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

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. Concluciones

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.

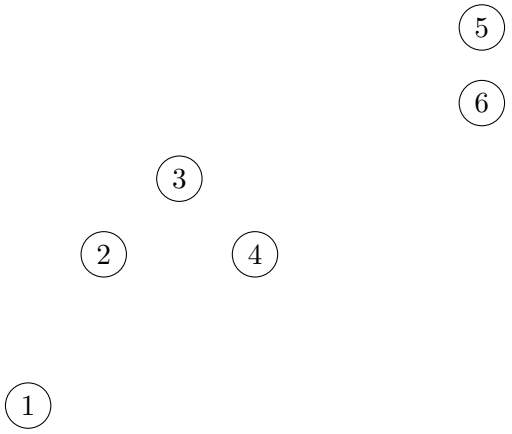


Figura 1: La entrada del problema, con las 5 ciudades y la cantidad de centrales

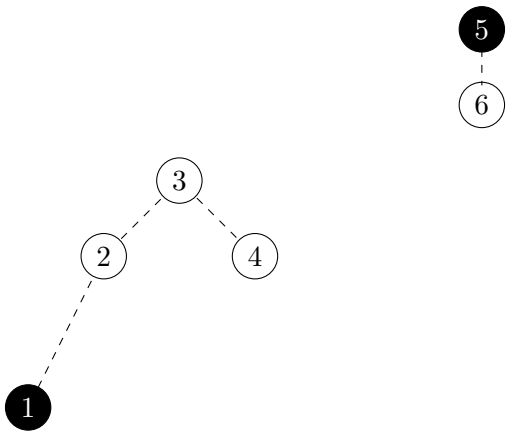


Figura 2: La solución al problema de la Figura 1, los nodos negros son los que contienen las centrales de distribución

2.2. Ideas para la resolución

Para la resolución del problema, se puede penzar 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 todos 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 **??** 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

El algoritmo propuesto lo que realiza es generar un *AGM* siguiendo al algoritmo de *Prim* (desde la Línea 5 a 30), y luego se ordenan las aristas del *AGM* para ir agregándolas siguiendo la idea de *Kruskal* y detenerse cuando se llega a k componentes conexas (desde la Línea 31 a 34).

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** MINIMIZARTUBERIAS(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)   ▷
    AgregarArista está especificado en el Algoritmo ??
34:     end for
35:     retornar ← g
36: end procedure
```

Una vez que tenemos el árbol, lo que tenemos que hacer es seleccionar un nodo cualquiera de cada componente conexa para colocar la central distribuidora. Para ésto al crear un nodo sin aristas, cada nodo va a estar asociado a una componente conexa distinta, y cada vez que se agrega una arista, se unen las componentes conexas de ambos nodos bajo una misma componente. Luego al tener el grafo armado, se recorre la lista de componentes conexas y se agarra un nodo para cada componente:

Algorithm 2 AgregarArista

```
1: procedure AGREGARARISTA(Grafo g, Nodo n1, Nodo n2)
2:     NUEVAARISTA(g, n1, n2)        ▷ agrega la arista entre n1 y n2, sin actualizar las componentes
    conexas
3:     if COMPONENTECONEXA(g,n1) ≠ COMPONENTECONEXA(g,n2) then
4:         nueva ← COMPONENTECONEXA(g,n1)
5:         vieja ← COMPONENTECONEXA(g,n2)
```

```
6:          for n ∈ Nodos(g) do        ▷ Le asigno a todos los nodos de la componente conexa vieja, la nueva componente conexa
7:              if ComponenteConexa(g,n) = vieja then
8:                  SetearComponente(g,n,nueva)
9:              end if
10:         end for
11:     end if
12: end procedure
```

Y por último para obtener un Nodo para cada componente conexa, vamos a crear un array con n elementos, cada posición i representa a la componente conexa i, y adentro puede tener un valor nulo representando que esa componente conexa no existe, o el valor de un nodo, entonces se va a retornar para cada componente conexa existente, un nodo, para que esos nodos representen las centrales:

Algorithm 3 NodosDeComponentes

```
1: procedure NodosDeComponentes(Grafo g)Nodo[CantidadNodos(g)]
2:     Nodos nodos[CantidadNodos(g)]
3:     for i = 0; i < CantidadNodos(g); i++ do ▷ Inicializo todas las componentes conexas (que son como máximo igual a la cantidad de nodos) asignandole ningún nodo
4:         nodos[i] = nil
5:     end for
6:     for n ∈ g do                ▷ para cada nodo, le asigno dento del array nodos tomando como índice la componente conexa el valor del nodo. Así al finalizar el ciclo sólo las componentes conexas existentes y con nodos tienen un valor dentro del array nodos, y va a tener el valor del último nodo correspondiente a esa componente conexa, ya que nodos de la misma componente se van pisando en el array
7:         nodos[ComponenteConexa(g,n)] ← n
8:     end for
9:     return nodos
10: end procedure
```

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 arisas 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).

$$P_p(g) = \sum_{i=0}^{i<n-1} \pi(a_i)$$

$$=$$

$$P_k(g) = \sum_{i=0}^{i<n-1} \pi(a_i)$$

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 mmismas para ambos algoritmos pero a_j y a_{j+2} son diferentes, entonces:

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

$$=$$

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

Simplificamos la igualdad de las suma, nos queda:

$$\pi(a') + \pi(b')$$

$$=$$

$$\pi(a) + \pi(b)$$

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 suposción que un algoritmo cambiaria una arista por otra con peso dinstinto.

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.

□

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 conzas, 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 llego a k componentes conezas. Supongamos que tenemos m aristas ordenadas de menor a mayor por su peso, $(a_0, a_1..a_i..a_{m-1})$, 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:

$$(\forall 0 \leq i < j) a_i \leq a_j$$
$$\wedge$$
$$(\forall j < i < m) a_i \geq a_j$$

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 terner 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 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, manteniendose 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.

□

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 $O(n)$ ya que agrega la relación en la matriz de adyacencia en $O(1)$ y luego tiene que actualizar las componentes conexas en $O(n)$ quedandonos una complejidad de $O(n * (n - k))$. Notar que si $k > n$, no se realiza ninguna iteración.

- Luego de tener el grafo solución, tenemos que obtener 1 nodo de cada componente, eso se realiza con la función *NodosDeComponentes* que inicializa un array de n elementos y luego recorre todos los nodos (n nodos) una vez, quedando $O(n+n) = O(n)$

Bajo éste análisis, la complejidad nos queda (1)

$$\begin{aligned} & O(n) + (n-1) * (O(n) + O(1) + O(n)) + O(n \log n) + O(n * (n-k)) + O(n) \\ & = \\ & O(n) + O(n^2) + O(n) + O(n^2) + O(n \log n) + O(n * (n-k)) + O(n) \\ & = \\ & O(3n) + O(2n^2) + O(n \log n) + O(n * (n-k)) \\ & = \\ & O(n^2) + O(n * (n-k)) \end{aligned} \tag{1}$$

Como el $O(n*(n-k))$ está acotado por $O(n^2)$ (cuando $k = 0$), entonces nos queda que la complejidad del algoritmo es:

$$O(n^2)$$

Cumpliendo así la complejidad requerida.

2.5. Mediciones de pérformance

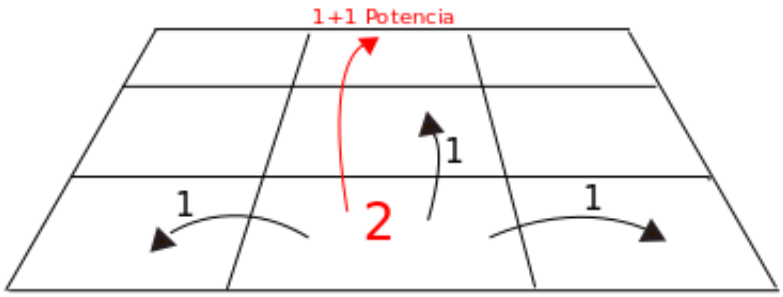
Para medir la pérformance del algoritmo, se crearon casos de prueba como se especifa en el Apéndice de casos de prueba (Sección 4) y se comparó contra la complejidad $O(n^2)$ calculada en la Sección 2.4

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 tamano n x 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. Éstan 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 does celdas hacia adelante, pero su reserva de potencias se reduciría en uno.

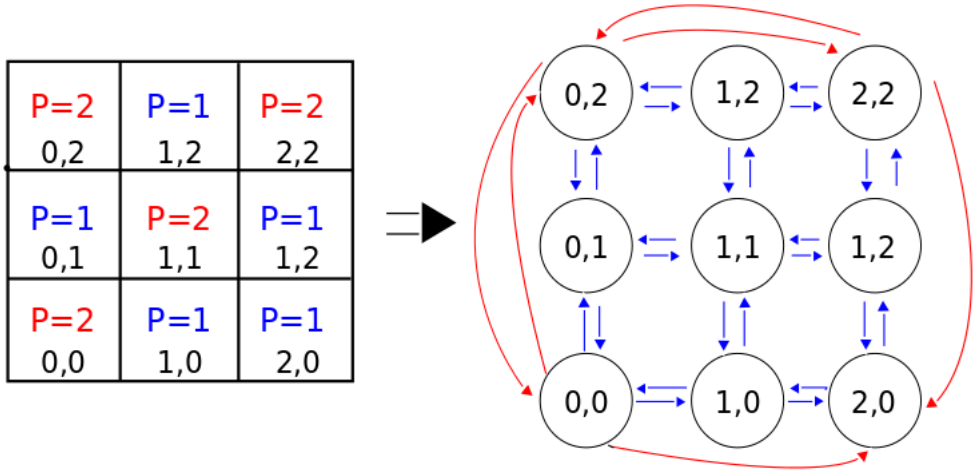


El objetivo del juego es llegar de una celda origen a otra destino, realizando la menor cantidad de saltos posibles. Por esto, se pide el diseno e implementación de un algoritmo que calcule y de como resultado uno de los caminos que lleguen de origen a destino cuya cantidad de saltos sea mínima.

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 resolucioón 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:

$$G_L = (V_L, E_L) / \; V_= V_{1_L}, ..., V_{n_L} = \{casillas \; 1..n \; con \; L \; potencias \; disponibles\}$$

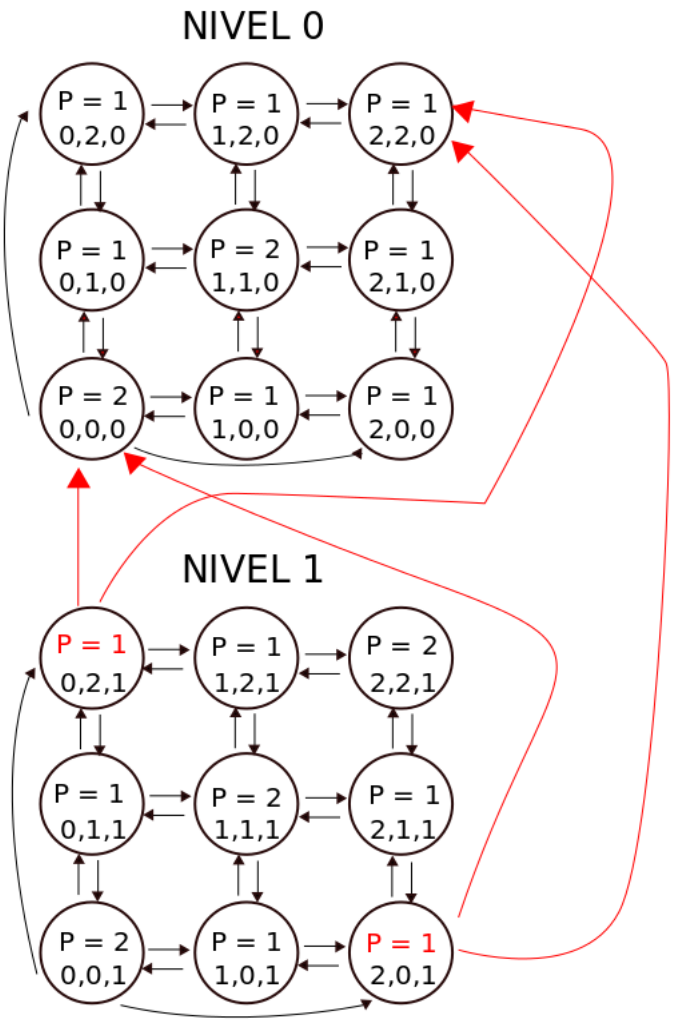
Por lo tanto, el grafo G es:

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

Donde T son las aristas entre niveles, descriptas a continuación:

Se tienen aristas dirigidas en cada nivel tal que si $v, u \in V_t$, $(u, v) \in E_t$ 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 $(j-m) + pot(v)$.

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 realize 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. Concluciones

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 ejercucio 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 resolvero 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

```c
typedef int Nodo;

struct GrafoMatrizAdyacencia{
    int nodos; //cantidad de nodos
    int *componente_conexa_nodos; //array para asignar componentes conexas
    int componentes_conexas;
    int **adyacencia; //la matriz ed adyacencia
    int aristas; //aristas colocadas
};

typedef struct GrafoMatrizAdyacencia Grafo;

typedef struct Ciudad {
    Nodo nodo;
    int x;
    int y;
} Ciudad;

typedef struct NodoDistancia {
    int agregado;
    int distancia;
    Nodo nodo;
} NodoDistancia;

typedef struct Arista{
    Nodo nodo1;
    Nodo nodo2;
    int distancia;
} Arista;

Grafo *crear_grafo(int nodos)
{
    int i;

    Grafo *g = NULL;
    if(nodos <= 0)
        return NULL;

    g = (Grafo *)malloc(sizeof(Grafo));
    g->nodos = nodos;
    g->componente_conexa_nodos = (int *)calloc(nodos, sizeof(int));
    g->componentes_conexas = nodos;
    g->aristas = 0;
    g->adyacencia = (int **)malloc(sizeof(int *) * nodos);

    for(i = 0; i < nodos; i++) {
        g->adyacencia[i] = (int *)calloc(nodos, sizeof(int));
    }
    for(i = 0; i < nodos; i++) {
        g->componente_conexa_nodos[i] = i;
    }
    return g;
}

void liberar_grafo(Grafo *g)
{
    int i;

    if(!g)
        return;

    free(g->componente_conexa_nodos);
    for(i = 0; i < g->nodos; i++) {
        free(g->adyacencia[i]);
    }
    free(g->adyacencia);
    free(g);
}

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

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

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

5.3. Ejercicio 3

6. Apéndice: Entregable e instrucciones de compilacion y testing