



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

## Trabajo Práctico 2

9 de mayo de 2014

Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Chapresto, Matías	201/12	matiaschapresto@gmail.com
Dato, Nicolás	676/12	nico_dato@hotmail.com
Fattori, Ezequiel	280/11	ezequieltori@hotmail.com
Vileriño, Silvio	106/12	svilerino@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

<b>1. Ejercicio 1</b>	<b>3</b>
1.1. Descripción del problema . . . . .	3
1.2. Ideas para la resolución . . . . .	3
1.2.1. Algoritmo . . . . .	3
1.3. Justificación del procedimiento . . . . .	3
1.4. Cota de complejidad . . . . .	3
1.5. Casos de prueba y resultado del programa . . . . .	3
1.6. Mediciones de performance . . . . .	3
1.7. Conclusiones . . . . .	3
<b>2. Ejercicio 2</b>	<b>4</b>
2.1. Descripción del problema . . . . .	4
2.1.1. Ejemplo . . . . .	4
2.2. Ideas para la resolución . . . . .	5
2.2.1. Algoritmo . . . . .	5
2.2.2. Ejemplo de ejecución . . . . .	6
2.3. Justificación del procedimiento . . . . .	6
2.3.1. Demostración de Kruskal sobre Prim . . . . .	7
2.3.2. Demostración de que aplicar Kruskal nos genera la solución . . . . .	8
2.4. Cota de complejidad . . . . .	8
2.5. Casos de prueba y resultado del programa . . . . .	9
2.6. Mediciones de performance . . . . .	9
2.7. Conclusiones . . . . .	9
<b>3. Ejercicio 3</b>	<b>10</b>
3.1. Descripción del problema . . . . .	10
3.2. Ideas para la resolución . . . . .	10
3.3. Estructuras de datos . . . . .	12
3.3.1. Algoritmo . . . . .	13
3.4. Cota de complejidad . . . . .	13
3.5. Casos de prueba y resultado del programa . . . . .	13
3.6. Mediciones de performance . . . . .	13
3.7. Conclusiones . . . . .	13
<b>4. Apéndice: Generación de casos de prueba</b>	<b>14</b>
<b>5. Apéndice: Código fuente relevante</b>	<b>15</b>
5.1. Ejercicio 1 . . . . .	15
5.2. Ejercicio 2 . . . . .	15
5.3. Ejercicio 3 . . . . .	18
<b>6. Apéndice: Entregable e instrucciones de compilacion y testing</b>	<b>19</b>

## Resumen

## 1. Ejercicio 1

1.1. Descripción del problema

1.2. Ideas para la resolución

1.2.1. Algoritmo

1.3. Justificación del procedimiento

1.4. Cota de complejidad

1.5. Casos de prueba y resultado del programa

1.6. Mediciones de performance

1.7. Conclusiones

## 2. Ejercicio 2

### 2.1. Descripción del problema

El problema se trata de un conjunto de ciudades hubicadas a cierta distancia entre ellas, las cuales todas deben de ser provistas de gas. Para ésto, tenemos una cantidad  $k$  de centrales distribuidoras de gas, y tuberías para conectar ciudades. Una ciudad tiene gas si hay un camino de tuberías que llegue hasta una central distribuidora, es decir, si una ciudad está conectada a otra ciudad por una tubería y a su vez ésta está conectada a otra ciudad por medio de una tubería la cual tiene una central distribuidora, entonces las 3 ciudades tienen gas. Se pide lograr que todas las ciudades tengan gas, pero que la longitud de la tubería más larga de la solución debe ser la más corta posible. La longitud de una tubería es igual a la distancia entre las 2 ciudades.

El algoritmo tiene que tener una complejidad de  $O(n^2)$ , con  $n$  la cantidad de ciudades.

De a partir de ahora,  $k$  va a ser siempre la cantidad de centrales distribuidoras disponibles, y  $n$  la cantidad de ciudades.

#### 2.1.1. Ejemplo

Como entrada podemos tener 6 ciudades y 2 centrales distribuidoras, como en la Figura 1, y la solución que se busca es como indica la Figura 2.

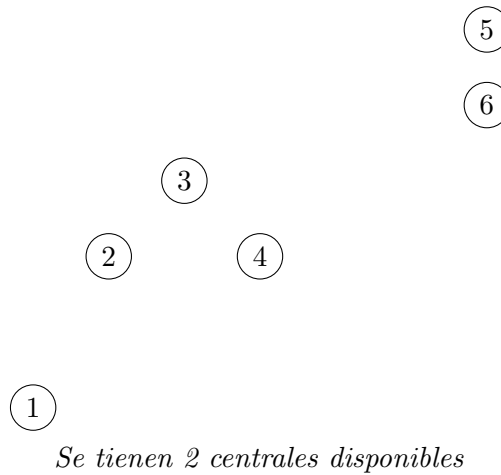


Figura 1: La entrada del problema, con las 5 ciudades y la cantidad de centrales

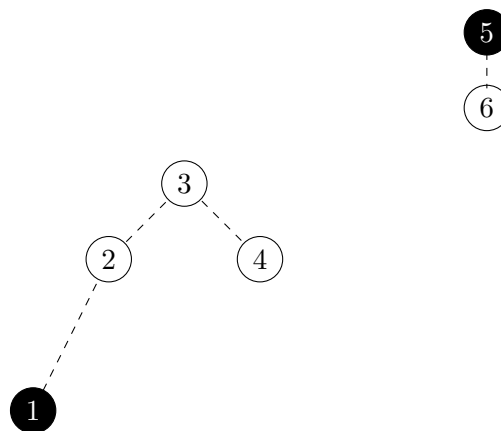


Figura 2: La solución al problema de la Figura 1, los nodos negros son los que contienen las centrales de distribución

## 2.2. Ideas para la resolución

Para la resolución del problema, se puede pensar en un *grafo*, donde las ciudades son nodos y las tuberías las aristas, cada arista va a tener una distancia asociada que es la distancia entre los nodos (ciudades) que conecta.

Como cada ciudad tiene gas si hay un camino hasta una central distribuidora, entonces todas las ciudades que se conectan a una misma central pertenecen a una misma componente conexa. Como tenemos un máximo de  $k$  centrales, la solución tiene que tener un máximo de  $k$  componentes conexas y las centrales se colocarán cada una en una componente conexa y en cualquier ciudad dentro de la componente conexa, ya que la distancia de las aristas dentro de una componente conexa no se verá modificada si se cambia de nodo la central dentro de la misma componente conexa.

Entonces, la solución que se pide es que el grafo tenga a lo sumo  $k$  componentes conexas, y que la distancia de la arista mas larga, sea la más corta posible.

Una idea que se propone es comenzar con todos los nodos sueltos, sin ninguna arista conectada, y calcular todas las distancias entre todos los nodos, es decir, calcular todas las tuberías posibles con sus respectivas distancias. Luego se ordenarán las aristas (tuberías) de menor a mayor de acuerdo a sus longitudes. Se calcula si la cantidad de componentes conexas es menor o igual a  $k$ , si es así, entonces el algoritmo termina, sino coloca la tubería más corta que no genere un ciclo y continúa preguntando sobre la cantidad de componentes conexas e iterando sobre las aristas siguiendo el orden de menor a mayor. Básicamente es aplicar *Kruskal* sobre un grafo completo.

Como se requiere tener todas las aristas ordenadas por peso, en un grafo completo tenemos  $n(n-1)/2$  aristas, dandonos una complejidad de al menos  $O(n^2 \log n^2)$ , y no cumple con el requerimiento. Para mejorar la complejidad del algoritmo, en vez de calcular la distancia de todas las aristas e iterar sobre todas las aristas, requiriendo ordenar por peso *todas* las aristas, primero creamos un árbol generador mínimo (con las aristas más cortas posibles), con una idea como el algoritmo de *Prim* y que tenga de cota  $O(n^2)$ , el árbol generador mínimo generado nos queda  $n - 1$  aristas, y luego, al igual que antes, se ordenan y se recorren esas  $n - 1$  aristas de menor a mayor.

En resumen, la idea es partir de un grafo completo con  $n$  nodos, aplicar *Prim* y obtener el *AGM*, luego sobre ese árbol aplicar *Kruskal* pero cortando el algoritmo cuando se tienen tantas componentes conexas como  $k$  centrales de gas. La colocación de las centrales de gas se colocan luego en un nodo cualquiera de los nodos de cada componente conexa.

En la sección 2.2.1 se propone un pseudocódigo, en la sección 2.2.2 se verá un ejemplo de ejecución del algoritmo, en la sección 2.3 se justificará la correctitud, y en la sección 2.4 se hará el cálculo de la complejidad del algoritmo.

### 2.2.1. Algoritmo

---

**Algorithm 1** minimizarTuberias

---

**Require:** *centrales*: cantidad de centrales disponibles, mayor que 0

**Require:** *ciudades*: las ciudades con sus posiciones

**Require:**  $n$ : cantidad de ciudades en el parámetro *ciudades*

**Ensure:** Retorna el grafo tal que hay tanas componentes conexas como *centrales*, y que la arista más larga es la más corta posible

```
1: procedure MINIMIZAR_TUBERIAS(Entero: centrales, Array: ciudades, Entero: n) → Grafo
2:   Grafo g ← NUEVOGRAFO(n)                                ▷ Creo un grafo con n nodos, sin aristas
3:   <bool agregado, entero distancia, entero nodo> nodos[n]  ▷ La distancia es desde n (índice del
   array) hasta nodos[n].nodo
4:   <nodo1, nodo2, distancia> aristas[n - 1]

5:   nodos[0] ← <true, 0, 0>
6:   for i ← 1; i < n; i++ do
```

```

7:     nodos[i] ← <false, DISTANCIA(ciudades[i], ciudades[0]), 0>
8: end for

9: for agregados ← 0; agregados < n - 1; agregados++ do
10:     distancia_minima ← ∞
11:     nodo_minimo ← 0
12:     for i ← 0; i < n; i++ do           ▷ Busco el nodo mas cercano al árbol que ya tenemos
13:         if nodos[i].agregado = false then
14:             if nodos[i].distancia < distancia_minima then
15:                 distancia_minima ← nodos[i].distancia
16:                 nodo_minimo ← i
17:             end if
18:         end if
19:     end for

20:     nodos[nodo_minimo].agregado ← true
21:     aristas[agregados] ← <nodo_minimo, nodo[nodo_minimo].nodo, distancia_minima>   ▷
Agrego el nodo encontrado

22:     for i ← 0; i < n; i++ do           ▷ Actualizo la distancia de los nodos no agregados aún
23:         if nodos[i].agregado = false then
24:             if nodos[i].distancia > DISTANCIA(ciudades[i], ciudades[nodo_minimo]) then
25:                 nodo[i].distancia ← DISTANCIA(ciudades[i], ciudades[nodo_minimo])
26:                 nodo[i].nodo ← nodo_minimo
27:             end if
28:         end if
29:     end for
30: end for

31: ORDENAR(aristas)                       ▷ Ordeno las aristas por la distancia

32: for componentes ← n; componentes > centrales; componentes-- do
33:     AGREGARARISTA(g, aristas[n - componentes].nodo1, aristas[n - componentes].nodo2)
34: end for
35: retornar ← g
36: end procedure

```

### 2.2.2. Ejemplo de ejecución

## 2.3. Justificación del procedimiento

Para la justificación vamos a justificar primero que si se aplica el algoritmo de *Kruskal* sobre el *AGM* generado por el algoritmo de *Prim* dado un grafo completo, entonces llegamos a un grafo con pesos de aristas iguales que aplicar *Kruskal* sobre el grafo completo directamente, aunque puede tener aristas diferentes de igual peso. Como tienen los mismos pesos aunque las aristas sean diferentes, a efectos de minimizar el peso de la arista de mayor peso da el mismo resultado. (Sección 2.3.1).

Luego vamos a justificar que aplicando el algoritmo de *Kruskal* sobre un grafo completo pero cortando cuando tenemos  $K$  componentes conexas, nos genera la solución que buscamos (Sección 2.3.2).

Teniendo demostrado esos 2 puntos, entonces aplicar *Kruskal* a un grafo completo nos da la solución que buscamos y aplicar *Kruskal* al resultado *Prim* (que es lo que realiza el algoritmo propuesto) nos da el mismo resultado que aplicar *Kruskal* directamente, siendo la solución buscada.

### 2.3.1. Demostración de Kruskal sobre Prim

Demostraremos que aplicar el algoritmo de *Kruskal* sobre un grafo completo da aristas con los mismos pesos que aplicar *Kruskal* sobre el *AGM* resultante del algoritmo de *Prim*. Como el problema pone restricción sobre el peso de las aristas (de las tuberías), entonces nos fijaremos que las aristas del resultado pueden ser diferentes pero si se mira sólo los pesos de las aristas, estos son los mismos.

*Demostración.* Un *AGM* de un grafo tiene como suma de los pesos de sus aristas, la menor suma posible de todos los árboles de un grafo. Si se aplica *Prim* y *Kruskal* a un mismo grafo (un grafo completo por ejemplo), ambos dan un *AGM* como resultado, por ende el peso del árbol generado por *Prim* del grafo  $g$  ( $P_p(g)$ ) es igual al peso del que genera *Kruskal* ( $P_k(g)$ ), si no fueran iguales, el que tiene mayor peso no sería un *AGM*. Como ámbos son árboles del mismo grafo, ambos tienen  $n - 1$  aristas ( $n =$  cantidad de nodos del grafo).

$$\begin{aligned} P_p(g) &= \sum_{i=0}^{i < n-1} \pi(a_i) \\ &= \\ P_k(g) &= \sum_{i=0}^{i < n-1} \pi(a_i) \end{aligned}$$

Como *Prim* y *Kruskal* son 2 algoritmos diferentes, podrían generar un árbol con diferentes aristas pero igual mantienen la igualdad en la suma de los pesos y la cantidad de aristas.

Supongamos que un algoritmo pone las mismas aristas que el otro pero en vez de poner la arista  $a$  pone la arista  $a'$  con distinto peso, entonces también tiene que haber cambiado otra arista  $b$  por  $b'$  para mantener la igualdad de la suma. Supongamos que tenemos todas las aristas  $a_i$  del *AGM* y las ordenamos de menor a mayor y de la arista  $a_0$  a  $a_{j-1}$  y de  $a_j + 2$  a  $a_{n-2}$  las aristas son las mismas para ambos algoritmos pero  $a_j$  y  $a_{j+2}$  son diferentes, entonces:

$$\begin{aligned} P_p(g) &= \sum_{i=0}^{i < j} \pi(a_i) + \pi(a') + \pi(b') + \sum_{i=j+2}^{i < n-1} \pi(a_i) \\ &= \\ P_k(g) &= \sum_{i=0}^{i < j} \pi(a_i) + \pi(a) + \pi(b) + \sum_{i=j+2}^{i < n-1} \pi(a_i) \end{aligned}$$

Simplificamos la igualdad de las suma, nos queda:

$$\begin{aligned} \pi(a') + \pi(b') \\ &= \\ \pi(a) + \pi(b) \end{aligned}$$

Como dijimos que el peso de  $a'$  es distinto de  $a$ , podemos suponer que  $\pi(a) > \pi(a')$  y por lo tanto  $\pi(b) < \pi(b')$  (se puede suponer también lo contrario y sería análogo). *Prim* entonces hubiera agregado  $a'$  con menor peso que  $a$ , pero como *Kruskal* recorre las aristas de menor a mayor peso, ya tendría que haber pasado por  $a'$ , y como las aristas anteriores a  $a'$  son las mismas en ambos algoritmos, agregar  $a'$  no genera ciclos, entonces *Kruskal* hubiera agregado a  $a'$ , entonces  $\pi(a') = \pi(a)$  porque  $a' = a$ . Aún quedaría  $\pi(b') \neq \pi(b)$ , pero como todas las demás aristas son las mismas (incluidas  $a'$  y  $a$ ), entonces para mantener la igualdad de la suma  $\pi(b') = \pi(b)$ , contradiciendo la suposición que un algoritmo cambiaría una arista por otra con peso distinto.

Si cambia una arista por otra de igual peso, no modifica la solución porque el peso de la arista de mayor peso seguiría siendo el mismo.



Como aplicar *Kruskal* a un árbol da el mismo árbol (porque es el único subgrafo que es un árbol), aplicar *Kruskal* al resultado de *Prim* da el mismo resultado que aplicar directamente *Prim*, y ya demostramos que aplicar *Prim* y *Kruskal* dan árboles con mismos pesos de aristas, por lo que aplicar *Kruskal* directamente o aplicar *Kruskal* a la salida de *Prim*, da árboles con los mismos pesos de aristas.  $\square$

### 2.3.2. Demostración de que aplicar Kruskal nos genera la solución

Demostraremos que teniendo un grafo completo, si aplicamos *Kruskal* y dejamos de agregar aristas cuando llegamos a  $k$  componentes conexas, tenemos la solución que buscamos:

*Demostración.* El algoritmo de *Kruskal* ordena las aristas de menor a mayor según su peso, y agrega 1 a 1 las aristas según dicho orden tal que no genere un ciclo. Al agregar una arista se disminuye en 1. Nosotros detendremos el algoritmo cuando se llega a  $k$  componentes conexas. Supongamos que tenemos  $m$  aristas ordenadas de menor a mayor por su peso,  $(a_0, a_1 \dots a_{m-1})$ , que se fueron agregando (aunque algunas generaban ciclos y no se agregaron), y que detuvimos el algoritmo de *Kruskal* al agregar la arista  $a_j$  porque se llegaron a  $k$  componentes conexas. Entonces, como las aristas estaban ordenadas:

$$\begin{aligned} (\forall 0 \leq i < j) a_i &\leq a_j \\ &\wedge \\ (\forall j < i < m) a_i &\geq a_j \end{aligned}$$

Todos los  $a_i, i < j$  están agregados al resultado de *Kruskal*.

El  $a_j$  es la arista de mayor peso (la tubería más larga), y es la que queremos minimiza.

Si existiese una mejor forma de armar las componentes conexas, entonces existe un  $a_i, a_i < a_j \implies i < j$ , tal que se puede reemplazar  $a_i$  por  $a_j$  y así tener la arista de mayor peso que es de menor peso que la que obtuvimos. Como  $i < j$ , *Kruskal* ya la analizó  $a_i$  antes de pasar por  $a_j$ .

Si  $a_i$  no generaba ciclos, entonces esa arista pertenece al resultado del algoritmo, y como no se detuvo en  $a_i$  significa que aún agregando esa arista (y todas las anteriores) no se llegaban a las  $k$  componentes conexas que se buscan, entonces no se puede sacar  $a_j$  porque no se llegaría a la cantidad de componentes conexas necesarias, contradiciendo que se puede quitar  $a_j$  para mejorar la solución.

Si  $a_i$  generaba un ciclo, entonces esa arista no pertenece al resultado del algoritmo. Notar que como  $a_i$  se analiza antes que  $a_j$  por tener menor peso, entonces  $a_i$  generaba un ciclo cuando  $a_j$  aún no estaba agregada. Si se quita  $a_j$  y se agrega  $a_i$ , al quitar  $a_j$  se aumenta en 1 la cantidad de componentes conexas (ahora tenemos  $k + 1$ ) y no es solución, y al agregar  $a_i$  se está generando un ciclo, por lo que no disminuye la cantidad de componentes conexas, manteniéndose en  $k + 1$  y no es solución tampoco, por lo que contradice que se puede quitar  $a_j$  para agregar  $a_i$  y tener una mejor solución.

Entonces, agregar aristas en orden según su peso si no genera ciclos y parar cuando se alcanzan las  $k$  componentes conexas, nos da como peso de la arista más larga, el peso mas chico posible, siendo la solución que se busca del problema.  $\square$

## 2.4. Cota de complejidad

Se analiza la complejidad de algoritmo 1, para ésto se supone que el cálculo de la distancia entre ciudades (el peso de las aristas/tuberías) se realiza en  $O(1)$

La cantidad de nodos se representará como  $n$ , y la cantidad de centrales como  $k$ .

- La inicialización de unas variables se realiza de la línea 5 a 8, y se realizan  $n$  iteraciones calculando la distancia y guardando en un vector, quedando  $O(n)$
- Para la creación del AGM entre las líneas 9 a 30 realiza  $n - 1$  iteraciones
  - Líneas 12 a 19, busca el nodo más cercano, realizando  $n$  iteraciones de comparaciones y asignaciones que son  $O(1)$ , quedandonos  $O(n)$

- Las líneas 20 y 21 son  $O(1)$
- La actualización de las distancias, entre las líneas 22 y 29 realiza  $n$  iteraciones de comparaciones y asignaciones que son  $O(1)$ , quedandonos  $O(n)$
- Ordenar las aristas en la línea 31, y se realizan sobre las aristas agregadas en la iteración anterior, la cual agrega  $n - 1$  aristas, entonces se ordenan  $n - 1$  elementos. Al  $n$  no estar acotado, se puede realizar con algoritmos como *MergeSort* o *HeapSort* en  $O(n \log n)$
- Entre las líneas 32 y 34 se realizan  $n - k$  iteraciones, implementando el grafo  $g$  en una *matriz de adyacencia*, *AgregarArista* se realiza en  $O(1)$ , quedandonos una complejidad de  $O(n - k)$ . Notar que si  $k > n$ , no se realiza ninguna iteración.

Bajo éste análisis, la complejidad nos queda (1)

$$\begin{aligned}
& O(n) + (n - 1) * (O(n) + O(1) + O(n)) + O(n \log n) + O(n - k) \\
& = O(n) + O(n^2) + O(n) + O(n^2) + O(n \log n) + O(n - k) \\
& = O(2n) + O(2n^2) + O(n \log n) + O(n - k) \\
& = O(n^2)
\end{aligned} \tag{1}$$

Cumpliendo así la complejidad requerida.

## 2.5. Casos de prueba y resultado del programa

## 2.6. Mediciones de performance

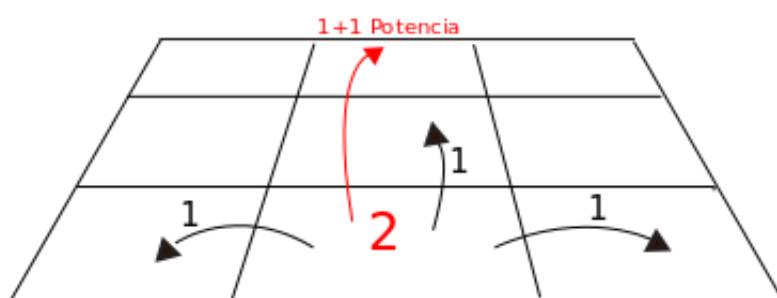
## 2.7. Conclusiones

### 3. Ejercicio 3

#### 3.1. Descripción del problema

El problema se trata de un juego que forma parte de un reality show. Este juego tiene un campo de juego cuadrado dividido en celdas, de tamaño  $n \times n$ . Cada una de estas celdas posee un resorte con el que se puede saltar hacia otras celdas. Estos resortes varían en sus potencias, siendo la potencia la cantidad de celdas que pueden propulsar al jugador. Es posible apuntar los resortes hacia adelante, atrás, izquierda y derecha. Adicionalmente cada jugador posee unidades extra de potencia, que le permiten potenciar sus saltos. Éstas son limitadas, y el jugador las podrá usar a elección en los saltos que le parezcan convenientes.

Un ejemplo de salto sería la siguiente situación: vemos que la celda tiene una potencia de 1, por lo que el jugador podría saltar una celda izquierda, hacia adelante o hacia la derecha. Y en caso de que quisiera usar una potencia, podría saltar dos celdas hacia adelante, pero su reserva de potencias se reduciría en uno.

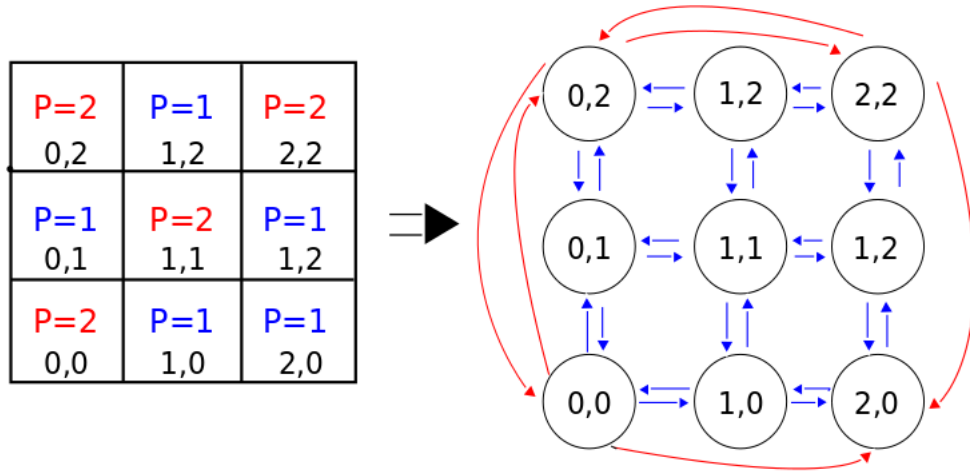


El objetivo del juego es llegar de una celda origen a otra destino, realizando la menor cantidad de saltos posibles. Por esto, se pide el diseño e implementación de un algoritmo que calcule y de como resultado uno de los caminos que lleguen de origen a destino cuya cantidad de saltos sea mínima.

El algoritmo tiene que tener una complejidad de  $O(n^3 * k)$ , con  $n$  la cantidad de filas y columnas del campo de juego y  $k$  la cantidad de potencias otorgadas al inicio del juego al jugador.

#### 3.2. Ideas para la resolución

Decidimos encarar la resolución como un problema de grafos. El campo de juego es representado como un grafo con un nodo por cada celda, y las aristas son orientadas, de peso constante y representan todos los posibles saltos entre celda y celda.



Por cada uno de los valores  $0..k$  de potencia tenemos un campo de juego de los anteriormente mencionados. Los vamos a llamar "*niveles*", y van a ser subgrafos interconectados para formar un grafo general que va a resolver el problema. Entre estos niveles va a haber aristas conectando los nodos de mayor nivel con los de menor nivel, representando el hecho de "*gastar*" potencias, es decir, si en un turno estoy en el nivel  $k$  y utilizo dos potencias, en el siguiente turno voy a estar posicionado en un nodo del nivel  $k - 2$ .

De esta forma, estar en el nodo  $(i, j)$  del nivel  $k$  representa estar posicionado en la celda de juego  $(i, j)$  y disponer de  $k$  potencias para utilizar. Podemos "*saltar entre celdas*", moviéndonos de un nodo a otro usando las aristas este nivel. En caso de querer usar las potencias, debemos movernos por aristas que conectan los nodos de nivel  $k$  con los niveles inferiores. Por ejemplo, si estamos en el nodo  $(i, j, k)$ , de potencia 2, y queremos saltar 3 nodos hacia la derecha, debemos tomar la arista orientada que nos lleva al nodo  $(i + 3, j, k - 1)$  (en caso de disponer de una potencia). De esto deducimos que las aristas entre niveles sólo salen de nodos de nivel superior hacia nodos de nivel inferior. Formalicemos esto:

Para cada nivel, sea:

$$G_L = (V_L, E_L) / V_L = V_{1L}, \dots, V_{nL} = \{\text{casillas } 1..n \text{ con } L \text{ potencias disponibles}\}$$

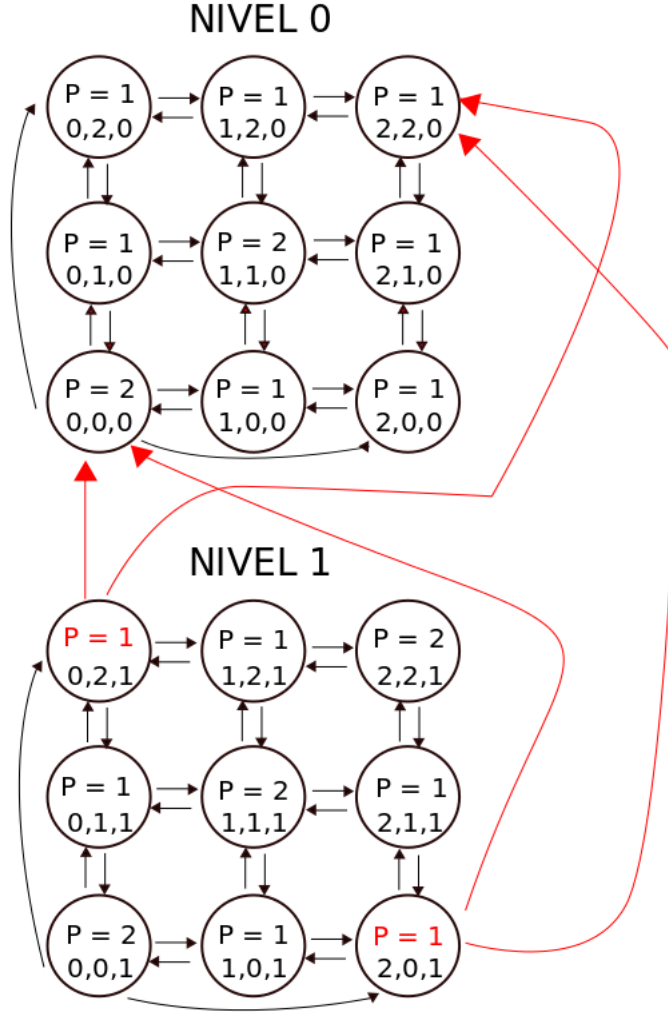
Por lo tanto, el grafo  $G$  es:

$$G = (W, A) / W = \cup_{i=1}^k V_i(G_i) \wedge A = (\cup_{L=1}^k E_L(G_L)) \cup T$$

Donde  $T$  son las aristas entre niveles, descritas a continuación:

Se tienen aristas dirigidas en cada nivel tal que si  $v, u \in V_i$ ,  $(u, v) \in E_i$  sii  $v$  es alcanzable desde  $u$  sin usar potencias. Dados niveles  $G_m, G_j / m < j$  y dos vértices  $v \in V_m(G_m) \wedge w \in V_j(G_j)$  existe una arista dirigida "*entre*" niveles, que pertenezca a  $T$ , sii  $w$  es alcanzable desde  $v$  usando estrictamente  $(j - m)$  unidades de potencia, es decir, que la distancia en casillas (verticales u horizontales) de  $v$  a  $w$  es igual a  $(j - m) + \text{pot}(v)$ .

En el siguiente diagrama mostramos las aristas entre niveles que deberían tomarse en caso de usar una potencia en las celdas  $(0, 2)$  y  $(2, 0)$



Notemos que al avanzar hacia niveles más bajos no se puede ir hacia arriba (lo cual tiene sentido pues el nivel actual indica la cantidad de potencias disponibles para usar, y este valor nunca puede aumentar)

Definimos  $n_0$  como el nodo origen  $/n_0 \in V_k$  (primer nivel, más alto), dado por las coordenadas cartesianas de la celda de comienzo de la entrada. Si la celda destino en el juego es  $(i, j)$  definimos el conjunto de nodos destino como los nodos  $(i, j, k)$ , con  $k$  entre 0 y la cantidad de potencias máxima, de forma que al llegar al nodo destino  $(i, j, l)$ ,  $l$  es la cantidad de potencias sin usar.

Las aristas son de peso constante en su totalidad (en particular, 1). Además, en cada arista se guarda un valor indicando la cantidad de niveles atravesadas en este paso, lo cual es equivalente a la cantidad de potencias utilizadas en el salto. No confundir este valor con el costo o peso de la arista. (Dibujo zarpado)

De esta forma, encontrar el camino que realice la menor cantidad de saltos se reduce a buscar el camino más corto en el grafo de  $n_0$  a algún nodo destino, guardando en cada caso el valor de cada arista (potencia usada) y la posición  $(i, j)$  del nodo sin importar el nivel actual. Un recorrido de anchura BFS del grafo nos brinda el camino más corto, comenzando la búsqueda desde  $n_0$ .

### 3.3. Estructuras de datos

A cada una de las celdas, las representamos con la clase *nodo*, que posee tres atributos:  $x$ ,  $y$  y  $level$ , (el numero de fila y columna de la celda que representa este nodo, y el nivel en el que se encuentra el nodo).

Representamos el grafo en memoria con una modificación del método de listas de adyacencia. La

estructura es un diccionario (java *HashMap*)

#### **3.3.1. Algoritmo**

#### **3.4. Cota de complejidad**

#### **3.5. Casos de prueba y resultado del programa**

#### **3.6. Mediciones de performance**

#### **3.7. Conclusiones**

## 4. Apéndice: Generación de casos de prueba

Para el testeo de los algoritmos y la medición de tiempos en función de la entrada, se programó una utilidad para generar los casos de prueba.

El programa recibe como parámetro el ejercicio del cual se quieren generar los casos, y las variables, como el tamaño de la entrada o en el ejercicio 1 la cantidad de días que estaba disponible el inspector.

Los números aleatorios que se generaron en los casos, se hizo con la función *random()* de C ( <http://linux.die.net/man/3/random> ) usando como semilla el tiempo en microsegundos.

Para el ejercicio 2, se le indica al programa la cantidad de ciudades, la cual se fue incrementando para la medición del tiempo, y la cantidad de centrales se indicaba como un número aleatorio entre 1 y la cantidad de ciudades.

Así se crearon muchos casos donde se fue incrementando el tamaño de la entrada y ejecutando varios tests de un mismo tamaño y varias veces el mismo test para obtener un promedio del tiempo que tarda en resolverlo y descartar algunas imperfecciones que puedan surgir por el entorno donde se está midiendo los tiempos, como puede ser que justo se ejecute otra tarea.

## 5. Apéndice: Código fuente relevante

### 5.1. Ejercicio 1

### 5.2. Ejercicio 2

```
1 typedef int Nodo;
2
3 struct GrafoMatrizAdyacencia{
4     int nodos; //cantidad de nodos
5     int *componente_conexa_nodos; //array para asignar componentes conexas
6     int componentes.conexas;
7     int **adyacencia; //la matriz ed adyacencia
8     int aristas; //aristas colocadas
9 };
10
11 typedef struct GrafoMatrizAdyacencia Grafo;
12
13 typedef struct Ciudad {
14     Nodo nodo;
15     int x;
16     int y;
17 } Ciudad;
18
19 typedef struct NodoDistancia {
20     int agregado;
21     int distancia;
22     Nodo nodo;
23 } NodoDistancia;
24
25 typedef struct Arista{
26     Nodo nodo1;
27     Nodo nodo2;
28     int distancia;
29 } Arista;
30
31 Grafo *crear_grafo(int nodos)
32 {
33     int i;
34
35     Grafo *g = NULL;
36     if(nodos <= 0)
37         return NULL;
38
39     g = (Grafo *)malloc(sizeof(Grafo));
40     g->nodos = nodos;
41     g->componente_conexa_nodos = (int *)calloc(nodos, sizeof(int));
42     g->componentes.conexas = nodos;
43     g->aristas = 0;
44     g->adyacencia = (int **)malloc(sizeof(int *) * nodos);
45
46     for(i = 0; i < nodos; i++) {
47         g->adyacencia[i] = (int *)calloc(nodos, sizeof(int));
48     }
49     for(i = 0; i < nodos; i++) {
50         g->componente_conexa_nodos[i] = i;
51     }
52     return g;
53 }
54
55 void liberar_grafo(Grafo *g)
56 {
57     int i;
58
59     if(!g)
60         return;
61
62     free(g->componente_conexa_nodos);
63     for(i = 0; i < g->nodos; i++) {
64         free(g->adyacencia[i]);
65     }
66     free(g->adyacencia);
67     free(g);
68 }
69
70 int agregar_arista(Grafo *g, Nodo nodo1, Nodo nodo2)
```



```

71 {
72     int i;
73     int nueva_componente, vieja_componente;
74
75     if (!g)
76         return -1;
77
78     if (nodo1 >= g->nodos || nodo2 >= g->nodos)
79         return -1;
80
81     g->adyacencia[nodo1][nodo2] = 1;
82     g->adyacencia[nodo2][nodo1] = 1;
83     g->aristas++;
84
85     if (g->componente_conexa_nodos[nodo1] != g->componente_conexa_nodos[nodo2]) {
86         if (g->componente_conexa_nodos[nodo1] <= g->componente_conexa_nodos[nodo2]) {
87             nueva_componente = g->componente_conexa_nodos[nodo1];
88             vieja_componente = g->componente_conexa_nodos[nodo2];
89         }
90         else {
91             nueva_componente = g->componente_conexa_nodos[nodo2];
92             vieja_componente = g->componente_conexa_nodos[nodo1];
93         }
94
95         for (i = 0; i < g->nodos; i++) { //actualizo la componente conexa de todos los nodos que
96             pertenecian a esa componente
97             if (g->componente_conexa_nodos[i] == vieja_componente) {
98                 g->componente_conexa_nodos[i] = nueva_componente;
99             }
100         }
101         g->componentes_conexas--;
102     }
103     return 0;
104 }
105 int cantidad_componentes_conexas(Grafo *g)
106 {
107     if (!g)
108         return -1;
109     return g->componentes_conexas;
110 }
111
112 int cantidad_aristas(Grafo *g)
113 {
114     if (!g)
115         return -1;
116     return g->aristas;
117 }
118
119 int cantidad_nodos(Grafo *g)
120 {
121     if (!g)
122         return -1;
123     return g->nodos;
124 }
125
126 int son_adyacentes(Grafo *g, Nodo nodo1, Nodo nodo2)
127 {
128     if (!g)
129         return 0;
130
131     return g->adyacencia[nodo1][nodo2];
132 }
133
134 //retorna un vector con cantidad_componentes_conexas() elementos, en cada iesimo elemento, hay
135 //un nodo correspondiente a la componente iesima. liberar el resultado con free()
136 Nodo *nodos_de_componentes(Grafo *g)
137 {
138     Nodo *nodos;
139     int i;
140
141     if (!g)
142         return NULL;
143
144     nodos = (Nodo *) malloc(sizeof(Nodo) * g->nodos);
145     for (i = 0; i < g->nodos; i++) {
146         nodos[i] = -1;
147     }

```

```

146     }
147     for (i = 0; i < g->nodos; i++){
148         nodos[g->componente_conexa_nodos[i]] = i;
149     }
150     return nodos;
151 }
152
153 int distancia(Ciudad *c1, Ciudad *c2)
154 {
155     if (!c1 || !c2)
156         return 0;
157
158     return (c1->x - c2->x) * (c1->x - c2->x) + (c1->y - c2->y) * (c1->y - c2->y); //el cuadrado
        de la distancia euclidiana
159 }
160
161 void ordenar_aristas(Arista *aristas, int n)
162 {
163     int m, i, j, k;
164     Arista *aux;
165
166     if (n <= 1)
167         return;
168     m = n / 2;
169     ordenar_aristas(aristas, m);
170     ordenar_aristas(aristas + m, n - m);
171     aux = (Arista *)malloc(sizeof(Arista) * n);
172     i = 0;
173     j = 0;
174     k = 0;
175     while (i < m || j < (n - m)){
176         if (j >= (n - m)){
177             memcpy(&(aux[k]), &(aristas[i]), sizeof(Arista));
178             i++;
179             k++;
180             continue;
181         }
182         if (i >= m){
183             memcpy(&(aux[k]), &(aristas[m + j]), sizeof(Arista));
184             j++;
185             k++;
186             continue;
187         }
188         if (aristas[i].distancia < aristas[m + j].distancia){
189             memcpy(&(aux[k]), &(aristas[i]), sizeof(Arista));
190             i++;
191             k++;
192             continue;
193         }
194         else{
195             memcpy(&(aux[k]), &(aristas[m + j]), sizeof(Arista));
196             j++;
197             k++;
198             continue;
199         }
200     }
201     memcpy(aristas, aux, n * sizeof(Arista));
202     free(aux);
203 }
204
205 Grafo *resolver(int k_centrales, Ciudad *ciudades, int n_ciudades, Nodo **centrales)
206 {
207     Grafo *g = NULL;
208     NodoDistancia *nodos = NULL;
209     Arista *aristas = NULL;
210     int i, agregados, distancia_minima = -1, nodo_minimo = -1, componentes;
211
212     if (k_centrales <= 0 || ciudades == NULL || n_ciudades <= 0 || centrales == NULL){
213         return NULL;
214     }
215
216     g = crear_grafo(n_ciudades);
217     nodos = (NodoDistancia *)malloc(sizeof(NodoDistancia) * n_ciudades);
218     aristas = (Arista *)malloc(sizeof(Arista) * (n_ciudades - 1));
219
220     nodos[0].agregado = 1;
221     nodos[0].distancia = 0;

```

```

222     nodos[0].nodo = 0;
223     for(i = 1; i < n_ciudades; i++){
224         nodos[i].agregado = 0;
225         nodos[i].distancia = distancia(&(ciudades[i]), &(ciudades[0]));
226         nodos[i].nodo = 0;
227     }
228
229     for(agregados = 0; agregados < n_ciudades - 1; agregados++){
230         distancia_minima = -1;
231         nodo_minimo = 0;
232         for(i = 0; i < n_ciudades; i++){
233             if(!nodos[i].agregado){
234                 if(distancia_minima == -1 || nodos[i].distancia < distancia_minima){
235                     distancia_minima = nodos[i].distancia;
236                     nodo_minimo = i;
237                 }
238             }
239         }
240
241         nodos[nodo_minimo].agregado = 1;
242         aristas[agregados].nodo1 = nodo_minimo;
243         aristas[agregados].nodo2 = nodos[nodo_minimo].nodo;
244         aristas[agregados].distancia = distancia_minima;
245
246         for(i = 0; i < n_ciudades; i++){
247             if(!nodos[i].agregado){
248                 if(nodos[i].distancia > distancia(&(ciudades[i]), &(ciudades[nodo_minimo]))){
249                     nodos[i].distancia = distancia(&(ciudades[i]), &(ciudades[nodo_minimo]));
250                     nodos[i].nodo = nodo_minimo;
251                 }
252             }
253         }
254     }
255
256     ordenar_aristas(aristas, n_ciudades - 1);
257
258     for(componentes = n_ciudades; componentes > k_centrales; componentes--){
259         agregar_arista(g, aristas[n_ciudades - componentes].nodo1, aristas[n_ciudades -
            componentes].nodo2);
260     }
261
262     *centrales = nodos_de_componentes(g);
263
264     free(nodos);
265     free(aristas);
266
267     return g;
268 }

```

### 5.3. Ejercicio 3

## 6. Apéndice: Entregable e instrucciones de compilacion y testing