DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Algoritmos y estructuras de datos III
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Federico De Rocco	403/13	fede.183@hotmail.com
Federico Nicolás Esquivel	915/12	alt.juss@gmail.com
Fernando Otero	424/11	fergabot@gmail.com

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

Índice

1. Ejercicio 1

1.1. Introducción

El problema consiste en hallar el camino que utilice la mayor cantidad posible de portales para llegar desde el piso 0 al N. En este contexto, los portales solo llevan a pisos superiores, nunca bajan. Se garantiza que existe un camino máximo entre los pisos mencionados y que no hay más de un portal que comunique el mismo par de pisos.

1.2. Desarrollo

A fin de encontrar el camino mas largo en tiempo cuadrático sobre la cantidad de pisos, planteamos una solución con programación dinámica. Podemos definir recursivamente el largo del camino con la máxima cantidad de portales atravesados para llegar a un piso dado hasta el piso n como:

$$P_N(p) = max_{p' \in suc(p)}\{1 + P_N(p')\}$$

Donde $suc(p)$ son los pisos que tienen un portal con origen en p.

Implementamos este comportamiento con una representación como listas de antecesores.

Partimos desde el último piso, indicándolo como accesible, e iteramos hacia abajo. Para cada piso, si es accesible desde el piso N, por cada antecesor calculamos el máximo entre el valor ya calculado (si lo hay) y la cantidad de portales hasta el enésimo piso sumado uno, y memorizamos este resultado. Podría existir un valor previo si el antecesor en cuestión también lo es para algún piso superior. Hecho esto, marcamos sus antecesores como accesibles y continuamos iterando hasta llegar al piso 0.

Cuando esto suceda, como sabemos que existe camino entre último piso y la planta baja, habremos calculado $P_N(0)$.

Implementamos este comportamiento con el siguiente algoritmo:

Pseudocodigo 1 Camino Máximo

```
1:  procedure CaminoMáximo
2:      pisos ← lista de antecesores
3:      accesible ← arreglo de tamaño N
4:      maximoDesdeN ← arreglo de tamaño N
5:      Inicializar accesible en 0 para todas sus posiciones
6:      Inicializar maximoDesdeN en 0 para todas sus posiciones
7:      accesible_N ← true
8:      para p = N hasta 0 hacer:
9:          piso ← pisos_i
10:         si accesible_p entonces:
11:             para antecesor en piso.antecesores hacer:
12:                 maximo ← máx{maximoDesdeN_piso.numero + 1, maximoDesdeN_antecesor}
13:                 maximoDesdeN_antecesor ← maximo
14:                 accesible_antecesor ← true
        devolver maximoDesdeN_0
```

En el código utilizamos una estructura propia para representar cada piso, que agrupa el número de piso y una lista de los números de los antecesores. Ésta lista se implementa como una lista enlazada, por lo que tiene costo de inserción constante. En el algoritmo descrito, *pisos* es una

arreglo de estas estructuras, ordenada por numero de piso cuando es construida. Esto tiene lugar de la siguiente manera:

Pseudocodigo 2 Inicialización Camino Máximo

```
    procedure INIT
2:      pisos ← arreglo de tamaño N
        para i = 0 hasta N hacer:
4:          antecesores ← nueva Lista de int
            piso ← nuevo Piso(i,antecesores)
6:          pisos_i ← piso
        para portal en portales hacer:
8:          pisos_{portal.hasta}.antecesores.agregarAtras(portal.desde)
```

1.3. Correctitud

Como iteramos desde el piso N al 0, partiendo de $P_N(N) = 0$, cuando estemos iterando sobre el piso p, si hay portales que lo conecten con el último piso, ya tendremos $P_N(p)$ correctamente calculado y memorizado. Podemos afirmar esto porque como los portales solo suben, el piso p no puede ser antecesor de ningún piso sobre el cual nos falte iterar, pues como puede verse en el algoritmo 1, $maximoDesdeN$ (el vector donde memorizamos los resultados de P_N) solo se modifica en las posiciones correspondientes a antecesores del piso sobre el cual se itera.

Por lo tanto, ya habremos calculado $P_N(p_s)$ para todos los sucesores p_s del piso p y memorizado el máximo entre ellos.

Una vez que la iteración sobre los pisos termina, sea $l = P_N(p)$ el largo de la ruta desde el piso p al N, supongamos que no es máxima. Si esto sucede, hay un camino desde este piso al último que usa al menos un portal más. Entonces existe p' sucesor de p, tal que $P_N(p') \geq l$.

Por otro lado l es el máximo de todos los caminos entre los sucesores de p y N, sumado uno, por lo que podemos decir que $l > P_N(p_s)$ para todo p_s sucesor de p. En particular, al ser sucesor de p, $l > P_N(p')$, lo que lleva a $l > P_N(p') \geq l$. Luego, esta suposición es absurda.

1.4. Complejidad

Como no hay portales de un piso superior a uno inferior, para cada piso sus antecesores serán pisos debajo de el. Luego, para un piso p, sus antecesores serán a lo sumo $p - 1$ pisos.

De acuerdo a lo expuesto en el algoritmo descrito en el pseudocódigo 1, entre las lineas 8 y 14 iteramos sobre la cantidad de pisos. Asimismo dentro de cada ejecución del ciclo, si el *p-ésimo* piso es accesible, repetimos las lineas 12 a 14 por cada antecesor. Es decir, estas operaciones se repiten $p-1$ veces. Cada una de estas sentencias se ejecuta en tiempo constante, ya que son comparaciones, asignaciones o accesos sobre arreglos. Por lo tanto, sumando el costo de las operaciones tenemos:

$$\sum_{p=1}^{N}(p-1) = \frac{1}{2}(N-1)n$$

Luego, podemos acotar el orden de complejidad de la ejecución del algoritmo por $O(\frac{1}{2}(N-1)N)$ que es equivalente a $O(N^2)$.

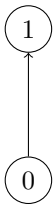
Por otro lado, el costo de inicializar las estructuras requeridas por el algoritmo usando el procedimiento descrito en el pseudocódigo 2 es $O(N + P)$, con P la cantidad de portales. En esta cota consideramos que el costo de inicialización del arreglo *pisos* es $O(N)$, y crear una lista vacía y el par que la relaciona al número de piso tiene costo constante.

De igual manera, el acceso sobre el vector de pisos y la inserción sobre la lista de antecesores tienen costo $O(1)$, y al repetirse una vez por cada portal, suman P a la complejidad. La cantidad de portales esta contemplada en el algoritmo como la suma de los antecesores de cada piso, que como establecimos anteriormente, es $\frac{1}{2}(N-1)N$, que podemos acotar asintóticamente por $O(N^2)$.

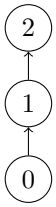
Finalmente, el costo de inicializar y ejecutar algoritmo se puede acotar por $O(N^2)$.

1.5. Experimentación

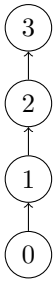
Se plantean una serie de casos de prueba para los cuales esperamos obtener resultados correctos en base a lo expuesto en las secciones anteriores. Las primeras tres pruebas se basan en generar un camino simple en el cual todos los nodos sean necesarios para llegar desde el primer piso al último. Los siguientes grafos son las representaciones de estos casos. El punto de partida es el nodo 1 y el de llegada es el que tenga mayor valor.



Resultado: 1

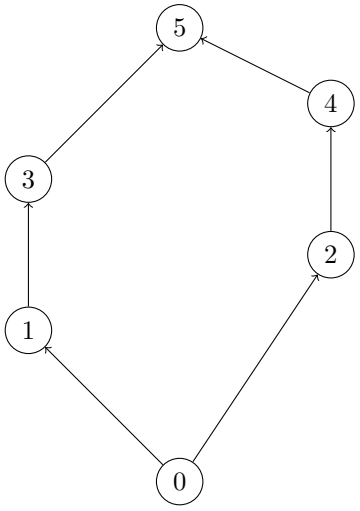


Resultado: 2



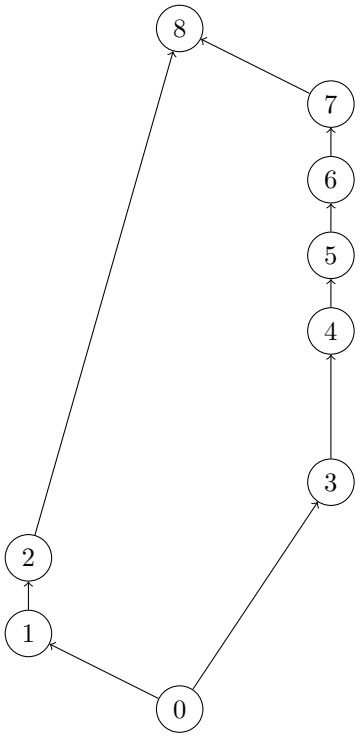
Resultado: 3

Para el siguiente experimento buscamos observar el funcionamiento cuando existen dos caminos con el mismo largo.



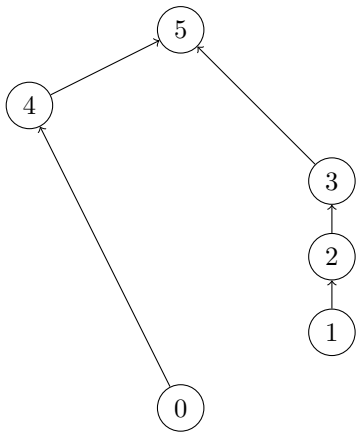
Resultado: 3

Después repetimos lo anterior pero con la diferencia que los dos caminos son distintos en longitud.

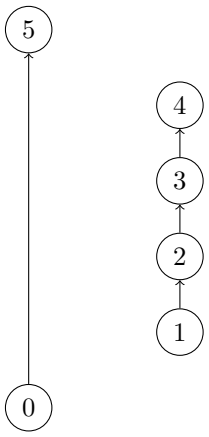


Resultado: 6 (Recorrerá el camino $0 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8$ que es el más largo).

Para los siguientes experimentos lo que haremos es tener por un lado un camino corto que nos lleva del origen al destino y otro camino que nos lleve a este, pero sin comenzar por el origen.



Resultado: 2.



Resultado: 1.

Como podemos ver la idea es que tenga por resultado la cantidad de portales que tiene que atravesar en el camino que comienza en el origen y concluye en el destino, y no los que no cumplan esta última propiedad.

Lo siguiente serán unos tests dedicados a la medición de tiempos. Graficaremos los casos por separado y con distintos tamaños ya que la diferencia de tiempos entre el mejor y peor caso es demasiado grande como para ponerlos en un solo gráfico, ya que se perderían detalles.

En estas mediciones no consideramos el tiempo de incialización, que consideramos parte de la lectura de la entrada.

Comenzemos por el peor caso el cual conseguimos mediante un grafo completo, por completo nos refirimos a que cada piso tiene un portal por cada uno de los siguientes. Con más posibilidades debemos que tener en cuenta todas para obtener el camino con mayor número de portales. Para el caso promedio simplemente nos armaremos un grafo aleatorio que partirá de la base que del caso anterior, osea que hay un camino que involucra a todos los nodos y agregaremos aristas entre los nodos de forma aleatoria. El tamaño de las muestra variara entre 200 y 290 con saltos de 10.

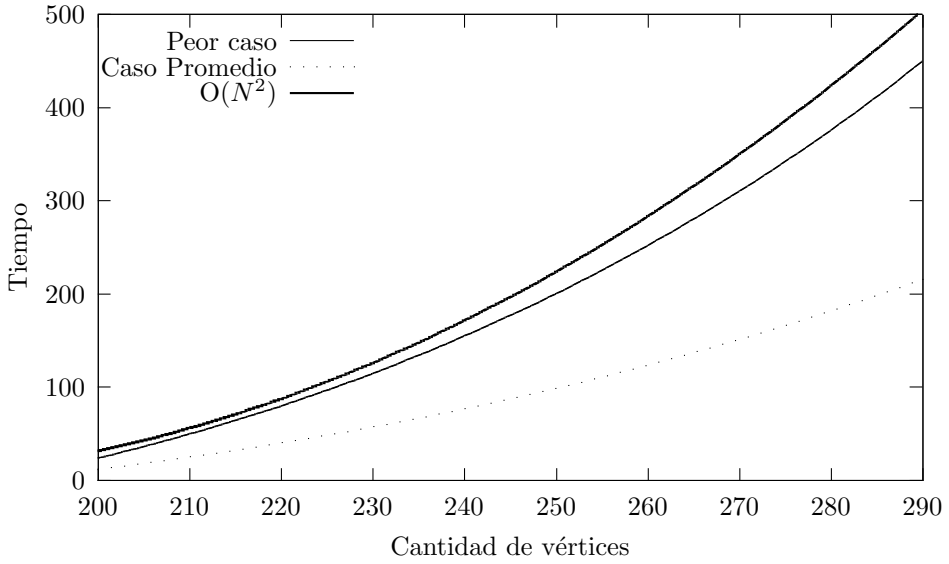


Figura 1: Comparación de tiempos variando la cantidad de pisos

Para el mejor caso, podríamos simplemente un grafo similar al del último test de correctitud. En otras palabras que el comienzo este unido con el fin y los nodos intermedios no influyan. Dado que este caso es poco interesante porque solamente hay una posibilidad y el tiempo sería constante, hemos decidido omitir este caso como trivial y asumir que el mejor caso a evaluar es en el cual exista un camino entre el primero y el último que pase por todos los nodos(semejante a los primeros tests de correctitud). Variaremos el tamaño de las muestras de 200 a 2000 con saltos de 200. Usamos esta muestra tan grande porque usando la anterior no nos permite apreciar suficiente diferencia.

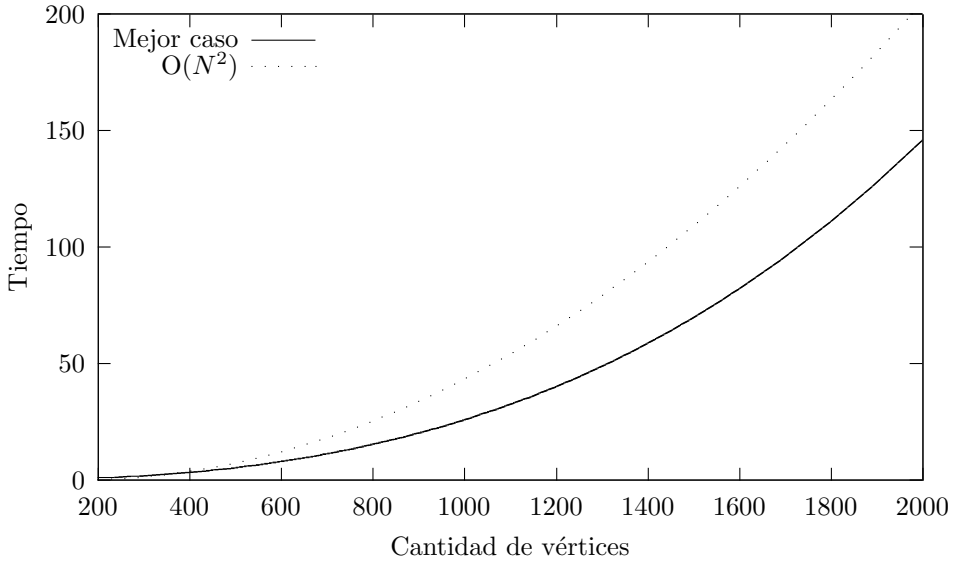


Figura 2: Comparación de tiempos variando la cantidad de pisos

2. Ejercicio 2

2.1. Introducción

En este problema buscamos minimizar el tiempo necesario para llegar desde el principio del pasillo en la planta baja al final del pasillo en el último piso. Se garantiza que existe al menos un camino entre estos puntos.

2.2. Desarrollo

Para resolver este ejercicio en la complejidad pedida consideramos usar un algoritmo Breadth-First Search. Sin embargo, como usar un portal tiene costo distinto a caminar en el mismo piso y los portales pueden estar a cualquier distancia entre si, necesitamos primero normalizar los costos. Para ello consideramos en lugar de un grafo con solo portales como vértices, uno donde cada posición de cada piso esta representada. De ésta forma, entre portales del mismo piso hay tantos vértices como posiciones intermedias, y atravesar cada una tiene costo 1, por lo que es suficiente contar la cantidad vértices. Como utilizar un portal tiene costo 2, agregamos un nodo intermedio entre las ubicaciones que comunica cada portal. De esta forma, la distancia entre cada posición y portal queda definida por la cantidad de vértices intermedios, con lo cual podemos aplicar BFS para obtener el resultado.

Para representar esta conversión, usamos una matriz de $N+1 \times L+1$ posiciones (incluimos desde la posición 0 a la L de cada piso), donde cada posición almacena su piso, metro, distancia hasta ella -inicialmente 0, indicando que no fue calculada- y sus vecinos. Para cada una, sus vecinos son los destinos de cualquier portal que este ubicado allí y las posiciones contiguas a izquierda y derecha, si existen. A partir de los parámetros de entrada N, L y la lista de portales, se construye esta matriz de acuerdo al siguiente procedimiento:

Pseudocodigo 3 Inicialización de Camino Mínimo

```
procedure INIT
    N ← cantidad de pisos
    L ← largo de pasillos
    portales ← lista de portales
    mapaPortales ←matriz de N + 1 filas y L + 1 columnas
    para i = 0 hasta N hacer:
        para j = 0 hasta L hacer:
            vecinos ← nueva lista vacía de Posicion
            pos ← nueva Posicion(i, j, vecinos)
            mapaPortales_{i,j} ← pos
    para i = 0 hasta N hacer:
        para j = 0 hasta L hacer:
            si j > 0 entonces:
                mapaPortales_{i,j}.vecinos.agregarAtras(mapaPortales_{i,j-1})
            si j < l entonces:
                mapaPortales_{i,j}.vecinos.agregarAtras(mapaPortales_{i,j+1})
    para cada portal en portales hacer:
        d ← portal.desde
        h ← portal.hasta
        mapaPortales_{d.piso,d.metros}.vecinos.agregarAtras(mapaPortales_{h.piso,h.metros})
        mapaPortales_{h.piso,h.metros}.vecinos.agregarAtras(mapaPortales_{d.piso,d.metros})
```

Mientras calculamos distancias entendemos que hay un portal entre una posición y su vecino si están en diferentes pisos, o a mas de un metro de distancia en el mismo piso. Si hubiera un portal entre dos ubicaciones contiguas no lo consideramos, dado que es más barato caminar entre ellas. Cuando existe un portal, encolamos el vértice intermedio, cuyo único vecino es el nodo destino del portal al que corresponde.
Se implementa este comportamiento por medio del siguiente algoritmo:

Pseudocodigo 4 Camino Mínimo

```
    procedure CaminoMinimo
        mapaPortales ← matriz de N filas y L columnas construida con el procedimiento 3
        verticesRestantes ← cola de Posicion
4:      verticesRestantes.encolar(mapaPortales_{0,0})
        mientras verticesRestantes no vacía hacer:
            pos ← verticesRestantes.desencolar()
            para Posiciondestino en pos.vecinos hacer:
8:              si destino.distancia = 0 entonces:
                    si hayPortal(pos, destino) entonces:
                        portalBuffer ← nueva Posicion(destino.piso, destino.metro)
                        portalBuffer.distancia = pos.distancia + 1
12:                     portalBuffer.vecinos.agregarAtras(destino)
                        verticesRestantes.encolar(portalBuffer)
                    si no:
                        destino.distancia = pos.distancia + 1
16:                     verticesRestantes.encolar(destino)
        devolver mapaPortales_{N,L}.distancia
```

Al terminar el algoritmo, devolvemos el costo de llegar a la última posición del último piso.

2.3. Correctitud

Tal como describimos en la sección anterior, representamos los costos añadiendo nodos intermedios. El costo de moverse entre dos posiciones en un piso es de un segundo por metro, es decir, igual a la cantidad de metros. En el algoritmo propuesto, construimos la matriz $mapaPortales$ de tamaño $N + 1 \times L + 1$ donde cada elemento $mapaPortales_{i,j}$ representa la posición en el piso i a los j metros del inicio del pasillo. De esta manera, todos los metros de cada pasillo están representados en la matriz. Adicionalmente, como se puede observar en el pseudocódigo 3 (lineas 11 a 16), cada elemento tiene como vecinos a sus posiciones aledañas.

Durante la ejecución del algoritmo tomamos cada elemento de la matriz como un vértice, por lo que la distancia en metros entre dos posiciones del mismo piso resulta ser igual a la cantidad de vértices intermedios.

Análogamente, al agregar un nodo intermedio para cada portal, igualamos el costo de utilizarlos al número de vértices a recorrer para llegar a destino agregando un nodo intermedio, de acuerdo a lo expuesto en el pseudocódigo 4 (lineas 9 a 13).

Finalmente, el algoritmo se reduce a BFS: Se encola un nodo inicial y mientras la cola no este vacía, se desencola un nodo n, se encolan todos sus vecinos y se le asigna a cada uno la distancia de n al origen más uno, si no estaba ya calculada. Por lo tanto, podemos asegurar que el algoritmo es correcto a partir de la correctitud de BFS, probada en la bibliografía [1].

2.4. Complejidad

Para la inicialización de la estructura en la que se basa el algoritmo, de acuerdo a lo descrito en el pseudocódigo 3, se itera sobre las $N+1$ filas y $L+1$ columnas de la matriz para inicializarla y luego asignar los vecinos adyacentes a cada posición. Finalmente, se itera sobre la cantidad de portales para añadir las posiciones conectadas a las listas de vecinos correspondientes. La lista de vecinos se implementa sobre una lista enlazada y sus inserciones se realizan en tiempo constante. De igual manera, la construcción de instancias del tipo *Posicion* también tiene costo $O(1)$. A partir de esto, podemos acotar el costo de inicialización por $O(NL + P)$

Dada la representación elegida, tenemos $NL + P$ vértices, uno por cada metro en cada pasillo, y uno intermedio por cada portal. Asimismo, la cantidad de aristas por cada posición $p_{i,j}$ es $2 + Portales(p_{i,j})$, donde i es el piso, j el metro y $Portales(x)$ la cantidad de portales con un extremo en la posición x.

Como cada portal es bidireccional, se representa en ambos extremos (ver pseudo 4, lineas 18 a 21), por lo que durante la ejecución de BFS se suma dos veces. Luego, la cantidad de aristas del grafo es $2(NL + P)$

Durante la ejecución del algoritmo, por cada vértice en la cola se ejecutan operaciones, que dadas las estructuras utilizadas son $O(1)$, tantas veces como vecinos tenga ese nodo. Si el grafo es conexo, como para cada vértice se encolan sus vecinos no visitados, el ciclo se ejecuta una vez por nodo. Si no fuera conexo, se visitarán menos vértices.

Entonces, cada ejecución del ciclo principal sobre un vértice v tiene un costo $O(1 + E_v)$, con E_v la cantidad de aristas incidentes a v. Como en el peor caso (grafo conexo) se visitan todos los nodos del grafo, la sumatoria de este costo sobre todos ellos es $O(|V| + |E|)$, donde V y E son los conjuntos de vértices y aristas del grafo respectivamente.

Como establecimos previamente, la cantidad de vértices del grafo que consideramos es $NL+P$, y la de aristas es $2(NL + P)$, por lo que el costo en peor caso suma $O((NL + P) + 2(NL + P))$, que queda acotado por $O(NL + P)$.

2.5. Experimentación

Dado que la complejidad temporal del código de inicialización es igual al del algoritmo, decidimos obviarlo de las mediciones de tiempos y concentrar nuestro análisis en el código que resuelve el problema.

Consideramos el grafo completo como el peor caso, ya que al ser conexo se analizan todos los vertices, y cada uno tiene la máxima cantidad de vecinos. En particular, al haber un portal entre el principio del pasillo en planta baja y el final en el último piso, el resultado esperado es 2. Al estar dado por la máxima cantidad de portales, que depende del tamaño del grafo ya que cada posición puede tener un portal a cada una de las $NL - 1$ posiciones restantes, observamos los tiempos variando NL.

Para el mejor caso consideramos que al tener como única restricción la existencia de un camino entre la primer posición de la planta baja y la última de la mas alta, basta incluir un portal desde el piso 0 al N. Elegimos un portal entre la posición inicial y final, con lo que el resultado esperado es el mismo al del caso anterior. Análogamente al anterior, este caso se da en la mínima cantidad de portales, por lo que realizamos el mismo análisis en función de NL.

En los experimentos realizados se observaron resultados similares al fijar una variable del producto NL, aumentando la otra para variarlo.

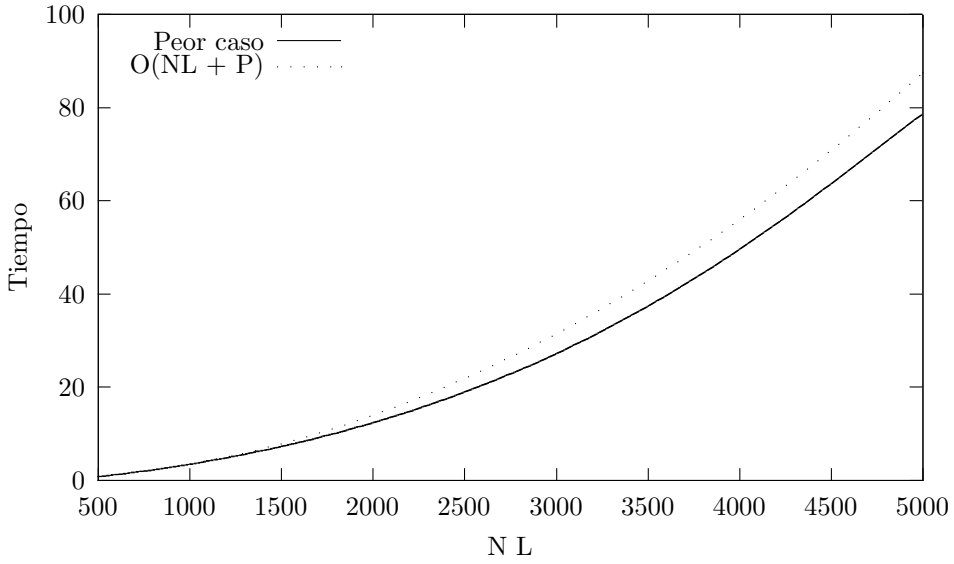


Figura 3: Comparación de tiempos variando NL

Omitimos de esta figura las mediciones del mejor caso, dado que involucra $2L$ vértices, pues la componente conexa que contiene la primer posición del piso 0 y la última del piso N esta compuesta por los nodos de ambos pasillos, sus tiempos de ejecución son mucho menores a los de peor caso, aun aumentando L para variar NL.

Como no realizamos optimizaciones sobre BFS para especializarlo al problema de la búsqueda de camino mínimo de uno a uno, el costo del algoritmo es el mismo para cualquier par de metros en los pisos elegidos, pues se analizan la misma cantidad de nodos y ejes. Esto es porque se calcula la distancia a todos los vértices de la componente conexa del nodo original. Debido a la forma de representar el problema que elegimos, en la ausencia de portales tenemos por cada piso una componente conexa. Al agregar portales entre pisos, las unimos.

Dado que para este algoritmo, una vez que se calcula la distancia a un nodo sabemos que su valor es final, podríamos detener la ejecución ni bien se calcule la distancia a la posición final. Para esto, agregamos al algoritmo descrito en el pseudocódigo 4 un chequeo sobre la distancia a la posición en $mapaPortales_{N,L}$ luego de calcular la distancia para cada vecino de una posición, y devolvemos ese valor si ya fue calculado.

La complejidad temporal sigue estando acotada por $O(NL + P)$, pero en ciertos casos mejora considerablemente el tiempo de ejecución.

Con esta modificación, entendemos que el peor caso se dará cuando se deban visitar todos los nodos antes de llegar al vértice objetivo. Esto sucede cuando se cuenta con un grafo completo entre los pisos 0 y N-1, con el último piso conectado en su posición 0 con algún nodo del anterior.

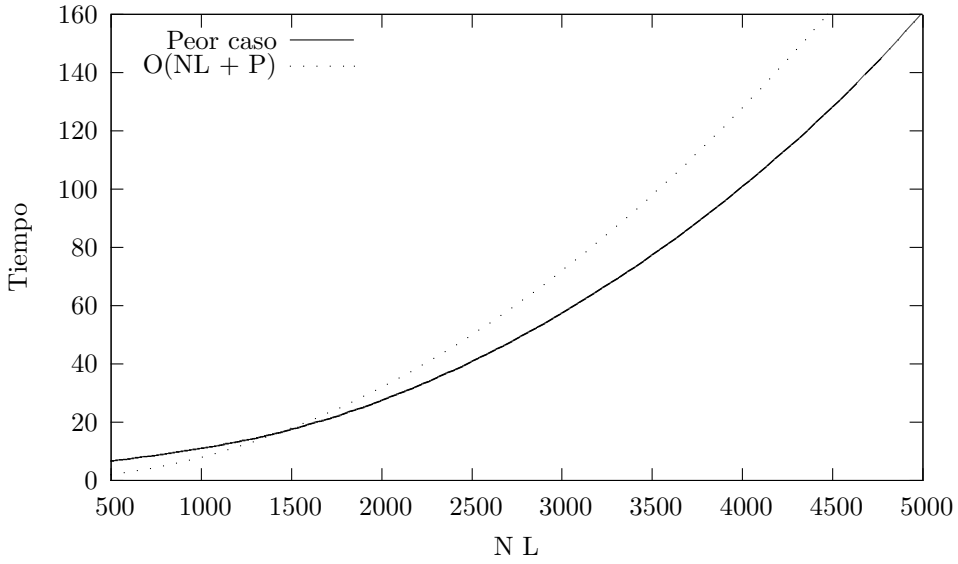


Figura 4: Comparación de tiempos variando NL

Nuevamente omitimos de la figura el mejor caso. Con esta modificación, en ese escenario solo se calcula la distancia al vértice objetivo independientemente de la cantidad de vértices o portales, por lo que su tiempo de ejecución es casi inmediato.

3. Ejercicio 3

3.1. Introducción

El problema consiste en obtener la mínima suma posible de las longitudes de los pasillos que debemos clausurar para que no existan ciclos en el grafo y además sea conexo.

3.2. Desarrollo

Lo que se nos pide sobre el grafo(debe ser conexo y no tener ciclos), es similar a buscar un árbol generador mínimo, con el detalle que tenemos que quedarnos con los pasillos con mayor costo de cierre. Para lograr esto tenemos el algoritmo de Kruskal que, utilizando una estructura de Union-Find, podemos resolver correctamente este problema.

Esta estructura utiliza dos arreglos donde se almacenan rangos y padres de cada nodo, utilizados para las heurísticas de path-compression y unión pesada. Dado un tamaño del conjunto, se inicializan los arreglos colocando en cada posición a si misma como padre y 0 como rango.

Para obtener el representante del grupo de un elemento(*find*) usamos el siguiente procedimiento:

- Obtenemos el padre del elemento.
- Mientras el padre no sea igual a su padre(el único elemento de un conjunto que tiene como padre a si mismo es el representante) buscamos el siguiente padre.
- Asignamos al elemento al representante que encontramos en el anterior.
- Devolvemos el representante.

Para unir dos conjuntos disjuntos(*union*) realizamos los siguientes pasos:

- Conseguimos los representantes de ambos conjuntos(usando *find*).
- Si son iguales, ambos elementos pertenecen al mismo conjunto.
- Caso contrario:
  - Conseguimos el rango de los dos y comparamos.
  - Si el rango de uno es mayor que el de otro entonces el representante de ese pasa a ser el representante del representante del otro, en otras palabras ahora son un solo conjunto.
  - Si son iguales, desempatamos a favor del primero e incrementamos su rango.

Gracias a la unión pesada y al path compression, estas operaciones tienen un costo cercano a $O(1)$.

Para resolver el problema, inicializamos primero las estructuras de soporte. A partir de la lista de pasillos de entrada, construimos un *max-heap* con todos sus elementos y obtenemos, a partir del máximo extremo nombrado en la lista de pasillos, la cantidad de esquinas. Adicionalmente, construimos una estructura union-find con tamaño igual a la cantidad de esquinas.

A partir de estas estructuras utilizamos el algoritmo de Kruskal con la particularidad de invertir el orden en que se recorren las aristas, dada por la utilización de un *max-heap* en lugar de un *min-heap*.

Implementamos este algoritmo de acuerdo al siguiente pseudocódigo:

Estructuras que usamos:
grafoUnionFind Es una estructura Union-Find.
colaDePasillos Es un *max-heap* donde tenemos a todos los pasillos de nuestro grafo.

Pseudocodigo 5 Desarmar Ciclos(Kruskal)

```
    procedure INICIALIZACIÓN(Lista(Pasillos) pasillos)
        n ← 0                                                          ▷ O(1)
        para cada Pasillo pasillo en pasillos hacer:                   ▷ O(MLogM)
            n ← máx{n, máx{pasillo.extremo1, pasillo.extremo2}}        ▷ O(1)
            colaDePasillos.encolar(pasillo)                            ▷ O(LogM)
6:      grafoUnionFind ← CrearUnionFind(n)   ▷ O(N) Con N la cantidad de esquinas
```

Pseudocodigo 6 Desarmar Ciclos(Kruskal)

```
    procedure DESARMAR
        sumaPasillosEliminados ← 0                                     ▷ O(1)
        mientras colaDePasillos no vacía hacer:
            pasillo ← colaDePasillos.desencolar()                      ▷ O(LogM)
            e1 ← pasillo.extremo1                                      ▷ O(1)
            e2 ← pasillo.extremo2                                      ▷ O(1)
7:          si grafoUnionFind.conectado(e1, e2) entonces:              ▷ O(1)
                sumaPasillosEliminados ← sumaPasillosEliminados + pasillo.longitud  ▷ O(1)
            si no:
                grafoUnionFind.union(e1, e2)   ▷ O(1) por path compression y union pesada
        devolver sumaPasillosEliminados
```

Para cada pasillo, ordenado de mayor a menor por el *max-heap*, controlamos si sus extremos están en el mismo conjunto Union-Find. Si es así, ese pasillo forma un ciclo y lo descartamos. En caso contrario unimos los dos extremos para representar que el pasillo forma parte del grafo final.

3.3. Correctitud

La demostración de correctitud de este algoritmo es análoga a la de Kruskal para árbol generador mínimo, pues solo difiere en que se analizan las aristas en orden inverso, de acuerdo a lo expuesto en el pseudocódigo 6. Utilizamos la demostración de la teórica cambiando los detalles necesarios para adecuarlo al árbol generador máximo.

Para ver que el algoritmo construye un árbol generador, como en cada paso el subgrafo B elegido hasta el momento es generador y acíclico, basta ver que el algoritmo termina con $m_B = n_G - 1$. Si $m_B < n_G - 1$, B es no conexo. Sea B_1 una componente conexa de B. Como G es conexo, va a existir alguna arista con un extremo en B_1 y el otro en V(G) − B_1, que por lo tanto no forma ciclo con las demás aristas de B. Entonces, si $m_B < n_G - 1$, el algoritmo puede realizar un paso más.

Sea G un grafo, T_K el árbol generado por el algoritmo de Kruskal y e_1, e_2, . . . , e_{n-1} la secuencia de aristas de T_K en el orden en que fueron elegidas por el algoritmo de Kruskal. Para cada árbol generador T de G definimos l(T) como el máximo k ≤ n tal que ∀j < k, $e_j \notin$ T.

Ahora, sea T un AGM que minimizar l(T). Si l(T) = n, entonces T coincide con T_K , con lo cual T_K resulta ser máximo. Si T_K no es máximo, entonces l(T) < n y $e_{l(T)} \notin$ T. Como T es conexo, en T hay un camino C que une los extremos de $e_{l(T)}$.

Como T_K es acíclico, hay alguna arista e en C tal que e $\notin T_K$. Como e_1, . . . , $e_{l(T)-1} \in$ T y T es

acíclico, e no forma ciclos con $e_1, \ldots, e_{l(T)-1}$. Por la forma en que fue elegida $e_{l(T)}$ por el algoritmo de Kruskal, peso($e_{l(T)}$) $\geq$ peso(e). Pero entonces T_0 = T − e ∪ $e_{l(T)}$ es un árbol generador de G de peso mayor o igual a T y l(T_0) < l(T), absurdo.

Luego T_K es un árbol generador máximo.

3.4. Complejidad

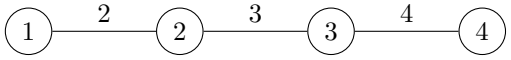
La complejidad de consturir las estructuras es de $O(MlogM)$, de acuerdo a lo expuesto en la sección 3.2 y el pseudocódigo 5. Como el grafo original es conexo la cantidad de nodos es menor o igual a la cantidad de aristas mas 1. De esta manera, podemos acotar $O(N)$ con $O(M)$, y la complejidad de la inicialización queda dominada por la construcción del *max-heap*.

Para la resolución usando Kruskal, de acuerdo a lo que señalamos en la sección de desarrollo (pseudocódigo 6), consideremos que la complejidad de las operaciones de cada paso del ciclo se acota por $O(LogM)$, con M cantidad de aristas. Esto es lo que nos cuesta sacar el próximo elemento de la cola, la operación de mayor costo. Esto se repite para todas las aristas, por lo tanto la complejidad de resolver por Kruskal es O(M Log M). Por último tenemos que la complejidad de armar la estructura y de resolver por Kruskal es, en peor caso, O(M Log M). Por lo tanto la complejidad con la que se resuelve el problema es de O(M Log M)

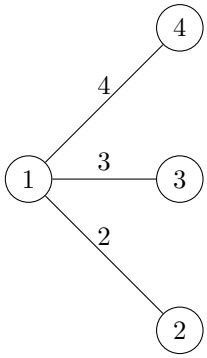
3.5. Experimentación

Se plantean una serie de casos de prueba para los cuales esperamos obtener resultados correctos en base a lo expuesto en las secciones anteriores.

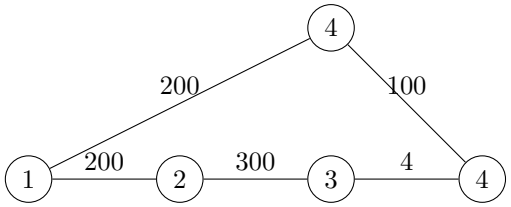
Para empezar haremos un test usando un árbol que se forma por un camino lineal entre los nodos (del 1 al 4). Como lo que buscamos es el árbol generador entonces si el grafo dado ya es un árbol entonces el resultado de aplicarle el algoritmo es 0 ya que no cerramos ningún camino.



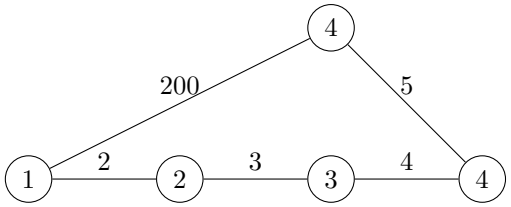
Para el segundo haremos lo mismo pero esta ves el árbol será un vértice unido a todas las demás aristas. Nuevamente el resultado es el mismo.



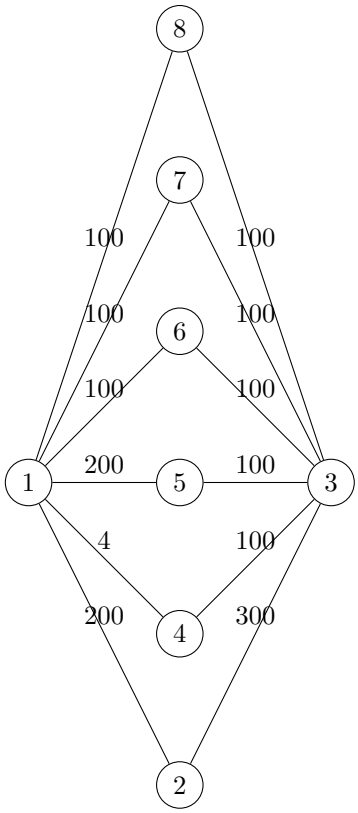
En el tercero usaremos un grafo que sea un circuito simple con una arista menos pesada que las otras. Lógicamente se eliminarla esta sola y el resultado será el peso de la misma.



El siguiente es similar al anterior solo que usando una arista muy pesada y las otras poco pesadas. Nuevamente tenemos el mismo resultado, se elimina la menos pesada.



Por último tenemos un caso de circuitos múltiples. Variaremos los pesos de las aristas, dejando una mucho menos pesada que las otras.



El resultado de este último es 404 que es lo que sale de haber cortado el pasillo de costo mínimo(4) y 4 pasillos de costo 100. Y con esto tenemos los distintos casos que Kruskal resuelve correctamente, los demás grafos se conformaran de combinaciones de los anteriores entonces, si el algoritmo resuelve correctamente ejemplos similares a estos, podrá resolver los demás.

Para hacer las mediciones de tiempos generaremos los casos mejor y peor, luego tomaremos el tiempo de cada uno ejecutándolo 100 veces y tomando el promedio para garantizar que la medición es creíble. Variaremos la cantidad de vértices de los grafos generados así como las aristas que los unen. En estas mediciones obviamos el costo de preparar las estructuras, dado que tiene la misma complejidad temporal que el algoritmo que resuelve el problema.

El peor caso en nuestro algoritmo de Kruskal es cuando debemos conectar muchos de los vértices en el Union-Find. En otras palabras, no tenemos que quitar a nadie del grafo para que este sea un árbol. La idea más obvia es usar un grafo que conecte linealmente los vértices, el primero con el segundo , el segundo con la tercero y así hasta llegar al último. Los pesos de las aristas van variando entre más pesados y menos pesados para hacer más costoso el ordenamiento en la cola de prioridad. Para el mejor caso debemos considerar que, contrario a lo anterior, el mejor escenario es cuando tenemos que conectar pocas aristas respecto al total. La idea a la que se llego es un grafo completo, ya que debemos conectar n-1 aristas en comparación a las n-1*n/2 que tenemos para formar un árbol. Utilizamos todas las aristas con el mismo peso para no tener impacto en el orden.

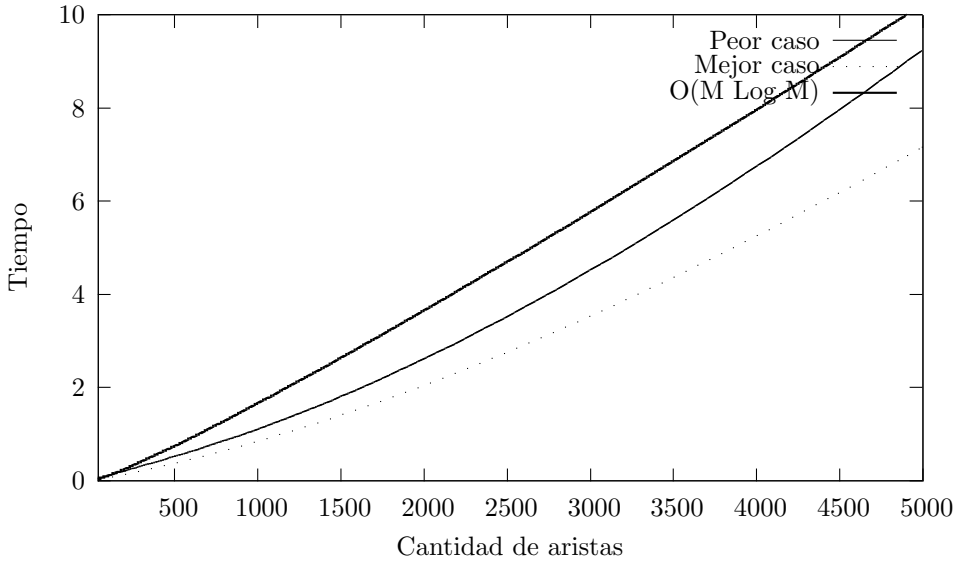


Figura 5: Comparación de tiempos variando la cantidad de aristas, los valores del gráfico están escalados. El escalado es de 1:100

Para generar estas variaciones lo que hacemos es, en el mejor caso, tenemos un grafo completo lo que significa que para una cantidad de 100 vértices tendremos 4950 aristas. Variamos la cantidad de vértices de 100 a 1000 con saltos de 100 para armar estos ejemplos. Para el peor caso tenemos un grafo en linea, por lo tanto tomamos las variaciones del anterior y le aplicamos la función $i(i-1)/2$ para asegurarnos que los dos tienen la misma cantidad de aristas.

4. Bibliografia

Referencias

[1] Cormen, Thomas H, and Charles E Leiserson. *Introduction To Algorithms, 3Rd Edition.*