DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

9 de octubre de 2015

Algoritmos y Estructuras de Datos III
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Ignacio Manuel Lebrero Rial	751/13	ignaciolebrero@gmail.com
Kevin Fuksman	682/13	kfuksman@hotmail.com
Damián Fixel	512/13	damianfixel@gmail.com
Federico Sebastían Sassone	602/13	fede.sassone@hotmail.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) 4576-3359
http://www.fcen.uba.ar

Índice General

1 Ejercicio 1

1.1 Problema: Saliendo del freezer

Estamos en el año 2048 y el Pabellón 0+infinito es todo un éxito. Los alumnos de Algoritmos III están contentos porque van a cursar este cuatrimestre en un aula que está en el piso N, que es el más alto de todos. Con los avances de la ciencia y la tecnología, escaleras y ascensores han quedado obsoletos, y la forma de subir de un piso a otro es a través de portales. El nuevo pabellón tiene P portales, cada uno de los cuales permite subir de un piso A a un piso más alto B (para bajar de piso hay que tirarse con paracaídas al piso 0 y luego volver a subir de ser necesario). Uno de los alumnos, que estaba cursando en el segundo cuatrimestre de 2015 y fue congelado por el método de criogenia, acaba de ser descongelado y no puede creer lo buenos que están estos portales, algo que en su época no existía. Luego de completar todos los censos de estudiantes desde el año 2016 en adelante, este alumno quiere usar la mayor cantidad de portales posibles para llegar al piso N y así seguir cursando Algoritmos III. Diseñar un algoritmo de complejidad $\mathcal{O}(N^2)$ para calcular la mayor cantidad de portales que puede utilizar el alumno para subir desde planta baja al piso N (sin tirarse nunca con paracaídas). Se asegura que en toda instancia del problema es posible realizar el recorrido deseado, y que no hay más de un portal que comunique el mismo par de pisos.

1.1.1 Explicacion del Problema

Para este problema se cuenta con un grafo dirigido, donde cada vertice pertenece a un piso y sus ejes conectan con vertices pertenecientes al mismo piso, en cuyo caso seran bidireccionales o unidireccionales en caso contrario.

El objetivo sera, dadas las conexiones de vertices de distintos pisos entre si, como moverse desde el vertice ubicado en el metro 0 del piso 0 hasta algun vertice en el piso N, de manera que utilice la mayor cantidad de vertices posibles.

Observar que como los vertices de distintos pisos estan conectados de manera unidireccional, no se generaran ciclos dentro del grafo.

1.1.2 Explicacion del Desarrollo

Para conseguir la cantidad maxima de portales que se pueden utilizar para ir desde el piso 0 hasta el ultimo desarrollaremos un algoritmo basado em programacion dinamica. Dentro de una matriz, las filas tendran el inicio del portal y las columnas tendran el destino. La parte inferior de la diagonal no se calculara porque no podemos dirigirnos hacia abajo.
Luego se inicializara la matriz completa en -2, a exepcion de la diagonal que tendra 0 (ya que la cantidad de portales para moverse entre el mismo piso es 0).
Una vez iniciada la matriz llamada M, para un portal P que comunica los pisos p1 y p2, pondremos en $M[p1, p2]$ el valor -1. Quedando la matriz conformada de la siguiente manera:

Las celadas inferiores a la diagonal, no interesan.
Las celdas en la diagonal, tienen valor 0
Las celdas superiores a la diagonal tienen valor -2 a menos que exista un portal que conecte esos pisos. En ese caso el valor es -1

Luego de armar la matriz:
1)Para cada piso i se pregunta si se conecta con el ultimo. En caso afirmativo se llena el vector $mejores[i]$ con 1. Caso contrario con -2.

2) se recorre la matriz de abajo hacia arriba y de derecha a izquierda y se completa de la siguiente manera:

Sean i,j los indices de la actual iteracion, si existe un portal desde el piso i al piso j y ademas el piso j esta conectado al ultimo piso, entonces encontramos una posible forma de ir desde el piso i hasta el ultimo: trasladarse del i al j, y luego el camino correspondiente entre j y el ultimo. Tambien sabemos que la longitud del camino correspondiente de ir del piso i al ultimo piso es igual a la longitud de ir de j al ultimo piso, mas 1 (el portal que nos lleva de i a j).
Sin embargo no podemos asegurarnos que este sea el mejor camino, por lo que luego de realizar esta operacion, diremos que $mejores[i]$ es el maximo entre el valor que ya tenia (que era el maximo hasta el momento) y el nuevo valor descubierto.

1.2 Justificación y Complejidad

```
FOR i desde 0 hasta matriz.length        // O(N)
    FOR j desde 0 hasta matriz.length    // O(N)
        IF j == i THEN                   // O(1)
            matriz[i][j] = 0             // O(1)
        ELSE
            matriz[i][j] = -2            // O(1)
        ENDIF
    ENDFOR
ENDFOR
```

Costo final de inicializacion de estructuras : $\mathcal{O}(n^2)$.

```
FOREACH portales as portal           //
   O(cantidadPortales)
    matriz[portal.getFrom()][portal.getTo()] = -1
    //seteo un -1 en las celdas de la matriz que
       correspondan a pisos conectados O(1)
ENDFOREACH
```

El costo de la función anterior esta determinada por la cantidad de portales que haya en el edificio. La cantidad máxima de portales posibles es exáctamente la cantidad de aristas del grafo completo:
Habiendo n pisos (n nodos), $\frac{n*(n-1)}{2}$.
Luego, el costo de esta sección es de $\mathcal{O}(\frac{n-1}{2} * n) = \mathcal{O}(n)$.

```
FOR i desde 0 hasta matriz.length -1                //
   O(N)
    IF matriz[i][matriz.length -1] == -1   THEN     //
       O(1)
        matriz[i][matriz.length-1] = 1              //
           O(1)
        mejores[i] = 1                              //
           O(1)
    ELSE
        mejores[i] = -2                             //
           O(1)
    ENDIF
ENDFOR
//Ponemos 1 en [i][piso n] si el piso i se conectaba con
   el ultimo piso, para todo i.
```

Costo de esta sección : $\mathcal{O}(N)$

```
FOR i desde matriz.length -2 hasta 0
                          //  O(N)
    FOR j desde matriz.length -2 hasta i
                          //  O(N)
        IF matriz[i][j] == -1 && mejores[j] > 0   THEN
                 //  O(1)
            matriz[i][j] = mejores[j] + 1
                                  //  O(1)
            mejores[i] = Maximo(mejores[i],
               matriz[i][j])   //  O(1)
        ENDIF
    ENDFOR
ENDFOR
    RETURN mejores[0]     // O(1)
    // Si hay conexion en [i][j] y j se conectaba con el
       ultimo piso, la celda [i][j] vale lo que valia el
       vector[j]. Por como recorremos la matriz, vamos
       guardando los "portales maximos" en cada iteracion
```

Costo de esta sección : $\mathcal{O}(N^2)$

Si sumamos todas las secciones de nuestro algoritmo es facil notar que la complejidad es $\mathcal{O}(2 * n^2 + n)$, la cual podemos acotarla por $\mathcal{O}(n^2)$ que era la complejidad pedida.

1.3 Correctitud

Vemamos como se comporta, una vez inicializada la estructura, nuestra función recursiva F; que asigna valores a las celdas de la matriz para resolver el problema.

$$\text{F(i , j)} = \begin{cases} 1 & \text{si j == n \&\& matriz[i][j] == 1} \\ -2 & \text{si j == n \&\& matriz[i][j] != 1} \\ \max\limits_{0 \le k \le n} F(i,k) + H & \text{si } n \text{ j != n} \end{cases}$$

Nuevamente, como aclaramos en la sección complejidad, nuestra recursión hace lo siguiente: Verifica conexiones de los pisos i con el último piso, si las hay, escribe 1 en [i][piso n] (pues 1 es el maximo de portales en llegar de i al último.). Si no hay conexión, coloca un -2 pues vamos a ignorar esa celda.

Luego, desde abajo a la derecha hacia la izquierda y arriba, chequea si los pisos i se conectan con j. Si lo hacen y a su vez j se conecta con el piso n, toma el máximo entre mejores[i] y mejores[j+1].

Esto chequea, para cada piso, cuántos portales tarda en llegar hasta el último, y si hay forma de llegar pasando por los pisos superiores (que ya recorrimos). La comparacientre mejores[i] y mejores[j+1] es porque comparamos el máximo valor desde el piso i hasta n contra el máximo valor en el piso j + la conexion de i con j.

En cada iteración vamos a conseguir la cantidad máxima de portales para llegar desde el piso i hasta el n. Comenzamos con i en el anteúltimo piso y vamos bajando; esto nos garantiza que al final del recorrido vamos a estar en el piso 0, por ende tendremos la cantidad máxima de portales necesaria para llegar desde el piso 0 hasta el n, que es lo que buscábamos.

En la función, la variable H va a estar dada por el resultado de la función iterativa max. Si el resultado es -2(es decir, no hay max pues no hay conexion de i con k) entonces H es 0, caso contrario es 1 (incrementamos en 1 al maximo que es distinto a -2, pues le agregamos al maximo que habiamos encontrado una conexion extra de i a k).

Una vez entendida nuestra función recursiva veamos gráficamente cómo funciona tomando la siguiente matriz:

	Portal					
	Al 0	Al 1	Al 2	Al 3	Al 4	Al 5
Piso 0		←	←	←	←	←
Piso 1			←	←	←	←
Piso 2				←	←	←
Piso 3					←	←
Piso 4						←
Piso 5						

Las celdas de color rojo no son recorridas por nuestro algoritmo ya que no hay portales que vayan a un piso inferior o al mismo piso. Sólo recorremos las que tienen flechas hacia la izquierda.

Una vez iniciada la matriz, tendrán -1 las celdas que representen una conexión entre pisos. Ejemplo, si hay un -1 en [Piso 3][Al 4] significa que hay un portal del piso 3 al 4. Si hay un -2 en esta misma celda, significará que estos pisos no están conectados.

Ahora recorremos la columna de Al5 reemplazando los -1 con 1. Esto deja momentáneamente un 1 como cantidad máxima de portales desde los pisos correspondientes hasta el piso n. Además guardamos en mejores[i] ese valor. Ejemplo, [Piso 3][Al 5] era -1, lo cambiamos a 1 y mejores[3] ahora es 5. Caso contrario quedan en -2 ambos, celda de la matriz e índice del vector.

Luego recorremos el resto de la matriz;
comenzando desde "abajo a la derecha", [Piso3][Al4], verificamos si hay conexión. Si no la hay, no hacemos nada y nos movemos a la izquierda, de no tener una celda con flecha como es el caso, vamos hacia arriba, [Piso2][Al4]. Esto mantiene nuestro "invariante"de la estructura, con el que garantizamos que siempre tenemos el máximo hasta el lugar recorrido, ya que el que no haya conexión significa que es una celda que no nos interesa.

Ahora, si hay conexión en [Piso3][Al4] y además mejores[4] es mayor a 0 (condición importante porque significa que hay forma de llegar desde 4 a 5, caso contrario no nos interesa la conexión [Piso3][Al4]), lo que hacemos es tomar [Piso3][Al4] y asignarle Mejores[4] + 1 (ya que la cantidad de portales para ir del 3 al 5 es igual a la catidad de portales para ir del 4 al 5 mas el portal que nos lleva del 3 al 4) , y luego a mejores[3] el máximo entre el valor viejo de mejores[3] y el nuevo valor de [Piso3][Al4].

En este paso tomamos una celda que conectaba a un piso i con uno j y le asignamos la máxima cantidad de portales para llegar al último piso si era posible, guardando además en mejores[i] esa máxima cantidad.
Una vez realizado esto, nos seguimos moviendo por la matriz. Como empezamos desde el último piso y vamos bajando, garantizando que en cada iteración guardamos el máximo posible en las celdas y en mejores[i] para todo piso i.

Entonces, finalmente, cuando terminamos de recorrer el Piso0 tendremos en mejores[0] la cantidad máxima de portales.

Como no asumimos nada de la matriz antes de aplicarle el algoritmo y analizamos todos los posibles "If's", esta correctitud vale para toda matriz.

1.4 Tests

Asumimos que no habría mejores y peores casos, ya que no importa cómo se distribuyan los portales, si o si nuestro algoritmo recorría toda la matriz (la diagonal superior).

Como no utilizamos ningún tipo de poda, (como por ejemplo no chequear los pisos que no se conectan con los siguientes) en los tests se verá reflejado que no existe ningún mejor ni peor caso. Esto sucede por un motivo en particular: si bien la parte inferior de la matriz no la tocamos en ningun momento, sin importar el caso en el que estemos, siempre vamos a estar completando y revisando toda la matriz del lado superior, por ende la unica diferencia que hay entre dos casos cualquiera es el resultado de la operación elemental y no la cantidad que estemos haciendo; es por eso que aunque llenemos nuestra matriz con portales o la dejemos vacía, el tiempo de computo no se verá drásticamente afectado.

1.4.1 Performance

A la hora de hacer nuestras mediciones, creíamos que si bien nuestra funcíon tiene una complejidad de $\mathcal{O}(n^2)$, en realidad iba a ser mucho menor ya que las operaciones que realizábamos eran sobre la mitad de la matriz.

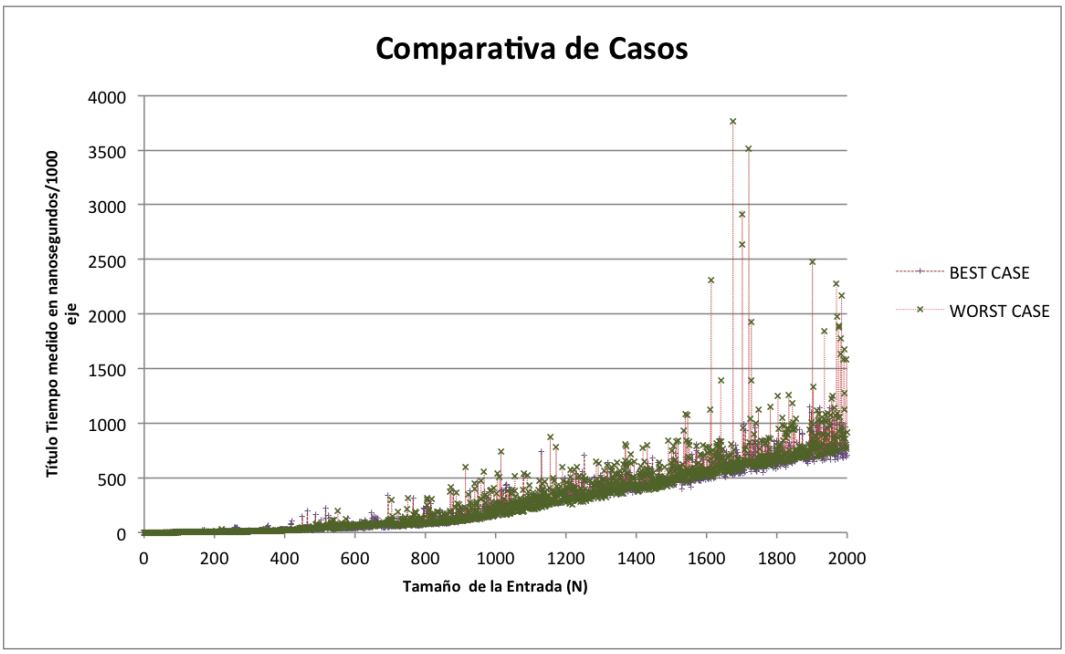


Figura 1: Comparación de matrices sin portales y completas.

Efectivamente sucede lo que sospechabamos en la seccion anterior: En la primera figura vemos como el mejor y el peor caso se comportan asintoticamente

de la misma manera ya que no importa qué portales tengamos, la parte superior de la matriz se completa siempre.

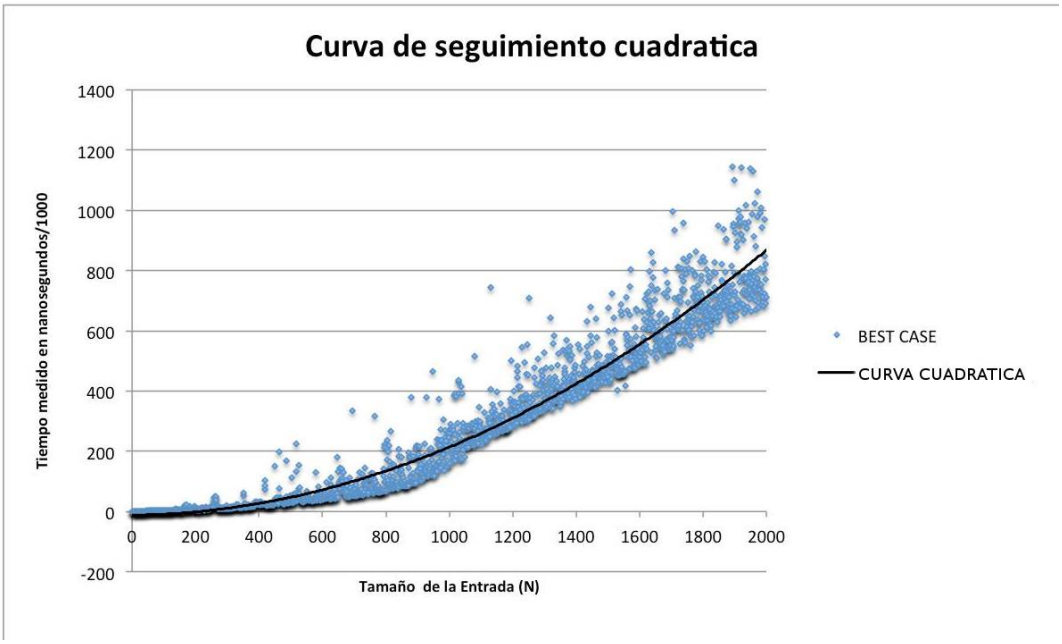


Figura 2: Comparación de matrices sin portales y funcion cuadratica.

En este segundo gráfico podemos ver que el comportamiento del algoritmo se puede aproximar con una curva cuadrática multiplicada por una constante baja. Esto sucede porque como habíamos mencionado anteriormente, nunca se recorre la matriz entera.

2 Ejercicio 2

2.1 Problema: Algo Rush

El Pabellón 0+infinito acaba de reabrir sus puertas con la novedad de que ahora tiene P portales que son bidireccionales; asimismo, los paracaídas fueron eliminados por considerarse inseguros. Cada piso del renovado pabellón consta de un pasillo de L metros de longitud, y cada portal permite viajar entre posiciones específicas de los pasillos de dos pisos. Más concretamente, cada portal puede describirse por medio de cuatro enteros no negativos A, DA, B y DB, los cuales indican que el portal comunica el piso A, a DA metros del comienzo del pasillo de ese piso, con el piso B, a DB metros del comienzo del pasillo de ese piso. Los alumnos, acostrumbrados a los portales que sólo permitan subir, están un poco confundidos al poder utilizar un mismo portal tanto para subir como para bajar entre dos pisos, o incluso para moverse entre posiciones diferentes dentro del pasillo de un mismo piso. Todos los alumnos de Algoritmos III quieren llegar primero a la clase, que es en un aula que está al final del piso N (el más alto del pabellón). Diseñar un algoritmo de complejidad O(NL + P) para calcular la mínima cantidad de segundos que se necesitan para llegar del comienzo del pasillo del piso 0 al final del pasillo del piso N, suponiendo que recorrer un metro requiere 1 segundo, y utilizar cualquier portal requiere 2 segundos (en cualquiera de las dos direcciones posibles). Se asegura que en toda instancia del problema es posible realizar el recorrido deseado, y que no hay más de un portal que comunique las mismas posiciones del mismo par de pisos. No obstante, puede haber más de un portal que comunique el mismo par de pisos, y portales que comuniquen posiciones diferentes dentro del pasillo de un mismo piso.

Este problema esta planeado de una manera parecida al anterior tendiendo como diferencia principal que ahora los portales son bidireccionales y que el primero quiere maximizar la cantidad de portales usados y este desea minimizar el tiempo para llegar desde el principio hasta el final. La medida del tiempo, com bien dice el enunciado, depende de la cantidad de portales que utilicemos y de la cantidad de metros que caminemos. Cada portal que utilicemos nos tomara 2 segundos y cada metro que camienmos nos aumentara 1 segundo.

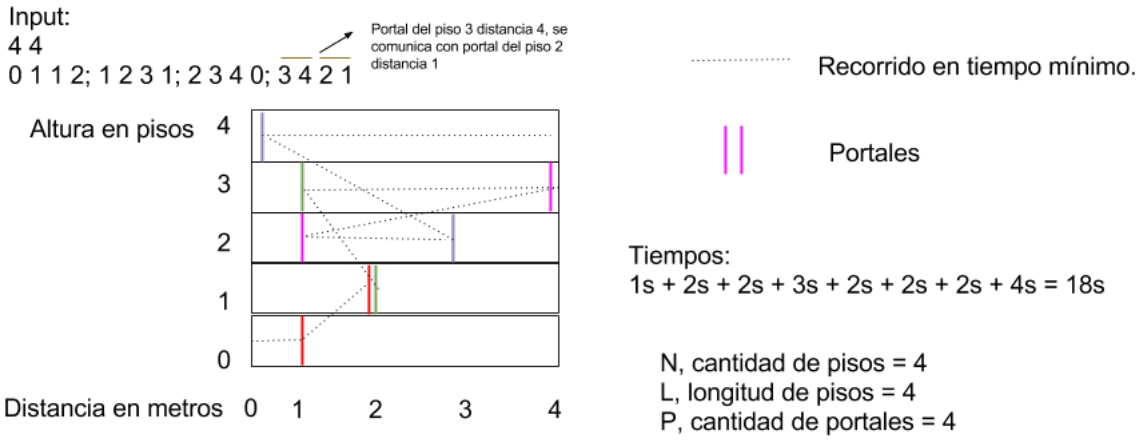


Figura 3: representacion de pisos y portales y su solucion

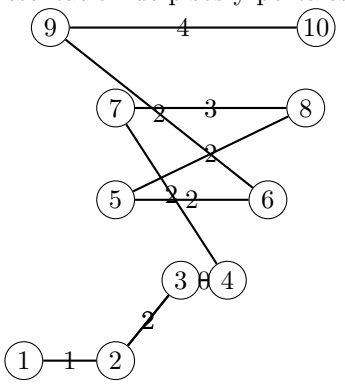


Figura 4: representacion en un grafo del ejemplo anterior

2.1.1 Explicacion del Problema

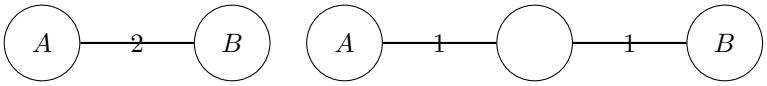
El problema consiste en minimizar la cantidad de tiempo que se tarda en llegar desde el inicio de la planta baja hasta el final del ultimo piso. Esto puede ser representado por un grafo donde los nodos representan los portales (el inicio de la planata baja y el final del ultimo piso tambien) y los ejes los metros (o segundos) entre un portal y otro, observar que para este caso los ejes son bidireccionales ya que se puede ir y volver por los portales. Si el eje representa la conexion entre dos portales que esten conectados su peso sera 2 y, de no estar conectados y ser adyacentes en el mismo piso, equivaldra a la distancia que exista entre ambos.

Para solucionar el problema se cuenta con una cota máxima de $O(N * L + P)$ con N cantidad de pisos, L longitud de cada piso y P cantidad total de portales.

2.1.2 Explicacion del Desarrollo

Se cuenta con un grafo con pesos distintos en sus ejes, sobre el mismo habra una cantidad total de $P+2$ nodos (cantidad de portales mas nodo inicial y final). A continuacion se analizara una modificacion del grafo, para esto definimos:

Sea G un grafo conexo con pesos posisitivos en sus ejes y v,w cualquier par de vertices adyacentes de G, llamamos G' al grafo donde el peso p en el eje que conecta v con w es reemplazado por $p-1$ nodos intermedios (de peso 1 en sus ejes) entre v y w, siendo asi G' un grafo con peso uniforme en sus ejes.



Ejemplo nodos fantasma para un eje de peso 2

El nuevo grafo G' colocara nodos intermedios entre los nodos de G, como puede haber un nodo en cada punta de los pisos (metro 0 y L) se pueden generar hasta $L-2$ nodos intermedios por piso, aplicandolo a todos los pisos queda $N*(L-2)$ nodos totales, falta tener en cuenta las conexiones entre portales a las cuales se les agrega un nodo a cada una con lo que quedaria $N*(L-2)+P$ con lo que puede ser acotado por $N*L+P$ nodos totales

Por lo visto en clase el algoritmo BSF tiene complejidad lineal sobre la cantidad de nodos del grafo y puede encontrar el camino minimo entre 2 nodos si los pesos son constantes en sus ejes, si trabajamos con G' podemos ver que cumple con las condiciones que pide BFS y su complejidad sera O(cantidad de nodos de G') = $O(N*L+P)$ que es la complejidad pedida.

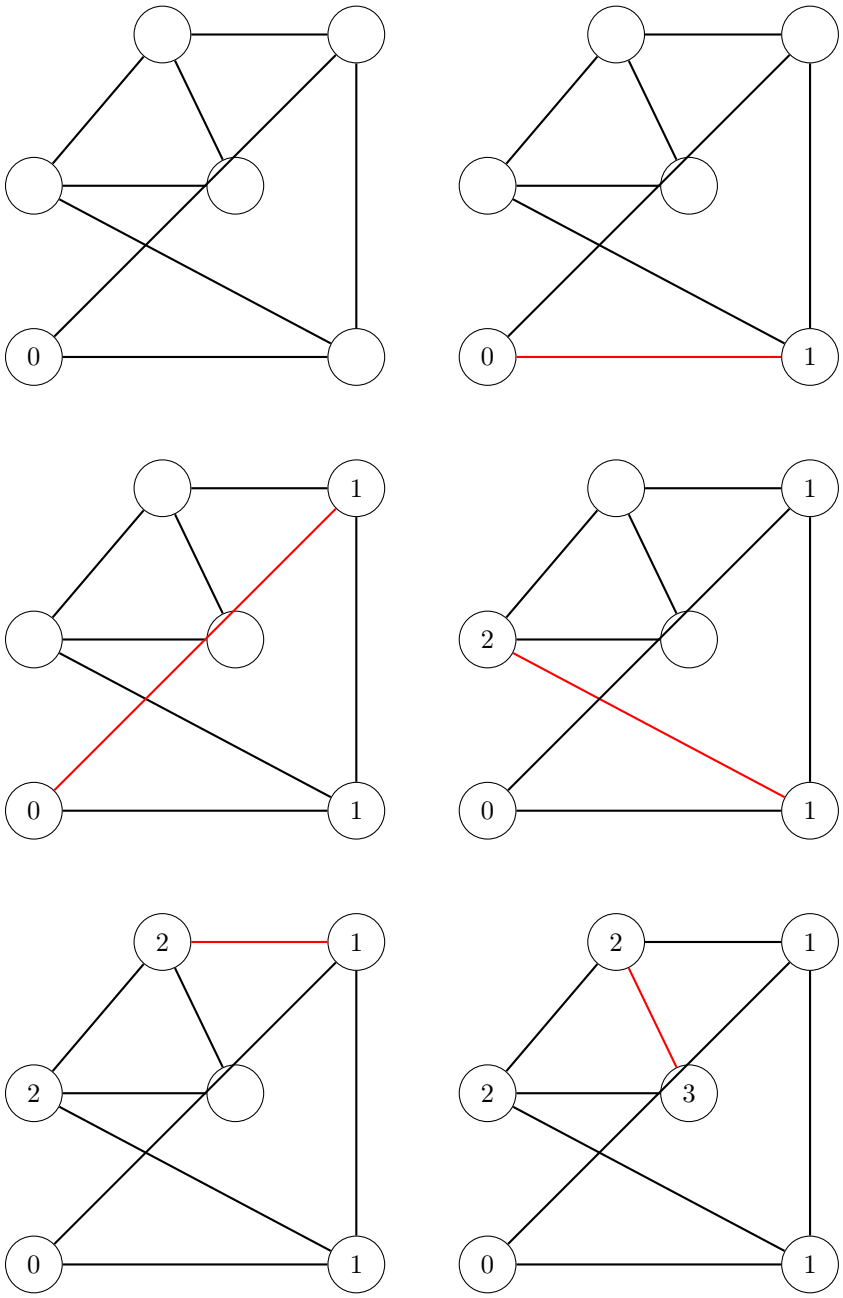


Figura 5: Algoritmo BFS

2.2 Justificación y Complejidad

La complejidad esta dada por: A) Transformar el grafo

```
Grafo2(List<Portal<Baldoza>> portales, int pisos, int
    mts) {
                this.mts = mts;//O(1)
                pisosUsados = new Boolean[pisos +
                    1];//O(pisos+1)
                FOR i desde 0 hasta pisosUsados.length
                        pisosUsados[i] = false;//O(1)
                ENDFOR//O(pisos + 1)
                nodos = new Nodo[pisos+1][mts+1];//O(1)
                nodosFantasmas = new
                    LinkedList<Nodo>();//O(1)
                idVertices = 0;//O(1)

                FOR portales AS portal
                        Baldoza b1 = (Baldoza)
                            portal.getDesde();//O(1)
                        Baldoza b2 = (Baldoza)
                            portal.getHasta();//O(1)
                        connect(b1,b2);//O(CONNECT(B1,B2))
                }//O(ITERACIONES*CONNECTBALDOZAS)
                ENDFORECH
END
```

Por ahora nuestra complejidad es de $(N + P * connect(Baldoza, Baldoza))$, por lo tanto debemos analizar la función connect para poder determinar nuestra complejidad.

Veamosla a continuación:

```
connect(Baldoza b1, Baldoza b2)
        checkFloor(b1);//O(CHECKFLOOR(B1))
        checkFloor(b2);//O(CHECKFLOOR(B2))

        addNodo("FANTASMA", nodoFantasma)
        connectNodos(b1.getPiso() + "," + b1.getMetros()
            , "FANTASMA,"+ nodoFantasma)
        connectNodos("FANTASMA,"+ nodoFantasma ,
            b2.getPiso() + "," + b2.getMetros())
        nodoFantasma++
END
```

Por lo tanto podemos acotar esta función por $(N + 2 * checkfloor() + 2 * connectNodos())$ en la cual nos falta determinar la complejidad de las 2 funciones. Nuestra complejidad nos quedaria : $(N + N * (2 * checkfloor() + 2 * connectNodos()))$

Ahora veamos una de las dos funciónes que nos quedan, connectNodos, para la cual vamos a necesitar el analisis conjunto con la función addVecino la cual nos provee nuestra clase NODO, que lo que permite es agregar un nuevo vecino:

```
connectNodos(String string, String string2)
        Nodo a = getNodo(string);//O(1)
        Nodo b = getNodo(string2);//O(1)
        IF ! a.getVecinos().contains(b) //O(P)
           a.addVecino(b);//O(ADDVECINO(A))
        ENDIF
        IF ! b.getVecinos().contains(a)//O(P)
            b.addVecino(a);//O(ADDVECINO(B))
        ENDIF
END

addVecino(Nodo vecino)
        vecinos.add(vecino);    //O(1)
```

Podemos acotar la función connectNodos en (P), ya que todas las operaciónes cuestan (1), excepto el contains , y esto surge de una buena elección de las estructuras.

En este estado del analisis podemos afirmar que nuestra complejidad temporal es de : $(N + N * (2 * checkfloor()))$

```
checkFloor(Baldoza b2){
        IF !pisosUsados[b2.getPiso()]
            pisosUsados[b2.getPiso()] = true;//O(1)
                FOR j desde 0 hasta mts + 1
                        addNodo(b2.getPiso(), j);  //O(1)
                ENDFOR
                FOR j desde 0 hasta mts
                        int k = j + 1;//O(1)
                        connect(b2.getPiso()+","+j,
                           b2.getPiso()+","+k); //O(1)
                ENDFOR
        ENDIF
END
```

Esta función es la encargada de generar todos los portales fantasmas por cada piso, una vez que se ingresa un nuevo portal, si y solo si no ingreso un portal del mismo piso anteriormente. Por lo tanto vamos a ver la cota cuando este piso no es creado es de (1). Podemos que ver que son todas operaciónes que cuestan (1), y vemos que hay dos For's que iteran para generar los nodos, estos hacen L+1 y L iteraciónes, por lo tanto la complejidad es : $(L + 1 + L)$, lo que nos queda, (L). Para llegar a esto debimos ver que la complejidad de addNodo era (1), la cual se puede extraer del siguiente analisis:

Para esta función hicimos un uso de una funcionalidad de java, que es la sobrecarga de metodos, la cual permite tener varios metodos con el mismo nombre, pero se utiliza la correspondiente a la aridad de los parametros. Por lo tanto tenemos 2 metodos que se llaman "addNodo".

```
addNodo(int piso, int mts)
    IF nodos[piso][mts] == null       // O(1)
        nodos[piso][mts] = new Nodo(idVertices) // O(1)
                idVertices++;
                    // O(1)
                nodos[piso][mts].setCoordenada(piso+","+mts);
                    // O(1)
    ENDIF
DEVOLVER  nodos[piso][mts]    // O(1)

addNodo(String string, int nodoFantasma)
                nodosFantasmas.add(nodoFantasma,new
                    Nodo(idVertices)) // O(1)
                idVertices++  // O(1)
                nodosFantasmas.get(nodoFantasma).setCoordenada(
                "FANTASMA,"+nodoFantasma) // O(1)
DEVOLVER nodosFantasmas.get(nodoFantasma) // O(1)
```

Por lo tanto la complejidad de la sección de creación del grafo tiene una complejidad de $(L * P)$, al parecer esto no cumple con la complejidad pedida de $(N * L + P)$, pero analicemos esta complejidad y veamos que en realidad si estamos cumpliendo.

Nosotros Tomamos una cota de $(L * P)$, pero la misma surge de no tener en cuenta el caso (1) que va a aparecer en checkfloor. Segun esta función solo vamos a realizar operaciones de costo (L) si y solo si el piso no fue creado, y la cantidad de piso creados es N, por lo tanto podemos adjustar la complejidad de creación del grafo por $(N * L)$.

B) Resolver el problema Esta sección se basa en la utilización se basa en el algoritmo para recorrer grafos, conocida como BFS, el algoritmo tiene complejidad $(n + m)$, siendo n los vertices y m las aristas de un grafo.Veamos como se relaciona con la complejidad que nos piden.

El pseudo-codigo:

```
solve(Nodo nodo, Nodo nodo2, int i)
        cola <- new LinkedList<Nodo>()
        cola.addFirst(nodo)
        nodo.setVisitado()
        WHILE ! cola.isEmpty()
        Nodo actual;
        actual = cola.pop()
        List<Nodo> vecinos = actual.getVecinos();
        FOREACH vecinos AS vecino
            IF !vecino.getVisitado()
                vecino.setVisitado()
                vecino.setLongitud(actual.getLongitud()+1)
                        cola.push(vecino)
                ENDIF
        ENDFOR
    ENDWHILE
DEVOLVER nodo2.getLongitud()
```

Podemos tener una cota de la cantidad de vertices y aristas y esta esta dada en gran medida por todos los nodos fantastas que creamos para resolver el problema. Como ya sabemos, el algoritmo generera todos los nodos fantasmas al ingregar un nuevo piso, por lo tanto la cota de nodos por piso es N * L , y la cantidad que se agregan para simular los portales es P. Por lo tanto tenemos (N * L) + P vertices.

Como el grafo es conexo sabemos que la relación entre vertices y aristas es al menos n = m - 1 , por lo tanto sabemos que m es al menos (N * L)+ P - 1, pero como estamos en complejidad y ya sabemos que m es una cota de n, podemos concluir que la complejidad de la resolución es de $((N * L) + P)$

Ahora solo nos queda sumar las dos complejidades, lo que nos quedaría de la siguiente manera : $(N + (N * L) + P) + (N * L)$. Por lo tanto podemos concluir que la complejidad cumple lo pedido y es de : $((N * L) + P)$.

2.3 Correctitud

El problema consiste en encontrar el camino de menor peso dentro del grafo entre dos nodos, Usando G' podemos decir que la cantidad de nodos del camino mas corto en G' sera igual al peso del camino minimo de G por $Lema2.1$.

Para Encontrar el camino usamos BFS, por lo visto en clase este nos dara el camino minimo en G'

2.3.1 Lema 2.1

sea G un grafo conexo con pesos positivos en sus ejes y G' un grafo con los mismos vertices que G talque el peso de sus ejes es reemplazado por $p-1$ nodos intermedios cuyos ejes tienen peso 1. Sean v,w vertices de G y c,c' un camino minimo entre v y w en G y G' respectivamente $\Rightarrow$ el peso total de c es igual al de c'.

Demostracion

Primero veamos que pinta tienen los caminos creados en G', sean v y $w \in V(G)$ dos nodos adyacentes y p el peso del eje que pasa entre ellos, por como esta definido G' se crean $p-1$ nodos intermedios con peso 1 en sus ejes $\Rightarrow$ existe un unico camino en G' para todo par v w y tiene la forma $v, v_1, v_2, ..., w$, ademas si llamamos a ese camino c su peso sera igual al peso del eje que conecta v y w en G (para esto basta ver que el peso del eje en G es igual a la cantidad de ejes de c en G' por como definimos los nodos fantasma).

Ahora sean $v', w' \in V(G')$ y c_{min} el camino minimo entre v' y w', por como esta formulado el problema no nos interesara el caso en que alguno de los dos nodos no pertenezca a G. Si ambos nodos pertencen a $G \Rightarrow$ pertenecen a G' y su camino sera de la forma $v', v_1, v_2, ..., w'$. Si en el medio del camino se cruzaron con otro nodos de G entonces el camino sera de la forma $v', v_1, v_2, ..., v'_j, ..., w'$, este camino se puede expresar como una union disjunta de ejes de la forma $v', v_1, v_2, ..., v'_1 \sqcup v'_1, ..., w'$ donde $peso(c_{min}) = peso(c_{min_1}) + peso(c_{min_2})$. Podemos observar que siempre que exista algun nodo intermedio que pertenezca a G podemos dividir nuevamente en uniones disjuntas, asi hasta quedarnos con caminos donde los unicos nodos que pertenecen a G son los de las puntas.

Ahora sean $v, w \in V(G' \cap G)$ veamos que si c es camino minimo en $G' \Rightarrow$ $peso(c) = peso(c')$ con c' camino minimo en G. Supongamos que no es camino minimo en $G \Rightarrow$ existe c'_2 que es menor a c' entonces puedo crear un camino $c_2 \in G'$ que contenga a los nodos de c'_2 y que por lo visto arriba su peso tendra la forma $peso(c_2) = peso(c'_{2_1}) + peso(c'_{2_2}) + peso(c'_{2_3})....peso(c'_{2_k})$ con k cantidad de nodos distintos del camino $c'_2 - 1$. Por hipotesis sabemos que $peso(c) = peso(c'_1) + peso(c'_2) + ... + peso(c'_n)$ con n cantidad de nodos distintos del camino $c' - 1$.
Como c'_2 es menor que $c' \Rightarrow \exists$ j tal que $peso(c'_{2_j}) < peso(c'_j) \Rightarrow c_2$ es menor que c. absurdo ya que c era camino minimo en G'.

2.4 Tests

Para analizar los casos de este ejercicio se tomaron en cuenta los mejores y peores casos en los que puede correr.

2.4.1 Mejor Caso

En el mejor de los casos existira un portal en el metro 1 del primer piso que este conectado con el metro $L-1$ del ultimo piso, con lo cual solo se necesitarian dos pasos del BFS para llegar al ultimo paso, siendo constante el tiempo de ejecucion.

2.4.2 Peor Caso

En el peor caso como BFS recorre armando un arbol de izquierda a derecha, si el unico portal que comunica con el ultimo piso se encuentra en el anteultimo piso en el metro $L-1$ y comunica con el metro 0 del ultimo piso entonces BFS tendra que recorrer todo el grafo obligatoriamente, llevandole $N * L + P$ pasos.

2.4.3 Caso General

Para el caso general se distribuyeron portales y se los conecto de manera pseudo-aleatoria, eligiendo dos pisos y colocando dos portales conectados entre ellos. Con lo que la complejidad quedara acotada por lo analizado previamente. En el gráfico de comparación de tiempos podemos ver que el caso promedio se asemeja al mejor caso y se mantienen muy por debajo del peor caso.

2.5 Performance

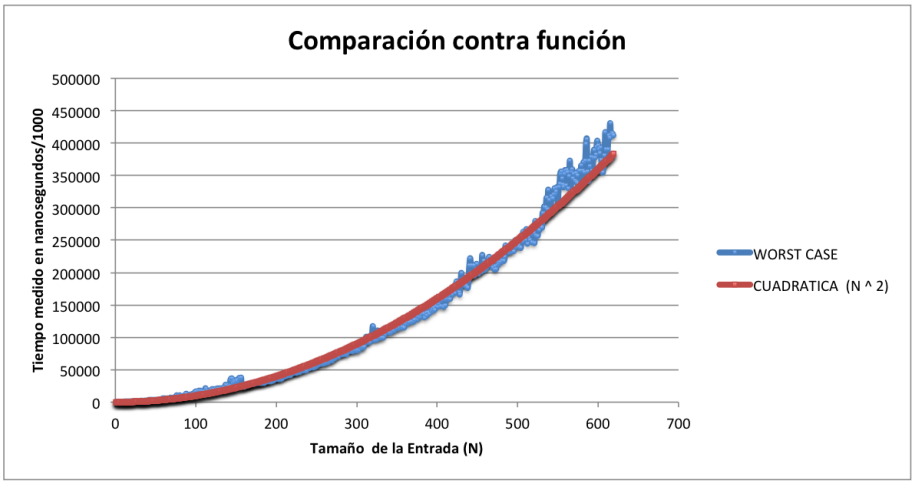


Figura 6: Peor caso en comparación con una función cuadrática.

Como mencionamos antes, el peor caso sucede cuando debemos recorrer todos los nodos. La cantidad de nodos viene dada por $N * L$. Para hacer los tests, decidimos utilizar edificios de forma cuadrada, es decir que la cantidad de

pisos es igual a la longitud de cada piso. Para estos casos, tenemos que $L = N$. Y como se ve en el grafico, el peor caso se comporta muy similar a esta manera.

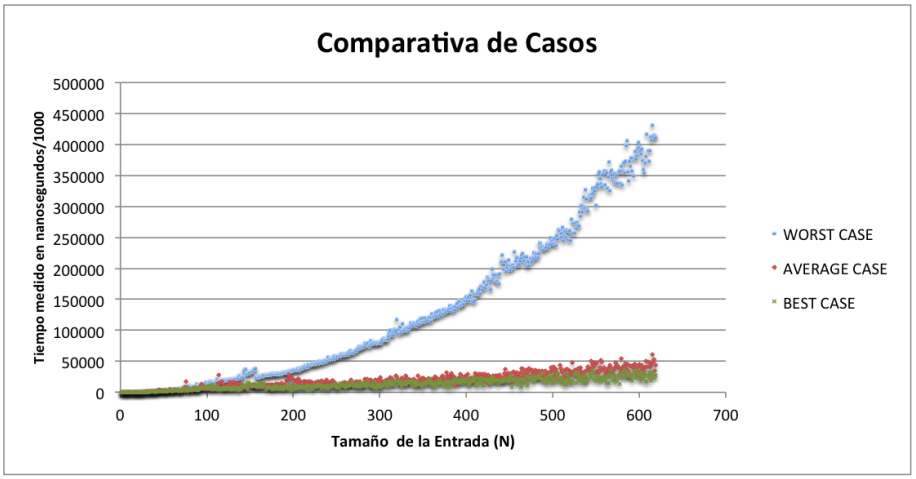


Figura 7: Comparación de los peores y mejores casos contra casos promedio.

Como podemos apreciar en el grafico, el caso promedio se aleja muchisimo del peor caso ya que al general de manera pseudo-aleatoria las conexiones entre los portales, la probabilidad de tener que recorrerlos absolutamente todos ($N * P + L$) es muy baja. Llamativamente, el caso promedio se asemeja al mejor caso. Creemos que la causa es que como un portal puede tener muchos vecinos, podemos encontrar el camino al final de manera mas rapida.

2.5.1 Optimizado vs no optimizado

Para mejorar el rendimiento del algoritmo se agrego una optimizacion donde al momento de encontrar el nodo buscado finalizara, a continuacion se haran algunas pruebas sobre el beneficio que brinda dicha optimizacion en los casos vistos en la sección 2.4.
Primero veamos un poco el algoritmo:

```
solve(Nodo nodo, Nodo nodo2, int i)
        cola <- new LinkedList<Nodo>()
        cola.addFirst(nodo)
        nodo.setVisitado()
        WHILE ! cola.isEmpty()
        Nodo actual;
        actual = cola.pop()
        List<Nodo> vecinos = actual.getVecinos();
        FOREACH vecinos AS vecino
            IF !vecino.getVisitado()
                vecino.setVisitado()
                vecino.setLongitud(actual.getLongitud()+1)
                        cola.push(vecino)
                ENDIF
        ENDFOR
```

```
    ENDWHILE
DEVOLVER nodo2.getLongitud()
```

Por como esta dado simplemente chequeara todos los nodos y, al terminar, quedara en *nodo*2 la longitud minima, a continuacion se considerara el siguiente codigo:

```
solve(Nodo nodo, Nodo nodo2, int i)
        cola <- new LinkedList<Nodo>()
        cola.addFirst(nodo)
        nodo.setVisitado()
        encontrado <- false
        WHILE !cola.isEmpty() AND !encontrado
        Nodo actual;
        actual = cola.pop()
        List<Nodo> vecinos = actual.getVecinos();
        FOR i<-0 WHILE i<vecinos.size() AND !encontrado
            STEP i++
             vecino <- vecinos.obtener(i)
             IF !vecino.getVisitado()
                 vecino.setVisitado()
                 vecino.setLongitud(actual.getLongitud()+1)
                 IF actual.getId() == nodo2.getId()
                     encontrado <- true
                 ENDIF
             cola.push(vecino)
             ENDIF
        ENDFOR
    ENDWHILE
DEVOLVER nodo2.getLongitud()
```

La variable encontrado sera colocada en true al momento de toparse con el nodo que era buscado, a primera vista esto deberia mejorar la performance del algoritmo, para ello se hara un analisis de cada caso:

Peor Caso

En la imagen se puede observar que no hay una gran diferencia entre los dos algoritmos, esto se debe a que en el peor caso solo habra una conexion con el ultimo piso y esta estara al final del recorrido, por lo que cualquiera de los dos algoritmos debera recorrer todo el grafo hasta llegar al nodo buscado. Incluso podemos decir que el algoritmo optimizado puede tardar un poco ya que posee 3 chequeos intermedios sobre la variable *encontrado*, que si bien no es un gran cambio afecta un poco.

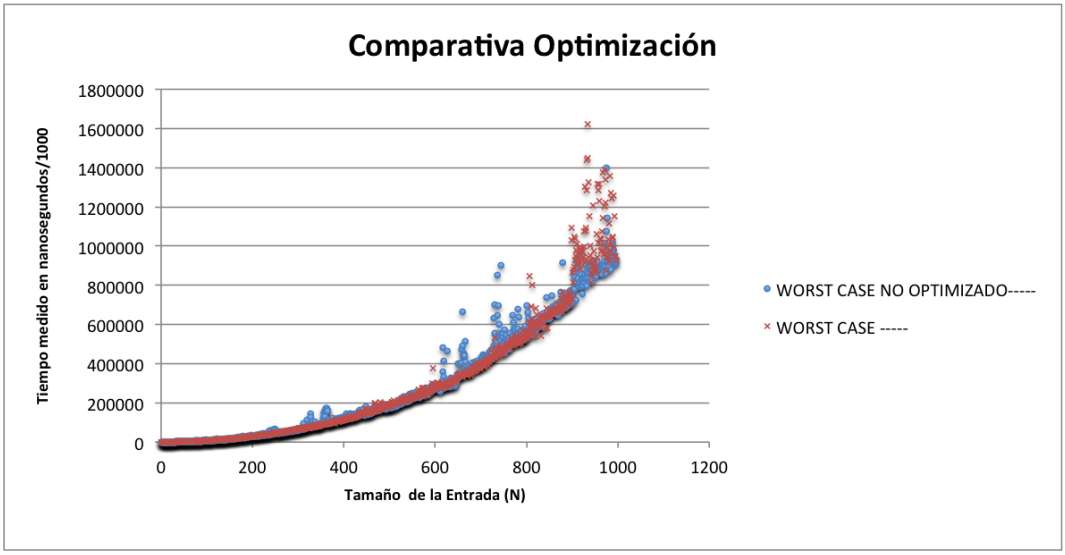


Figura 8: Comparacion peores casos

Mejor Caso

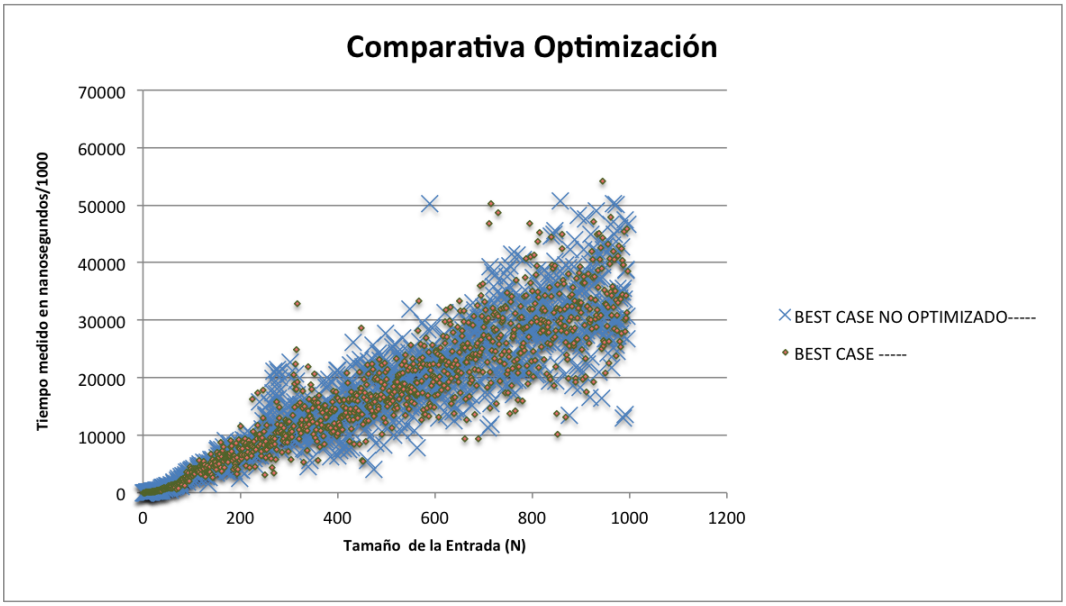


Figura 9: Comparacion mejores casos

En el mejor caso se construyo el grafo de manera que hubiera un portal apenas comienza el piso 0 que conecte con el final del ultimo piso. Se espero que el rendimiento del algoritmo optimizado fuera superior al otro pero en el grafico se puede observar que no.

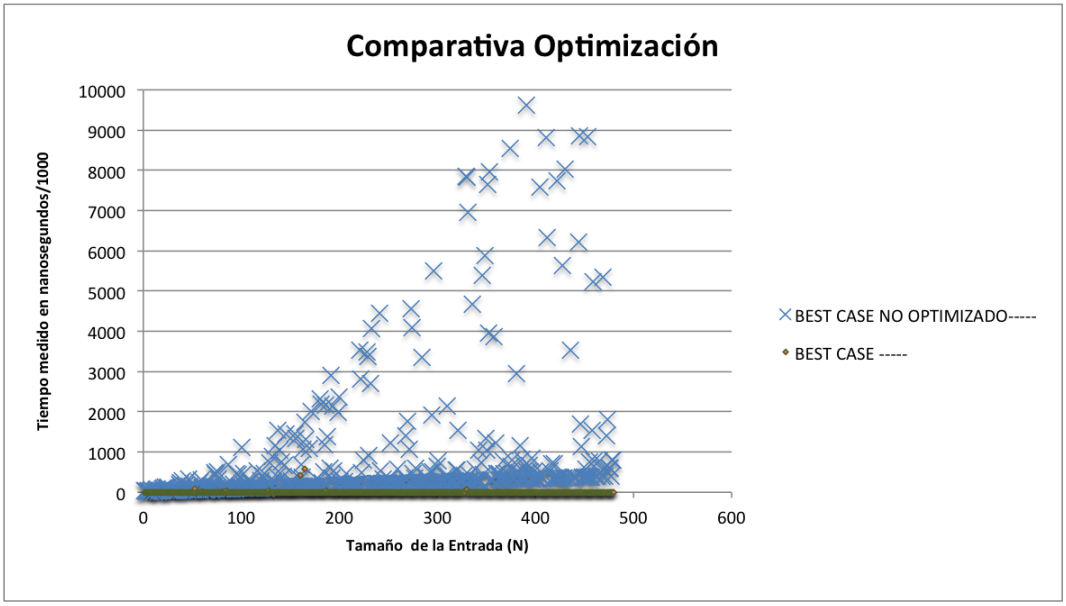


Figura 10: Comparacion cantidad de iteraciones mejores casos

Haciendo mas pruebas se pudo ver que el algoritmo optimizado casi siempre hizo 4 iteraciones (ya que al tener portales lanzados aleatoriamente puede haberlo enviado a otro lugar) antes de terminar mientras que el otro tuvo cantidad variable y creciente, lo que contradice el primer grafico. La hipotesis es que hay algun costo externo que no pudo ser observado en la experimentacion. Pero teoricamente y por la prueba de iteraciones maximas podemos decir que el algoritmo optimizado para este caso es mucho mas efectivo que el otro, teniendo un costo de 4 iteraciones constantemente.

Caso General

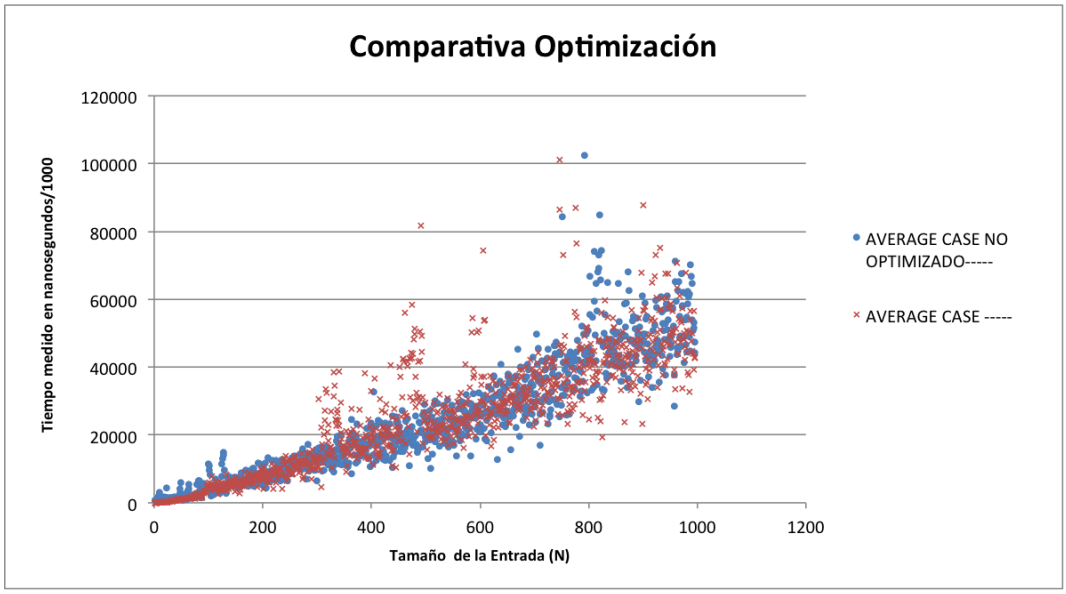


Figura 11: Comparacion casos generales

En el caso general el algoritmo optimizado tuvo un poco mas de rendimiento que el otro, se puede observar en el grafico que su curva tiene una pendiente menos eleveda que el no optimizado, como en el caso general no sabemos cual es el camino general, el primer algoritmo tiende a ser un poco mejor.

Luego decidimos comparar el mejor y peor caso del algoritmo, y analizamos los tiempos teniendo en cuenta la creación del grafo o no. En la siguiente imagen se observa la gran diferencia de tiempos y esto es debido al gran costo de incialización de estructuras.

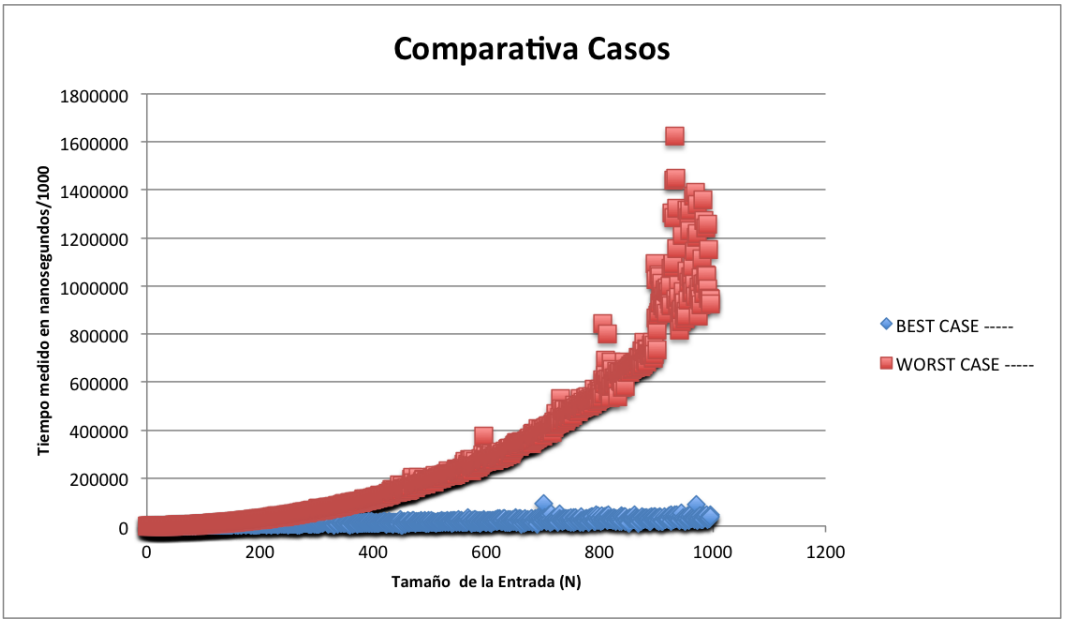


Figura 12: Comparacion Best Vs Worst

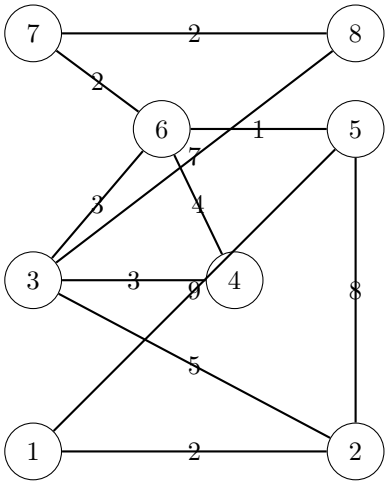
3 Ejercicio 3

3.1 Problema: Perdidos en los Pasillos

El Pabellón 0+infinito, nuevamente remodelado, ahora tiene un diseñõ basado en un conjunto de M pasillos de distintas longitudes con intersecciones en donde se unen dos o más pasillos. Es así que puede modelarse como un grafo con pesos en los ejes, donde cada eje es un pasillo (de peso igual a la longitud del pasillo), y cada vértices es una intersección o un extremo donde termina un pasillo sin unirse con ninguń otro. El decano junto con el director del Departamento de Computación, están preocupados porque talvez existen ciclos en dicho grafo, lo que podría perjudicar a los alumnos al hacer que se pierdan buscando las aulas. Por tal motivo el decano decidió clausurar los pasillos que sea necesario de manera tal que no queden ciclos en el grafo que representa al pabellón. El problema es que cuanto más largo es un pasillo, más costoso es clausurarlo. Disenãr un algoritmo de complejidad O(M log M) para calcular la mínima suma posible de las longitudes de los pasillos que deberían ser clausurados (eventualmente ninguno) para que no existan ciclos formados por tres o más pasillos en el grafo que representa al pabellón. Se asegura que en toda instancia del problema el grafo que representa al pabellón es conexo.

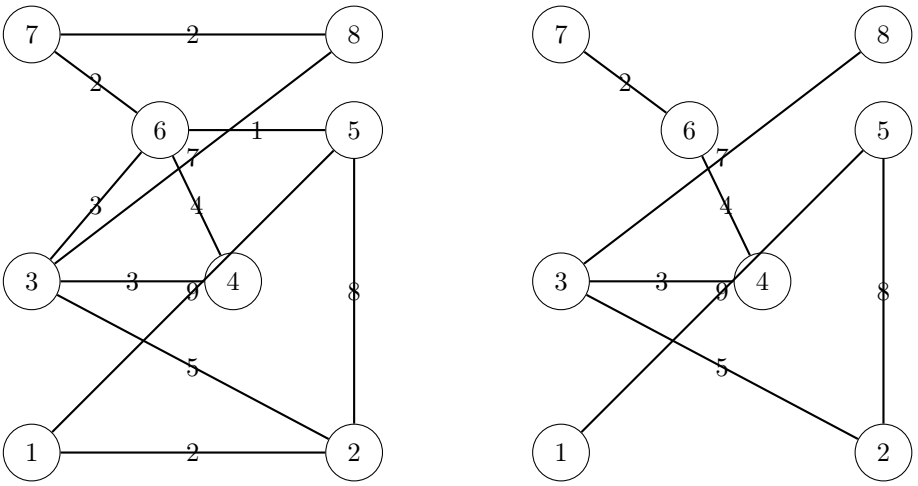
3.1.1 Explicacion del Problema

En este problema se tiene un edificio con pisos de longitud variable y portales que comunican pisos entre si. Por enunciado, este se puede representar como un grafo con pesos en sus ejes.



Ejemplo de grafo que representa el problema

Lo que se quiere es eliminar (en el que caso de hubiera un ciclo) la menor cantidad de metros de pasillo posible (minima cantidad de peso total del ciclo) de manera tal que el grafo siga siendo conexo.



Grafo donde fueron eliminados los ejes menos pesados de los ciclos

En el ejemplo se tomaron los pasillos de menor tamaÃ±*odandodebajatansolo*6*metros*

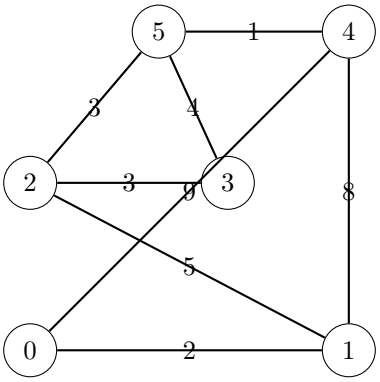
3.1.2 Explicación del Desarrollo

Para resolver el problema se planteo eliminar el eje de menor peso en cada ciclo para asi eliminar la menor cantidad de metros posible de manera que el grafo siga siendo conexo.

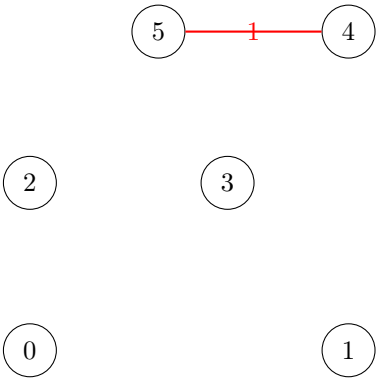
Analizandolo podemos ver que el resultado de eliminar estos ejes resulta en un arbol, el cual por haber eliminado los ejes de menor peso resulta ser el arbol generador maximo. Dicho esto el problema queda reducido a generar el arbol generador maximo del grafo, recordemos que la complejidad debe quedar acotada por $O(M * logM)$. Para generar el arbol se recurrio al algoritmo de kruscal, si bien este genera el arbol generador minimo, si se multiplican todos los pesos por -1 el arbol generador minimo sera el maximo del grafo (esto se analizara en detalle en la seccion 3.3).

Si bien kruscal genera el el AGM que se busca no cumple con la complejidad pedida ya que buscar el eje de menor peso y agregarlo a los revizados no tiene (por lo pronto) tiempo constante, para esto se recurrio a $UnionFind$ con las huristicas $pathcompression$ y $link - by - rank$ (Ver seccion 3.3).

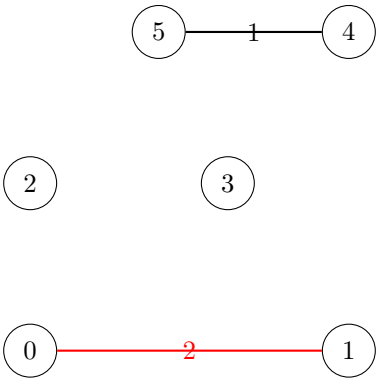
A continuación vemos el funcionamiento del Algoritmo de Kruskal.



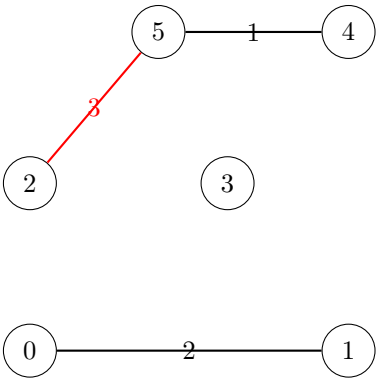
Grafo original



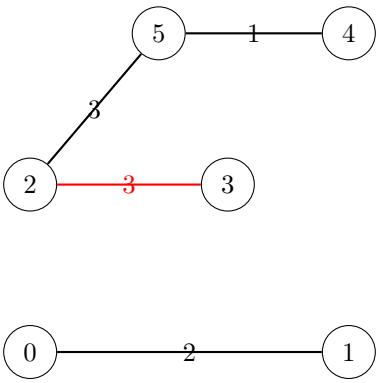
Agrega 1



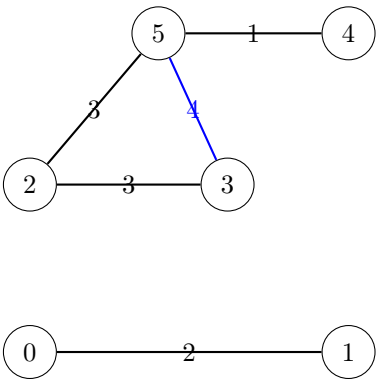
Agrega 2



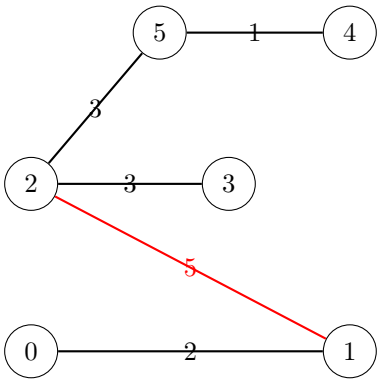
Agrega 3



agrega 3



Al intentar agregar 4 se forma un circuito



Al agregar 5 se termina de formar el arbol

algoritmo de kruskal para arbol generador minimo salteando los pasos finales donde solo saltea ejes.

3.2 Justificación y Complejidad

```
grafo <- new Grafo();         // O(1)
FOREACH pasillos as pasillo  // O(M)
    grafo.addVertice(pasillo.getExtremo1(),
       pasillo.getExtremo2(),
    pasillo.getLongitud());
ENDFOREACH
union <- new UnionFind(grafo.getVertices().size() + 1);
   // O(M)
```

Veamos el costo de la funcion addVertice con mas detalle

```
addVertice(int nodo1, int nodo2, int peso )
        vertice <- new Vertice(nodo1, nodo2, -peso); //
           O(1)
        vertice.setId(idVertices); // O(1)
        idVertices++;   // O(1)
        vertices.add(vertice); // O(1)
        this.peso += peso; // O(1)
        }
```

Costo final de la función addVertice $\mathcal{O}(1)$, por lo tanto el costo de la creación del grafo es $\mathcal{O}(M)$

```
vertices <- grafo.getSortedVertices(); // O(M * log(M))
i = 0;
peso = 0;
WHILE i < vertices.size()
        IF
            union.find((Integer)vertices.get(i).getNodo1())
            !=
            union.find((Integer)vertices.get(i).getNodo2())
        THEN
```

```
        union.union((Integer)vertices.get(i).getNodo1(),
        (Integer) vertices.get(i).getNodo2());
        peso += vertices.get(i).getPeso();
        ENDIF
        i++;
                }
ENDWHILE
DEVOLVER -( ( grafo.getPeso()*-1 ) - peso;
```

3.3 Correctitud

Primero observemos que los pesos del grafo G fueron modificados por su peso multiplicado por -1 lo que da otro grafo G', esto hace que todos los pesos queden negativos. luego se corre el algoritmo de kruskal sobre ese grafo, por lo visto en clase este nos dara el arbol generador minimo de G' que por $Lema 3.1$ resulta ser el arbol generador maximo de G.

Ahora veamos que el arbol generador maximo de G resuelve el problema: Se necesita cerrar la menor cantidad de metros de pasillo tal que no queden ciclos entonces se debe sacar el eje de menor peso de cada ciclo, sea e el eje de meor peso de algun ciclo de G entonces el arbol generador maximo tomara primero los ejes mas grandes del algun ciclo, dado que e es el menor de los ejes de algun ciclo al momento en el que es tomado ya fueron agregados k ejes al arbol generador tal que $\forall k_i\ p(k_i) >= p(e)$. si agrego e al arbol se formara un ciclo ya que como e tiene el menor peso del mismo sera tomado ultimo. A continuaion como e genera un ciclo no es agregado al arbol $\rightarrow$ el eje que no es agregado al arbol generador maximo es el de peso minimo.

Con esto queda probado que para todo ciclo del grafo G los unicos ejes que seran removidos seran los de menor peso dentro de los ciclos.

3.3.1 Lema 3.1

Dado un grafo G con pesos en sus ejes y dado G' un grafo tal que $V(G) = V(G')$ y $\forall e \in E(G), e' \in E(G')\ p(e) * -1 = p(e') \rightarrow$ el arbol generador minimo de G es el arbol generador maximo de G'

Sea H el peso del AGM en G, y sea n la cantidad de sus vertices, como sabemos que es un arbol tiene $n-1$ ejes. Entonces sabemos que $H = \sum_{k=1}^{N-1} P(e_k)$.

Tambien sabemos que, al contener a todos los vertices, independientemente del valor de sus ejes, tambien es un arbol generador en el grafo g'. Sea T el peso de este arbol generador de g' Entonces $T = \sum_{k=1}^{N-1} P(e_k) * -1$

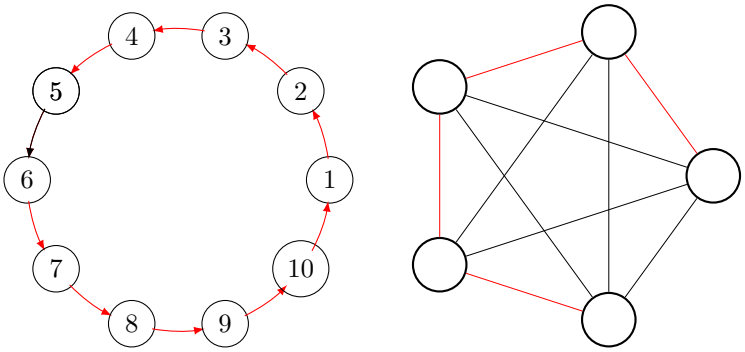
Y como la multiplicacion es distributiva, tenemos que $T = [\sum_{k=1}^{N-1} P(e_k)] * -1$

Entonces tenemos que $T = H * -1$. Ahora sabemos, que H era un AGM por ende $\forall a$ arbol, $H \leq$ Peso(a). Reemplazando, tenemos que $\forall a$ arbol, $T/-1 \leq Peso(a)$. Pero si paso el -1 para al pasar el -1 dividiendo, tenemos que dar vuelta la

desigualdad, quedando que: $\forall a$ arbol, $T \geq \text{Peso}(a)$. Osea Es Arbol Generador Maximo.

3.4 Tests

Para testear el mejor y peor caso decidimos comparar dos grafos que tengan la misma cantidad de aristas pero que difieran ampliamente en la cantidad de vertices. Describamoslo con un ejemplo: Tenemos por un lado el grafo $k5$, que tiene 10 aristas y por otro un ciclo simple de 11 vertices y 10 aristas. Lo que sucede es que el algoritmo de Kruskal corta al agregar la arista $n-1$(siendo n la cantidad de vertices) por ende en el segundo grafo, el algoritmo esta obligado a recorrer todas las aristas hasta conformar el AGM. Como vemos en la imagen, el algoritmo para el primer grafo, agregara todas las aristas que forman el pentagono y concluira sin haber revisado las aristas de la estrella.
Ademas pudimos observar que dependiendo de los pesos de las aristas de K_5 podiamos tener un mejor y un peor caso, en el que por ejemplo generemos el Arbol generador solo mirando solo las aristas que son el AG en si mismo, o podiamos estar en el peor caso que seria un ciclo.



3.4.1 Performance

En esta sección utilizaremos lo mencionado en los tests: Vamos a comparar que diferencia hay entre dos grafos que tienen la misma cantidad de aristas, difieran ampliamente en la cantidad de vertices y que ademas, las $n-1$ aristas mas pesadas, sean las que efectivamente conforman nuestro AGM. Con esto ultimo logramos que el algoritmo de Kruskal corte mucho antes de tener que recorrer todas las aristas del grafo. Esta idea la conceptualizamos en el siguiente gráfico, en el cuál se observa que la cantidad de aristas que revisa el algoritmo de Kruskal es mucho mayor para el grafo donde, a misma cantidad de aristas, menor cantidad de vertices.

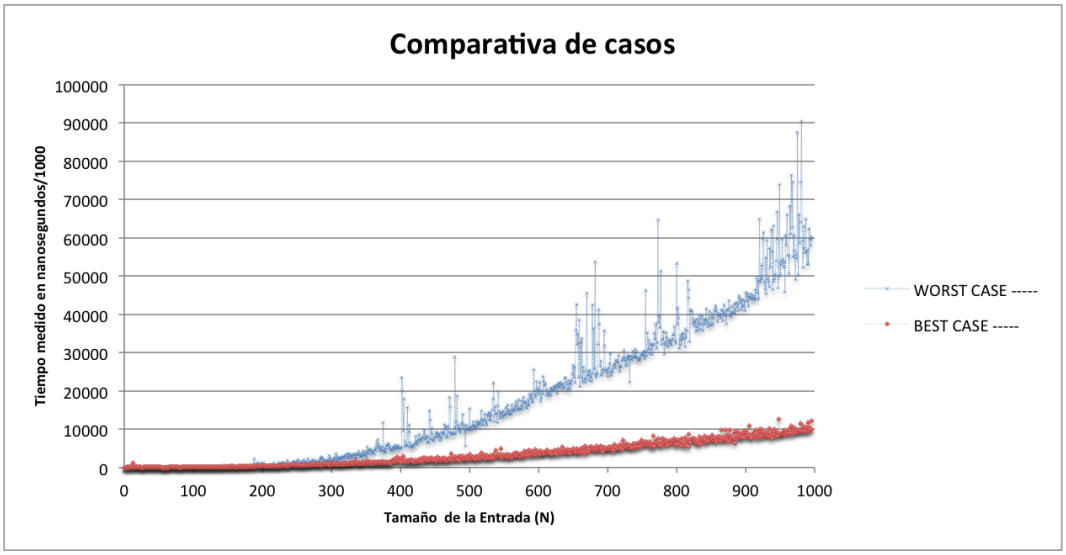


Figura 13: Tiempo de ejecución en función de la cantidad de aristas.

Luego decidimos ver que sucedia si no elegiamos las aristas del k5 para estar en un mejor caso y decidimos hacer que los pesos que se le asigaban fuesen random. En el siguiente grafico se observan cosas interesantes, como puede ser que hay dos o tres casos en el cual el random de pesos cayo en el mejor caso, pero tambien vemos que es muy particular ya que la mayoria de casos se mantienen con el crecimiento de la complejidad.

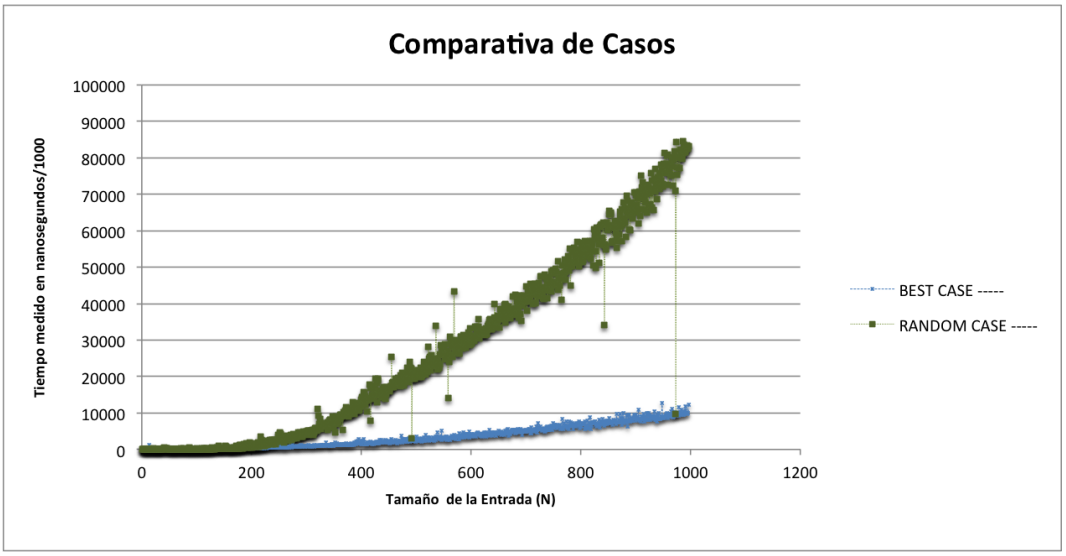


Figura 14: Tiempo de ejecución en función de la cantidad de aristas.

Hay tres picos en las mediciónes del caso random, esto lo consideramos como ruido de la virtual machine de java en la que corrieron nuestros tests.

4 Apendice

4.1 Cambios en Reentrega

En estos ejercicio se:

4.1.1 Ejercicio 1

-Mejoraron los casos de test y se rehicieron los graficos explicativos.
-Se rehizo la Correctitud.
-Se agregaron comentarios a la Justificación y Complejidad.

4.1.2 Ejercicio 2

-Se amplio la explicacion y el desarrollo del problema
-Se saco la demostracion de BFS por haber sido vista en clase
-Se cambio la estructura para almacenar los nodos, ahora utilizamos una matriz.
-Arreglo el análisis de complejidad.
-Se modifico el metodo *connect* dentro de la clase $Grafo2$ para que tuviera en cuenta portales dentro de un mismo piso
-Se agrego la posibilidad de solucionar el problema sin la optimizacion de cortar al encontrar el nodo buscado
-Cambiaron los tests para experimentar $optimazacionvsnooptimizacion$

4.1.3 Ejercicio 3

-Se Amplio la explicacion y el desarrollo del problema
-Se saco la demostracion de Kruscal por haber sido vista en clase
-Se mejoraron los casos de test y se re hicieron los graficos explicativos. - Se agregaron euristicas al union find para mejorar su complejidad. (By Rank y Path Compresion).