i] = true
20       cola.push(i)
21     fin si
22   fin para
23
24   nodoActual = cola.front()
25 fin mientras
26
27 return predecesores
28 fin
```

- Las declaraciones de los vectores (lineas 4-5) toman $O(n)$, por otra parte la creación de la cola vacia se realiza en tiempo $O(1)$.
- El while (lineas 13-25) itera tantas veces como nodos visite entre el inicio y el destino, su peor caso es que atraviese todos los nodos antes de lograrlo, lo cuál nos deja con una complejidad de $n * O(cuerpoWhile)$.
- El for (lineas 16-22) realiza n pasos donde se ejecutan operaciones con costo $O(1)$, la complejidad resultante de él resulta $O(n)$.
- Dado los 2 ítems anteriores, concluimos que la complejidad del while es $O(n^2)$.
- El vector predecesores se devuelve por copia, temporalmente eso requiere $O(n)$ tiempo.

Llamaremos $h(n)$ a la función que caracteriza la complejidad de este algoritmo. Por todo lo mencionado, $h(n) \in O(n^2) + 3O(n)$, lo que implica, por propiedades de la función de orden O, que $h(n) \in O(n^2)$.

Conclusión

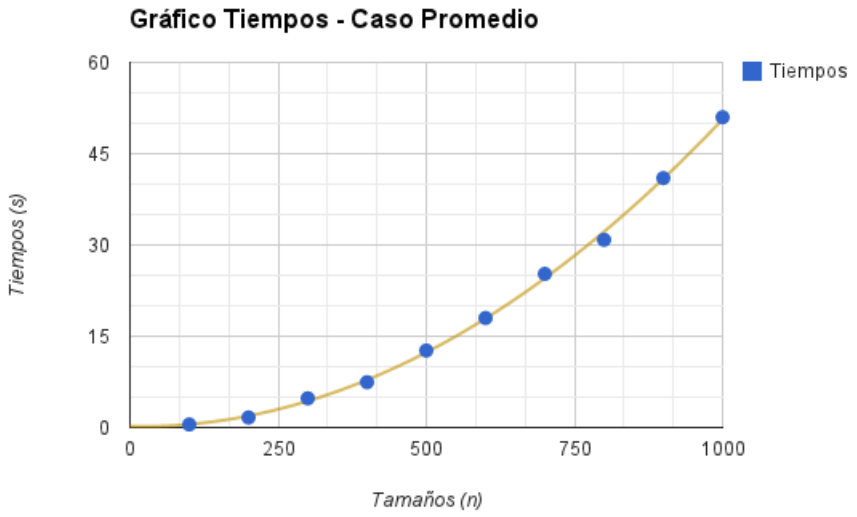
La resolución del problema, sin tener en cuenta el input y el output, utiliza 1 llamado de cada uno de los algoritmos mencionados. Denominaremos $c(n)$ al costo de resolver el problema $\Rightarrow c(n) \in f(n) + g(n) + h(n) \Rightarrow c(n) \in O(n^2) + O(n^2) + O(n^2) = 3O(n^2) = O(n^2)$ Finalmente, la complejidad de la solución planteada es estrictamente mejor que $O(n^3)$, con lo cuál se cumple lo pedido por la empresa.

3.5. Experimentación

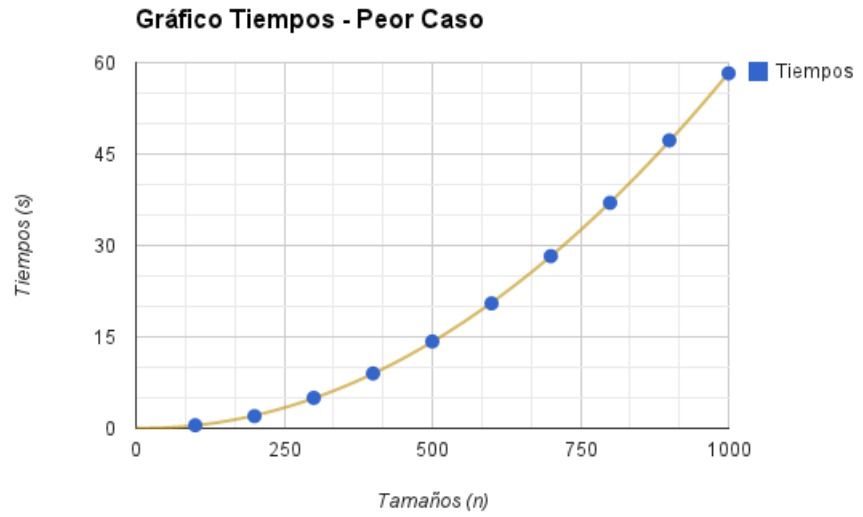
Introducción

Para la experimentación, se midieron 2 casos: promedio, peor y aumentando la cantidad de equipos sin aumentar la cantidad de aristas. Estos 3 escenarios lograron mostrar que se cumplió con la cota de complejidad temporal pedida y nos dieron ciertas conclusiones interesantes. La técnica utilizada para tomar valores fue la siguiente: construimos generadores para cada caso particular, ellos nos generaron 100 instancias distintas de cada uno de sus casos, luego se llevaban acabo 100 repeticiones de resolución sobre cada instancia. Sobre cada instancia particular, tomabamos su mejor tiempo, debido a que es el valor que mejor representa el tiempo actual que tomó resolver el ejercicio. Es sabido que el cpu atiende muchas tareas en simuláneo y por ende es complicado medir efectivamente el tiempo real de la rutina ya que deberíamos hacer que el procesador sólo ejecute la rutina frenando toda otra actividad del equipo. Habiendo hecho esto, tomamos el promedio de las mediciones de cada instancia, lo que nos da una buena aproximación. Las mediciones están hechas en segundos y el tamaño está medido en n, la cantidad de equipos. Para cada caso particular se detallará cuál es la relación entre el n y la cantidad de conexiones para poder hacer un análisis más minucioso.

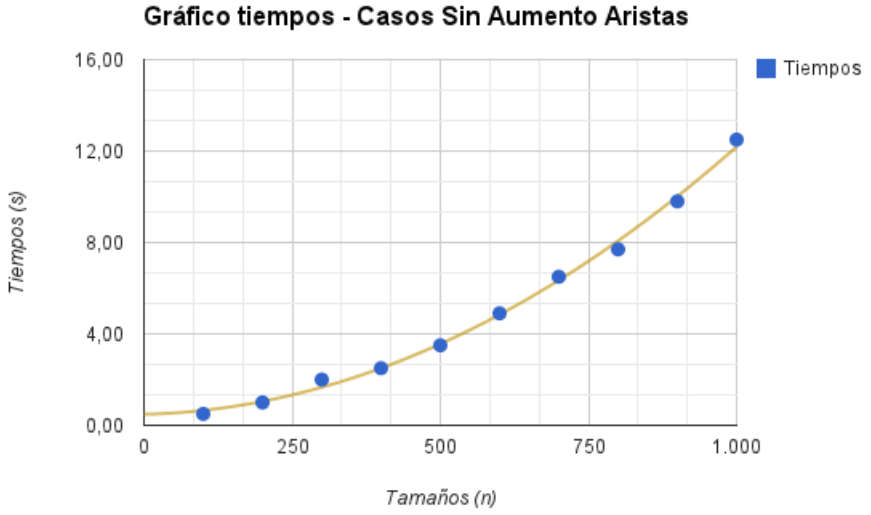
Gráficos y conclusiones



Un caso sin sorpresas. Ayuda a ver cierta curvatura polinómica que evidencia la complejidad temporal. Se puede notar, para ciertos tamaños de instancias, que el incremento no es regular. Esto sucede porque pueden haber casos que se descartan tempranamente ya que esos grafos no poseen un AGM subyacente. Para estos casos, la cantidad de aristas, m, esta acotada de la siguiente manera $0 \leq m \leq \frac{n*(n-1)}{2}$. En estas situaciones puede suceder que el factor que mas aporta a los tiempos de ejecución varíe entre la cantidad de equipos y la cantidad de aristas. Más adelante, basandonos en otras circumstancias, veremos que es lo que más determinante.



Se puede observar que la cota de complejidad es efectivamente polinomial. La cantidad de conexiones, dadas estas circumstancias, están fijas y equivalen a $\frac{n*(n-1)}{2}$, formando un grafo completo. La diferencia entre casos peores y promedios es notable, llegando a tener diferencias de más de 6 segundos para ciertas instancias. En estos casos, el predominio está claramente dado por el número de conexiones en materia de tiempos. El crecimiento es mucho más regular debido a que no se puede dar un caso de rápida terminación, un grafo completo siempre contiene un AGM.



Lo que se hizo en este caso es tomar un grafo de 100 nodos que sea completo, medir los tiempos, e ir aumentando en 100 la cantidad de nodos sin generar nuevos ejes, de forma tal que los grafos son inexorablemente inválidos para presentar una solución ya que cada nodo nuevo aporta una componente conexa adicional. ¿Qué es lo que genera el aumento de tiempos? El tamaño de la matriz de adyacencia pesada. Que no haya conexiones adicionales, no implica que pueda optar por no recorrer toda la matriz en pos de armar el AGM. Si bien esta caso termina siempre en el nodo 101, para todos los nodos anteriores debe iterar hasta n en busca de nuevas conexiones. Esto nos da la pauta de que, para esta problemática en particular, la elección de una estructura alternativa que favorezca no iterar de más, como lista de adyacencias pesadas, hubiese sido una mejor decisión. No por la cota de complejidad, sino por los tiempos en la práctica.

Comentarios finales

1. Nos hubiese gustado, en los 2 primeros gráficos, comparar contra funciones cuadráticas. No llegamos a hacerlo por cuestiones de tiempo y limitaciones del software elegido.
2. La idea de cambiar de estructura a listas de adyacencia se nos ocurrió pasado el tiempo de haber armado la rutina entera, los pseudo-códigos y gran parte del informe, incluyendo la complejidad. Una buena lección para recordarnos que el trade-off del acceso a una posición en $O(1)$ puede costar significativamente.