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
Bouzón, María Belén	128/13	belenbouzon@hotmail.com
Jiménez, Paula	655/10	puly05@gmail.com
Montepagano, Pablo	205/12	pablo@montepagano.com.ar
Rey, Maximiliano	037/13	rey.maximiliano@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. Problema 1: Saliendo del Freezer

1.1. Descripción de la problemática

Este problema modela un edificio de N pisos (y planta baja) en el cual en vez de tener escaleras o ascensores, hay portales que lo transportan a uno desde un piso a otro. Los portales solo permiten ascender de un piso más bajo a uno más alto. En cada piso puede haber cualquier cantidad de portales, teniendo en cuenta que no hay dos portales que salgan del mismo piso y lleguen a un mismo piso. Todas las instancias del problema deben tener al menos un camino de portales entre la planta baja y el piso N.

Se pide diseñar un algoritmo de complejidad O(n^2) que encuentre un camino que llegue desde planta baja hasta el piso N y utilice la mayor cantidad de portales posible.

El formato de entrada es un archivo de texto. Cada problema está representado en dos líneas de texto. La primera es el valor de N. La segunda es una lista de portales que sigue el siguiente formato:

```
pd ph[; pd ph]    // pd (piso desde), ph (piso hasta) enteros, pares de p separados por "; "
```

Para la salida, lo único que interesa es la cantidad de portales que hay que utilizar para recorrer el camino que usa la cantidad máxima posible de portales. La salida es una línea de texto que tiene ese valor.

Por ejemplo, un edificio de cinco pisos que tiene, desde cada piso, un portal para llegar a todos los pisos superiores se escribiría de este modo en el formato de entrada:

```
5
0 1; 0 2; 0 3; 0 4; 0 5; 1 2; 1 3; 1 4; 1 5; 2 3; 2 4; 2 5; 3 4; 3 5; 4 5;
```

Y el resultado a este ejemplo es, por supuesto, 5, ya que el camino más largo es el que pasa por todos los pisos.

1.2. Algoritmo desarrollado

El algoritmo diseñado utiliza la técnica de programación dinámica con un enfoque *bottom-up*. Podemos subdividir el problema original en sub-problemas. Los subproblemas implican resolver lo mismo pero, en vez de partiendo desde la planta baja, partiendo desde un piso más alto. Cuanto más cercano al piso superior (N) esté el piso desde el que partimos, más chico es el subproblema.

En primer lugar *parseamos* los datos de entrada, generando una tabla de tamaño N. La tabla contiene una lista de enteros en cada posición. La lista que se encuentra en la posición i-ésima de la tabla representa a los portales que salen desde el piso i. Los valores de la lista son los pisos a los que se llega desde ese piso usando un portal.

Luego, inicializamos en `-1` una tabla de resultados parciales. Nos referimos a resultados parciales porque en la posición i-ésima de la tabla resultados encontraremos la respuesta al subproblema de "máxima cantidad de portales que se pueden utilizar para llegar al piso N partiendo desde el piso i". Si no es posible llegar al piso N desde el piso i, guardaremos el valor -1 en esa posición.

Nótese que ninguna de las dos tablas tiene un lugar para el piso N, ya que no hay portales que partan desde el piso N ni tiene sentido el problema de llegar al piso N desde el mismo piso N.

Finalmente, el algoritmo itera por cada piso del edificio desde el piso $N - 1$ hasta el 0, iterando en cada piso por todos los portales del piso. Para cada portal calculamos la cantidad máxima de portales que se pueden usar para llegar a N usando ese portal. Esto se hace en tiempo constante, ya que hay que verificar solo tres cosas:

1. leer el valor de N del archivo de entrada
2. inicializar la tabla `portales` con listas (de enteros) vacías
3. inicializar la tabla `resultados` con `-1`
4. llenar la tabla `portales` con los portales de cada piso
5. Para cada piso `p` desde $N - 1$ hasta 0:

5.1. Inicializar variable `MAX` en -1

5.2. Para cada portal `l` en `portales[p]`:

5.2.1. inicializamos una variable `costo` en -1 para representar la máxima cantidad de portales que se pueden utilizar para llegar a N usando el portal `l`

5.2.2. Tenemos tres opciones:

5.2.2.1. Si el portal va directo a N, el costo es 1

5.2.2.2. Si el portal va a un piso que tiene en la tabla `resultados` valor -1, no es posible llegar a N usando ese portal, y dejamos el costo en -1

5.2.2.3. Finalmente, si el portal va a un piso que tiene en la tabla `resultados` un valor distinto a -1, entonces el costo será ese valor + 1 (por el portal `l`)[1]

5.2.3. Ahora que ya tenemos el costo, si el costo obtenido para este portal es mayor a la variable `MAX`, actualizamos el valor de ésta con el costo obtenido en esta iteración

5.3. Guardar en `resultados[p]` en valor de `MAX`

6. El resultado del problema completo se encontrará en `resultados[0]`

1.3. Tests de correctitud

Los casos de test provistos por la cátedra fueron replicados en el script `test.sh`. Para correrlo, primero es necesario tener el código del ejercicio compilado, lo cual se hace ejecutando el comando `make` en el directorio del código fuente.

1.4. Análisis de la complejidad

En en código fuente hemos comentado las distintas partes indicando los costos de cada una. Si llamamos N a la cantidad de pisos del edificio y P a la cantidad total de portales, la conclusión a la que llegamos es que el algoritmo tiene un costo de $\mathcal{O}(N + P)$.

Ahora bien, la cantidad de portales pertenece a $\mathcal{O}(N^2)$. En un edificio con la máxima cantidad posible de portales tendremos N portales en el piso 0, N-1 portales en el piso 1, etc. hasta el piso N-1, donde tendremos un solo portal. Esa sumatoria se escribe de este modo:

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} = \frac{N^2 + N}{2}$$

Por lo tanto, podemos reescribir el costo del algoritmo en función de N como perteneciente a $\mathcal{O}(N + N^2)$, lo cual pertenece a $\mathcal{O}(N^2)$, que es lo que se pide en el enunciado.

[1] Nótese que, como los portales siempre van a pisos más alto, las posiciones de `resultados` de los pisos más altos ya van a haber sido completadas en iteraciones anteriores del algoritmo.

1.5. Código fuente

```
int resolver(){
     int i = n-1;
     while (i>=0){  // este ciclo corre N veces (una vez por cada piso entre 0 y N-1)
        int max = -1;   // O(1)
        for (std::list<unsigned int>::iterator it=portales[i].begin(); it != portales[i].end(); ++it){
            // este ciclo corre tantas veces como portales haya en este piso
            int costo = -1;      // O(1)
            if (*it == n){       // O(1)
                // el portal va directo a N
                costo = 1;       // O(1)
            } else if (resultados[*it] == -1){      // O(1)
                // no se puede llegar a N por este portal
                costo = -1;      // O(1)
            } else{
                costo = resultados[*it] + 1;        // O(1)
            }
            if (costo > max){       // O(1)
                max = costo;        // O(1)
            }
        }
        resultados[i] = max;        // O(1)
        i--;                        // O(1)
    }
    int res = resultados[0];        // O(1)
    return res;                     // O(1)
}
```

1.6. Medición empírica de la complejidad

Para la medición de tiempos agregamos código al ejercicio de manera tal de que el programa pueda ser llamado con el argumento `--tiempos`. Al llamarlo de ese modo, vamos generando peores casos de tamaño creciente, y ejecutando, para cada uno, cincuenta veces el algoritmo. De cada corrida se toma la medición en milisegundos de la ejecución utilizando la biblioteca chrono de la STL de C++.

Los peores casos que generamos son aquellos donde existe la máxima cantidad posible de portales.

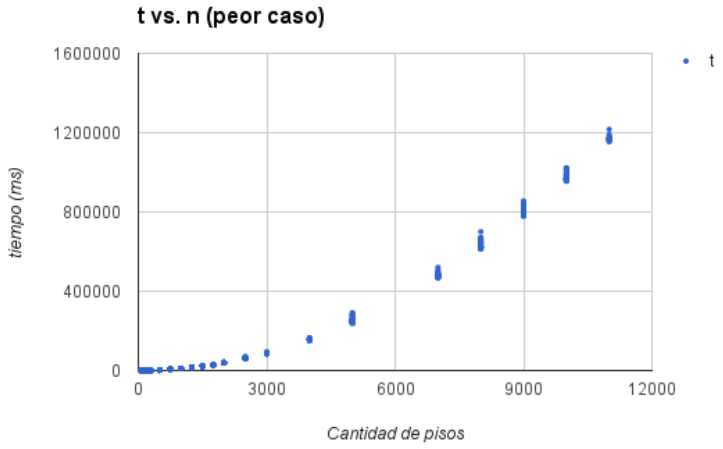


Figura 1: Relación entre tiempo de ejecución y cantidad de pisos (peor caso).

DIBUJAR DOS CURVAS DE N^2 QUE ENCIERREN A LA GRÁFICA EN EL GRAFICO 1.

Como se puede apreciar en los gráficos, el crecimiento de los tiempos de ejecución coincide con la complejidad calculada de forma analítica.

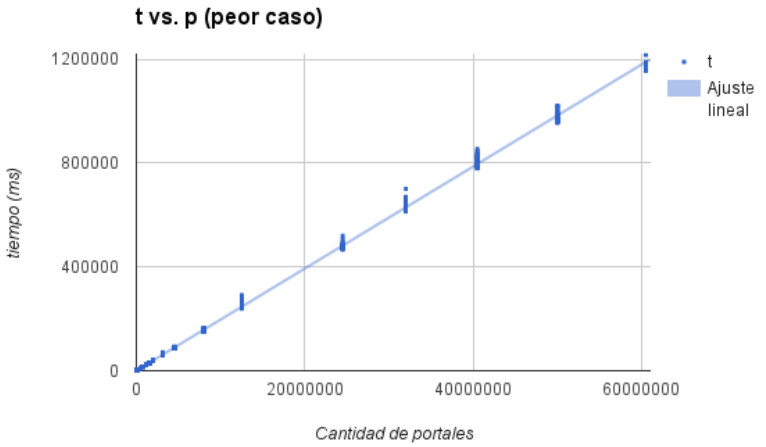


Figura 2: Relación entre tiempo de ejecución y cantidad de portales (peor caso).

2. Problema 2: Algo Rush

2.1. Descripción de la problemática

En este caso, se tiene un edificio con N pisos y con pasillos de longitud L y existen portales bidireccionales que comunican dos puntos determinados del mismo. Utilizar un portal consume dos segundos y caminar un metro consume un segundo. Se sabe que no hay más de un portal que comunique las mismas posiciones del mismo par de pisos. Se requiere un algoritmo que determine la cantidad minima de segundos en ir desde el inicio del piso 0 hasta el final del último piso.

2.2. Resolución propuesta y justificación

Lo primero que hace el algoritmo es transformar la entrada en un grafo. Según nuestra representación, cada portal es considerado como un nodo. Las aristas que parten de un nodo llegan a todos aquellos otros nodos a los que se puede alcanzar desde el actual, ya sea porque son un punto de destino de teletransporte del mismo, o porque se encuentran en el mismo piso y se puede alcanzar caminando.

Como estructura auxiliar para construir el grafo se utiliza un diccionario de diccionarios de nodos, cuyas claves para acceder a un nodo son el piso y la distaancia respectivamente. Un nodo contiene: un identificador (un int propio de cada nodo), la posicion en el pasillo donde se encuentra, un conjunto de nodos a los cuales se puede llegar caminando y un conjunto de nodos que son puntos de destino de teletransporte.

Posterirmente un segundo algoritmo, utilizando el grafo generado por el primero, se encarga de determinar el camino entre ambos extremos del grafo (desde el principio del piso 0 hasta el final del último piso, basadose en el BFS relajando ejes. El BFS se utiliza para calcular caminos mínimos en ejes sin peso; para esto, determina en cada paso la distancia al inicio de todos los adyacentes a determinado nodo X, considerando que, o bien el camino mínimo desde la raiz hasta el nodo evaluado pasaba por X, por lo cual es el camino mínimo hasta X más uno, o existía un camino más rapido hasta el nodo, pero entonces, el mismo ya se habrá evaluado anteriormente, por lo cual ya se conocerá el camino mínimo. El BFS relajando ejes consiste en adaptar un grafo pesado para simular un grafico sin pesos en aristas. Para esto, cuando una aristaa contiene un peso superior a 1, coloca nodos “fantasma” entre ambos para aumentar la distancia entre los nodos. El algoritmo utilizado en el TP utiliza una cola en donde ingresa los adyacentes de los nodos evaluados para procesarlos luego de forma ordenada. Para emular los nodos fantasma, el algoritmo crea un solo nodo extra, que almacena la cantidad de nodos fantasma que deberían existir, y, si el algoritmo detecta que el nodo evaluado emula a más de un fantasma, altera dicho valor y lo vuelve a encolar.

2.3. Análisis de la complejidad

Inicialmente, se debe construir un diccionario de N elementos y luego N diccionarios de L elementos, lo cual tiene un costo de O(nL). Luego se procesa la entrada que determina las posiciones de los portales y se van creando los nodos y clasificando en los diccionarios. Como la consulta y escritura en el diccionario es O(1), la complejidad total es de O(P). Finalmente, la complejidad de formar el grafo es de O(P+NL). Para la busqueda del camino mínimo se crea inicialmente un diccionario de N*L elementos, lo cual tiene una complejidad de O(NL). Luego comienza a ejecutarse el algoritmo que calcula el camnino mínimo. En el peor caso, el algoritmo debe recorrer cada uno de los nodos. Sabemos que el grafo no tiene más de N*L + P nodos,incluyendo los nodos fantasma, ya que dos nodos calificados en un mismo piso no pueden estar separados por una distancia superior a L; además, se debe agregar un nodo fantasma para marcar el paso de un nodo a otro por teletransporte, por lo cual existen P nodos fantasma extra. Entonces, la complejidad del algoritmo de camino mínimo es O(NL). Finalmente, el costo de construir el grafo y hallar el camino mínimo entre sus extremos es O(P+ NL).

2.4. Código fuente

2.5. Experimentación

2.5.1. Piensa con portales

En este test existe un portal que lleva directamente desde la entrada al aula. El algoritmo deberia utilizarlo y llegar al aula en dos segundos.

2.5.2. Encontre un atajo

En este test existen dos portales en el primer piso con una longitud considerable. El primero de ellos toma un camino largo hacia el aula y el otro uno más corto. Como debido a la distancia entre el origen de ambos caminos, el algoritmo comenzará a adentrarse en el camino largo antes del corto. Se pretende testear si el algoritmo finalmente devuelve la solución correcta a pesar de esto.

2.5.3. ¿y_como_llegue_aqui?

En esta oportunidad se introduce un ciclo en el grafico. Existen dos portales en el piso cero, uno de los cuales entra al ciclo y otro que lleva a la dirección correcta.

2.5.4. Linea recta

Este test se basa en probar la correctitud algoritmo cuando existen portales superpuestos. Existe un único camino entre el inicio y el aula, en donde todos los portales intermedios tienen otro superpuesto que lleva a otro piso, por lo que no es necesario camninar, excepto en el primer y el último piso.

2.5.5. cuello_botella

Este test tambíen esta centrado en probar el algoritmo cuando existen portales superpuestos. En este caso, existen muchos portales con un extremo en un mismo punto.

2.5.6. No quiero caminar

Este test se centra en explotar la posibilidad de utilizar un portal que comunica dos puntos del mismo piso. En el mismo, existen dos pisos con un portal en cada extremo y se necesita llegar de uno a otro dentro de cualquier camino existente; sin embargo, otro portal que cuyos extremos son muy cercanos a cada uno de los portales, por lo cual el camino mínimo deberia utilizar estos portales intermedios.

2.5.7. la_venganza_de_los_ciclos

En la entrada de este test se presentan dos ciclos en el grafo; dentro del camino mínimo se incluyen algunos nodos pertenecientes a los mismos. Nuevamente se pretende evaluar el comportamiento del algoritmo frente a los ciclos, pero en esta ocacion incluyendolos dentro de un camino simple hasta el aula.

2.5.8. salidas_de_emergencia

En esta ocación se prueba el uso intensivo de nodos fantasma. En este test solo existen portales que comunican el principio de un piso con el final del piso anterior.

2.5.9. Constrastación Empírica de la complejidad

Para poner a prueba de manera empirica la complejidad algorítmica calculada para nuestra resolución, diseñamos tres casos distintos de pruebas:

- Caso en el que se incrementa la cantidad de portales, dejando fijos los valores de la cantidad de pisos y la longitud de los pasillos.
- Caso en el que se incrementa la cantidad de pisos, dejando fijos los valores de la cantidad de portales y la longitud de los pasillos.
- Caso en el que se incrementa la longtud de los pasillos, dejando fijos los valores de la cantidad de pisos y la cantidad de portales.

Estos casos fueron diseñados partiendo del análisis de la complejidad dada. Siendo la misma O(NL+P), dedujimos que tomando como variable sólo una de ellas de manera alternada en tres casos distintos, se podrían obtener las funciones $f(N) = a*N + b$, $f(L) = a*L + b$ y $f(P) = b + P$, todas ellas lineales.

A continuación se pueden ver, en las figuras 3, 4 y 5 los resultados de las mediciones de tiempos obtenidas a partir de grafos válidos generados de forma automática para cada caso en particular.

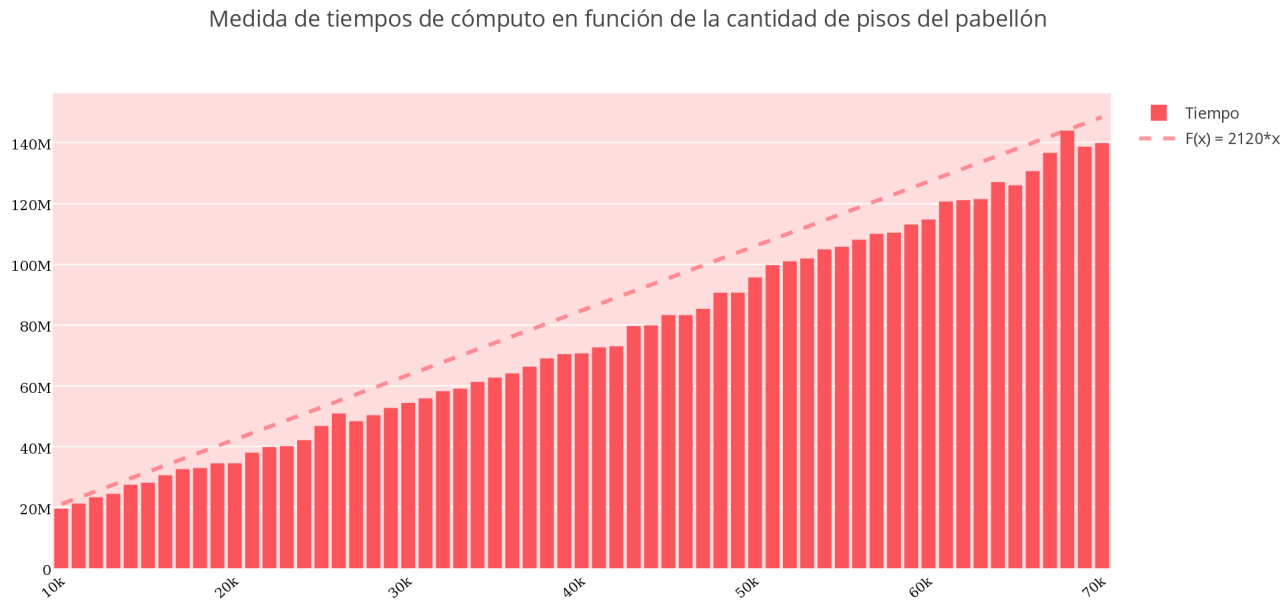


Figura 3: Medición de tiempos en función de la cantidad de pisos

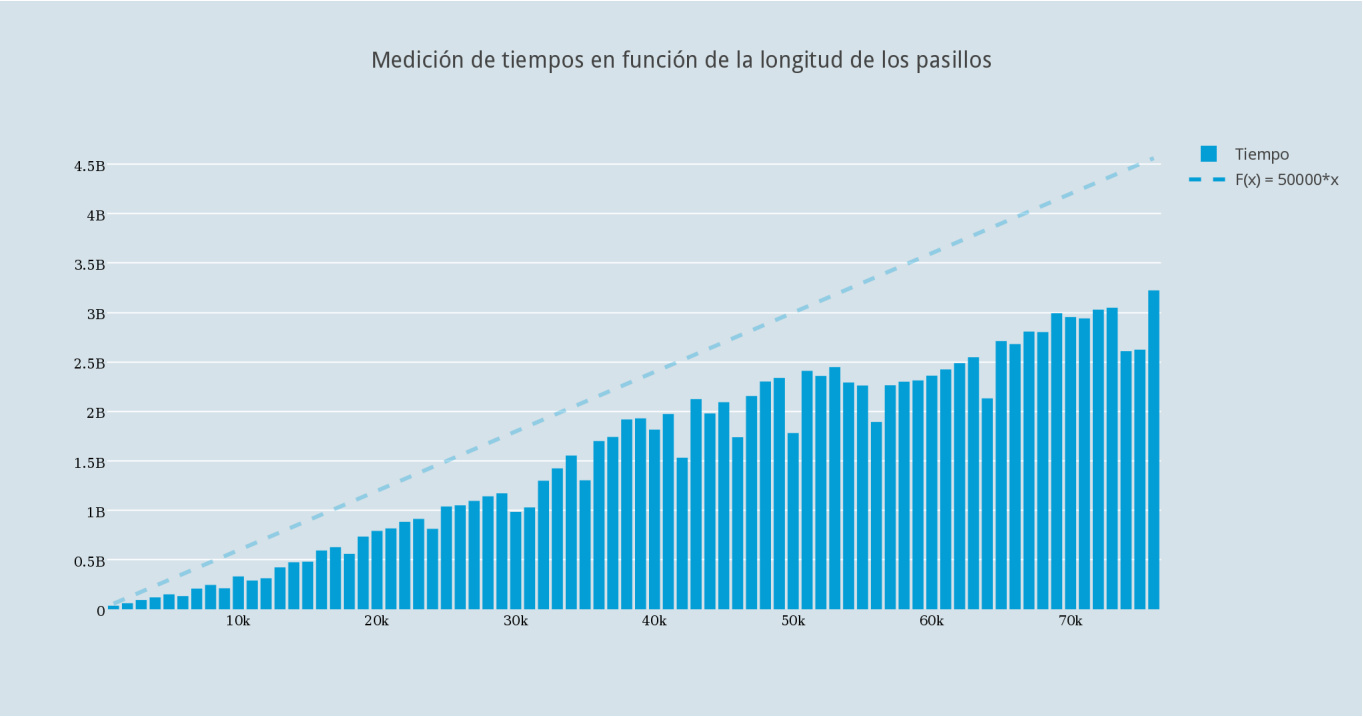


Figura 4: Medición de tiempos en función de la longitud de los pasillos

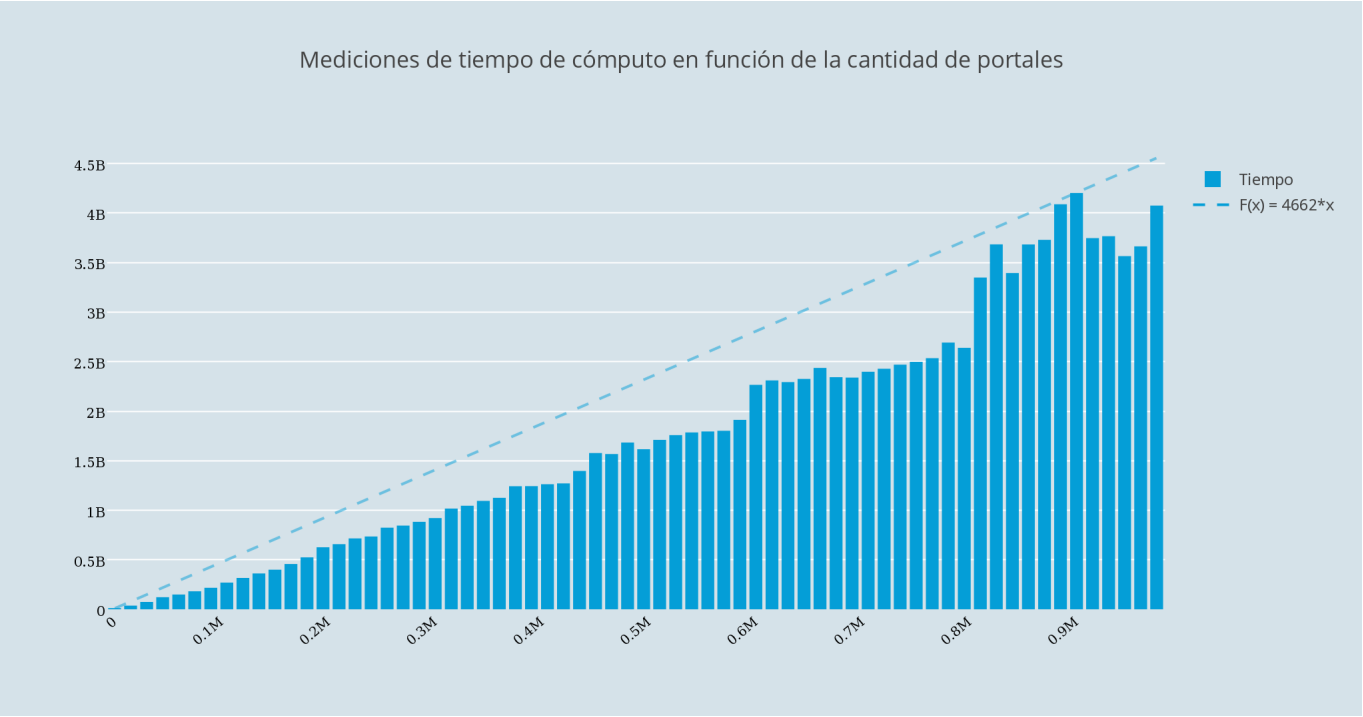


Figura 5: Medición de tiempos en función de la cantidad de portales

Como es posible observar, los tiempos medidos en cada uno de los casos parecen comportarse de forma tal que su crecimiento fuera lineal en función de la variable escogida.

Si bien tenemos presente que hacer estas pruebas no garantiza la reconstrucción de la complejidad O(NL+P) (del mismo modo que ninguna prueba empírica de este estilo *prueba* de ninguna manera una

determinada complejidad temporal) consideramos que las experimentaciones expuestas ayudan a reforzar en cierta medida los resultados teóricos deducidos a priori.

3. Problema 3: Perdidos en los Pasillos

3.1. Descripción de la problemática

En este ejercicio se nos presenta un pabellón con M pasillos de longitudes -potencialmente- distintas, representados por las aristas de un grafo y un conjunto vértices que pueden ser tanto intersecciones en las cuales dos o más de ellos convergen, o extremos incididos por un sólo corredor. A partir de este contexto se nos pide desarrollar un algoritmo que elimine cualquier ciclo del grafo dado, logrando crear un árbol generador cuyo peso (dado por la sumatoria de los pesos de cada arista) sea mayor o igual al de cualquier otro árbol generador posible (conocido como árbol generador máximo). Cabe destacar que queremos encontrar el árbol generador máximo ya que de esta manera nos aseguramos de que los pasillos a clausurar sean los de menor longitud en el ciclo y por lo tanto los que tienen menor costo de clausura.

Nota: Para correr el programa con un input específico ya sea para obtener el output y/o las mediciones de tiempo, utilizar la clase Main.java. En el caso que se quieran correr los tests, utilizar la clase TestEj3.java la cual utiliza la clase Ejercicio3.java. Main y Ejercicio3 hacen exactamente lo mismo salvo que la primera implementa las mediciones de tiempo y los métodos para leer y escribir archivos que no son necesarios para correr los tests.

3.2. Resolución propuesta y justificación

Para resolver el problema, desarrollamos un método que hace uso de una adaptación del algoritmo de Kruskal, mediante el cuál es posible encontrar un árbol generador máximo con una complejidad de $\mathcal{O}(m \log m)$ (siendo m la cantidad de aristas).
El algoritmo propuesto inicialmente ordena de mayor a menor cada pasillo de acuerdo a sus longitudes y los toma uno a uno verificando si conectan vértices entre los cuales ya existe un camino o no. De ser cierto, la arista en cuestión es descartada. De ser falso, la misma pasa a ser parte del árbol y se repite el procedimiento con la siguiente hasta que se hayan analizado todas. Las longitudes de las aristas descartadas se suman para llevar la cuenta de los metros de pasillo que deben ser clausurados.
Para que las complejidades se ajustaran a los requerimientos, fue necesario desarrollar la clase *UnionFind*, la cual implementa los métodos *findSet*, *unionSet* e *isSameSet* . El primero de ellos permite, dado un elemento, hallar al representante del conjunto en el que se encuentra. El segundo, por su parte, realiza la unión de dos conjuntos. Por último, el tercero analiza si los dos valores parametrizados están incluidos en el mismo conjunto.

El pseudocódigo de nuestro algoritmo es el siguiente:

```
Ordenar pasillos de mayor a menor
suma = metros a clausurar (inicializada en cero)
for Cantidad de pasillos do
  if El pasillo conecta dos conjuntos disjuntos de pasillos then
    Unir el conjunto mas chico al mas grande
  else
    suma+ = la longitud del pasillo actual
  end if
end for
devolver suma
```

El algoritmo para ordenar la lista de pasillos asumimos que es correcto ya que estamos utilizando el método *sort* de *java.util.Collections* para ordenar de mayor a menor. Luego decimos que el ciclo termina ya que al utilizar un iterador de una lista finita y ciclar mientras exista un elemento siguiente, se puede asegurar que el ciclo va a llegar al último elemento y terminar por poscondición de la función *hasNext()*.

Dentro del ciclo analizamos cada arista para decidir si pasa a formar parte del árbol generador máximo ya que esa arista une dos componentes conexas o en caso contrario, al agregarla se estaría generando

un ciclo, entonces solo se guarda la longitud y se continúa con la próxima en la lista. Al tener las aristas ordenadas de mayor a menor nos aseguramos que siempre vamos a intentar agregar las de mayor peso primero y que si al probar con un eje se forma un ciclo C, ese eje va a ser el de menor peso de C.

Para probar que esto es correcto tenemos que analizar el funcionamiento de la clase *UnionFind* dado que se encarga de decirle a kruskal, de manera eficiente, si tiene que poner la arista o no. El constructor crea dos arreglos y utiliza un *for* para inicializarlos, el cual tiene un invariante simple que asegura que cuando se cumple que $\{i >= n\}$ termina. Al crear estas estructuras definimos un bosque que no tiene ninguna arista y cada nodo es su propio padre. También definimos un *rank* que nos va a servir para comparar los tamaños de las componentes conexas que vaya creando kruskal.

En *isSameSet* no hay nada que analizar ya que solo llama a *findSet* con los parámetros de entrada y devuelve true si dio el mismo resultado para i y para j. *findSet* se encarga de averiguar cual es la raíz del árbol[2] al que pertenece el nodo que le llega por parámetro y lo hace de la siguiente manera:

- obtiene el padre directo del nodo.
- copia el nodo en una nueva variable $elemento$.
- cicla mientras el padre del elemento no sea si mismo.
- Dentro del ciclo el elemento pasa a ser el padre y se obtiene el padre de este nuevo elemento, de esta manera se van actualizando las variables obteniendo los ancestros hasta que eventualmente se llega a la raíz, la cual es la condición de terminación del ciclo ya que todo árbol tiene raíz, y por lo tanto termina.
- Comprime el árbol modificando al nodo para que su padre directo se la raíz, salteando a todos los ancestros que hubiera en el medio.
- Por último devuelve al nodo raíz.

Por último tenemos a *unionSet* que se encarga de encontrar la mejor manera de unir dos componentes conexas. Para eso obtiene las raíces de los dos nodos a unir y los respectivos ranks de las raíces. Luego une el árbol de menor rango al de mayor ya que es menos costoso unir el mas chico al mas grande que al revés. La unión es simplemente definir a la raíz de árbol mas grande, como el padre de la raíz del mas chico, agregando así una nueva rama. En el caso que los dos tuvieran el mismo rango se le suma uno a la nueva raíz para constatar que el árbol que se formó es mas grande que los dos que se tenía anteriormente.

En conclusión, podemos asegurar que en cada paso de kruskal agregamos una arista al árbol generador o sumamos la longitud, del pasillo a clausurar, a la solución del problema y como el AG que se genera es máximo, entonces se cumple que la suma de los metros a clausurar es mínima.

3.3. Análisis de la complejidad

Nuestra solución tiene una complejidad temporal de $\mathcal{O}(m \log m)$ siendo m la cantidad de aristas del grafo. En el análisis también se va a utilizar n como la cantidad de nodos.

Para empezar, al crear el grafo, lo primero que hacemos es ordenar la lista, lo que tiene una complejidad de $\mathcal{O}(m \log m)$ por la documentación de *sort*[3]. Luego se crea una nueva instancia de *UnionFind* dentro de la cual se ejecuta un *for* n veces por lo que podríamos decir que la complejidad es $\mathcal{O}(n)$, pero como el enunciado asegura que las instancias del problema siempre son conexas, entonces $m = n - 1$ como mínimo y como estamos buscando sacar ciclos, la mayoría de las instancias van a tener $m > n$ por lo tanto se puede asumir como cota superior, que tarda $\mathcal{O}(m)$.

La función *kruskal( )* se basa en un ciclo que recorre todas las aristas de la lista una sola vez, con lo cual ejecuta $\mathcal{O}(m)$ iteraciones. Dentro del mismo se utilizan las funciones de *UnionFind*. A continuación pasamos a detallar las mismas:

[2]Definimos *raíz* al nodo que se tiene como padre a si mismo y que un árbol tiene una sola raíz.

[3]http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#sort(java.util.List, %20java.util.Comparator)

- *isSameSet* se utiliza como máximo una sola vez dentro del ciclo y esta llama a *findSet* la cual ejecuta un *while* que en teoría itera $\mathcal{O}(altura\_del\_arbol)$ en el peor caso, pero por los teoremas demostrados en clase y como estamos utilizando la *representación con bosques*, *path compression* y *union by rank* se puede decir que la complejidad de este paso queda acotada por $\mathcal{O}(m)$ y puede ser despreciada.
- *unionSet* también se ejecuta a lo sumo una vez. Dentro de la función se llama dos veces a *findSet* pero como necesariamente (en esta implementación) se llamó antes a *isSameSet* para los mismos argumentos, entonces se puede afirmar que ya se realizó el *path compression* para esos nodos, por lo tanto se obtiene la raíz en $\mathcal{O}(1)$. Obtener el rank, setear al nuevo padre e incrementar un rank también toman tiempo constante.

En la implementación de *kruskal* se hizo una pequeña optimización que consiste en contar la cantidad de aristas que se agregan al AG para que una vez que se llegue al máximo $(n-1)$ no se intente agregar mas al árbol y solo se sumen las distancias. Sin embargo esto no modifica la complejidad final ya que de todos modos se van a tener que recorrer todas las aristas.

Analizando las complejidades de las distintas partes se puede ver que la que mas pesa es la de ordenar las aristas y por lo tanto la cota final es la que afirmamos al principio:

$$\mathcal{O}(m \log m) + \mathcal{O}(c * m) = \mathcal{O}(m \log m) \text{ Con } c \text{ una constante.}$$

3.4. Código fuente

A continuación se incluyen las partes más relevantes del código.
La clase *Main.java* se basa en crear el grafo con la lista ps y llamar a *kruskal*:

```
        nodos = lect.CantIntersecciones();
        grafo = new Grafo(nodos, ps);
        esc.EscribirInt(grafo.kruskal());
```

Los métodos que manipulan al grafo que se encuentran en la clase *Grafo.java*

```
public Grafo( int n, List<Pasillo> ps ) {
   OrdenarDesc(ps);
   _pasillos = ps;
   _nodos = n;
   _sets = new UnionFind(n);
}

public void OrdenarDesc(List<Pasillo> ps){
   //ps.Sort(new PasilloComparator()); //Con java 7 no compila.
   Collections.sort(ps, new PasilloComparator());
}

public final int kruskal() {
   int suma = 0;
   int aristasAG = 0;
   Pasillo p;
   ListIterator<Pasillo> it = _pasillos.listIterator();
   while(it.hasNext()){
      p = it.next();
      if(aristasAG < _nodos - 1){
         if(_sets.isSameSet(p.getExtremo1(), p.getExtremo2())){
            //si estan en el mismo set
            //tengo que sumar la long del pasillo a clausurar.
            suma += p.getLongitud();
         }else{
            _sets.unionSet(p.getExtremo1(), p.getExtremo2());
            //sumo una arista al AG.
            aristasAG++;
         }
      }else{
         // si ya puse n-1 aristas solo sumo las distancias.
         suma += p.getLongitud();
      }
   }
   return suma;
}
```

Por último la clase *UnionFind.java*

```
1   public UnionFind( int n ) {
2     // Crea un bosque de n nodos.
3     this._parent = new ArrayList<Integer>(n);
4     this._rank = new ArrayList<Integer>(n);
5     // Cada nodo tiene rank 0 al principio.
6     // Cada nodo es su propio padre al principio.
7     for(int i = 0; i < n; i++){
8        this._parent.add(i);
9        this._rank.add(0);
10    }
11  }
12
13  public final int findSet( int i ) {
14  // Si el padre es el mismo nodo, devuelvo ese nodo.
15  // Si no, busco el padre del padre hasta llegar a la raiz
16  // y actualizo el padre del nodo.
17  int parent = this._parent.get(i);
18  int element = i;
19  while(parent != element){
20     element = parent;
21     parent = this._parent.get(element);
22  }
23  this._parent.set(i, parent);
24  return parent;
25  }
26
27  public final boolean isSameSet( int i, int j ) {
28     // Devuelve si 2 nodos pertenecen o no al mismo conjunto.
29     int root_i = this.findSet(i);
30     int root_j = this.findSet(j);
31     return root_i == root_j;
32  }
33
34  public final void unionSet( int i, int j ) {
35    // El que tenga menor rank pasara a formar parte del que tenga mayor rank.
36    // Si ambos tienen igual rank es lo mismo cual uno a cual, pero debo
37    // aumentar el rank del que sea el padre.
38    int root_i = this.findSet(i);
39    int root_j = this.findSet(j);
40    int rank_i = this._rank.get(root_i);
41    int rank_j = this._rank.get(root_j);
42
43    if(rank_i < rank_j){
44       this._parent.set(root_i, root_j);
45    }else{
46       this._parent.set(root_j, root_i);
47    }
48
49    if(rank_i == rank_j){
50       this._rank.set(root_i, rank_i + 1);
51    }
52  }
```

3.5. Experimentación

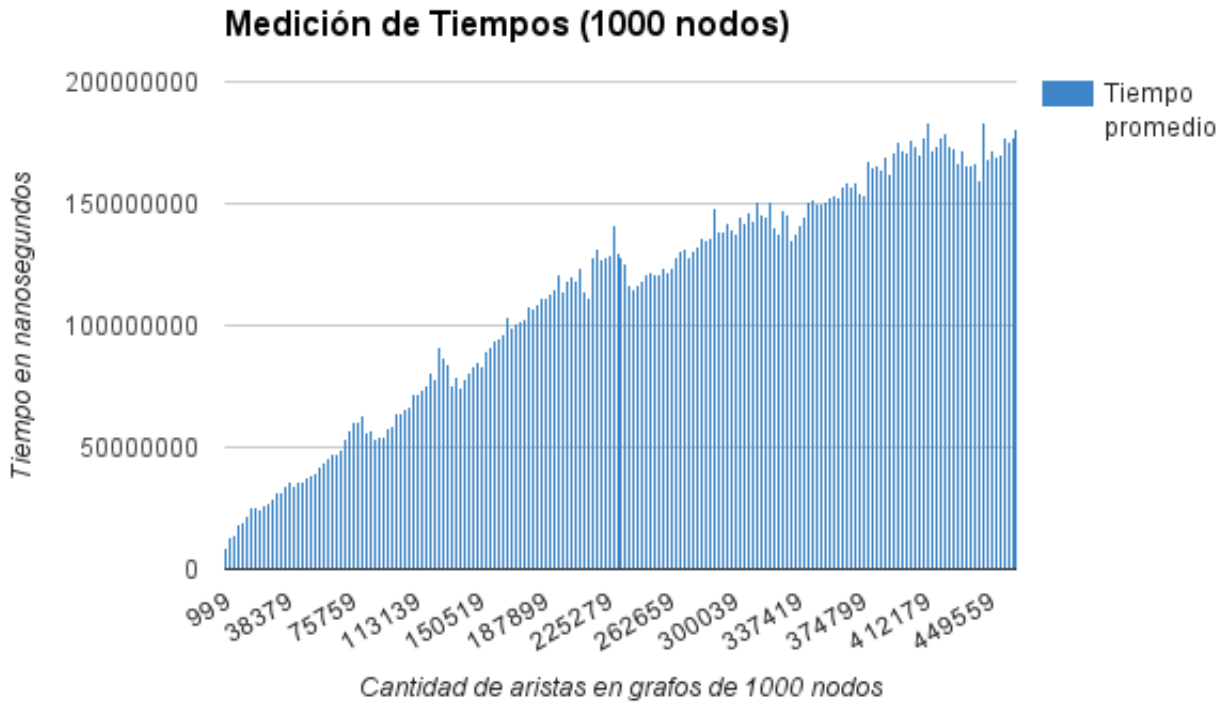
Para analizar la performance del algoritmo creamos dos scripts: uno que se encarga de generar casos de test pseudoaletorios[4], el cual utiliza un script de python[5] que se encarga de generar grafos conexos con un n y un m dados. Combinando estos dos fijamos el n, luego generamos grafos conexos con aristas elegidas al azar y con m que empieza valiendo $n-1$ y se va incrementando con un cierto espaciado. El otro script[6] se encarga de correr el programa una cierta cantidad de veces para cada instancia generada por el primer script, luego descarta los outliers y calcula el tiempo promedio entre todas las corridas. Estos promedios se guardan en el archivo *resultados-N.out* con N la cantidad de nodos que se fijó para generar los grafos.

[4] genTest.sh
[5] random_connected_graph.py; Source: https://gist.github.com/bwbaugh/4602818#file-random_connected_graph-py
[6] testPerformance.sh

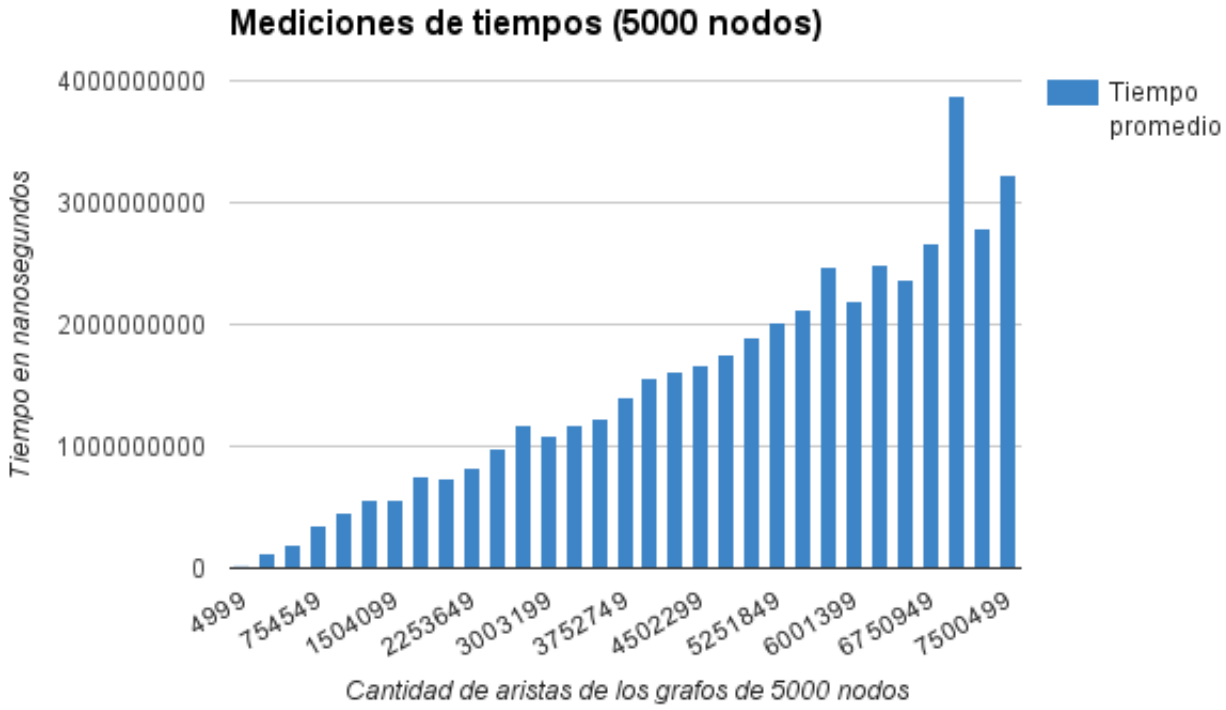
3.5.1. Constrastación Empírica de la complejidad

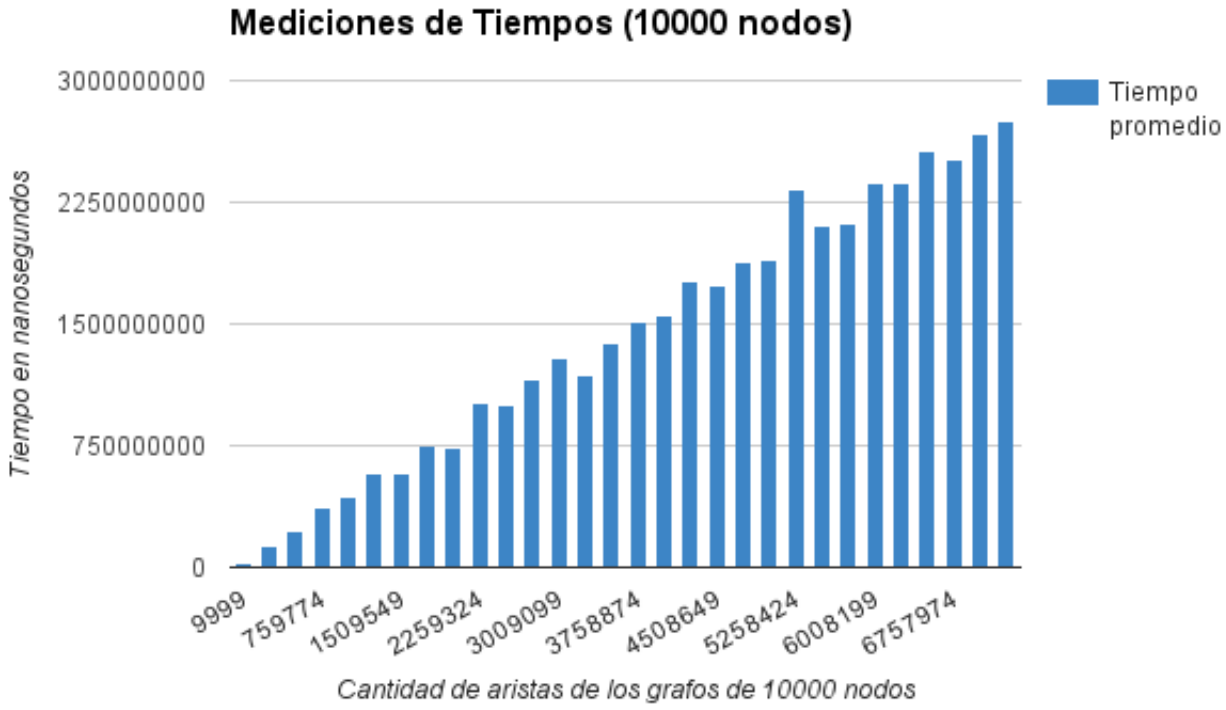
A continuación presentamos tres gráficos que muestran el comportamiento en la práctica del algoritmo. Para obtener estos resultados fijamos tres valores de n y luego se fueron modificando los valores de m. Las instancias fueron corridas varias veces para minimizar el error de medición.

El primer gáfico presenta grafos de 1000 nodos, el segundo grafos de 5000 y el tercero de 10000.

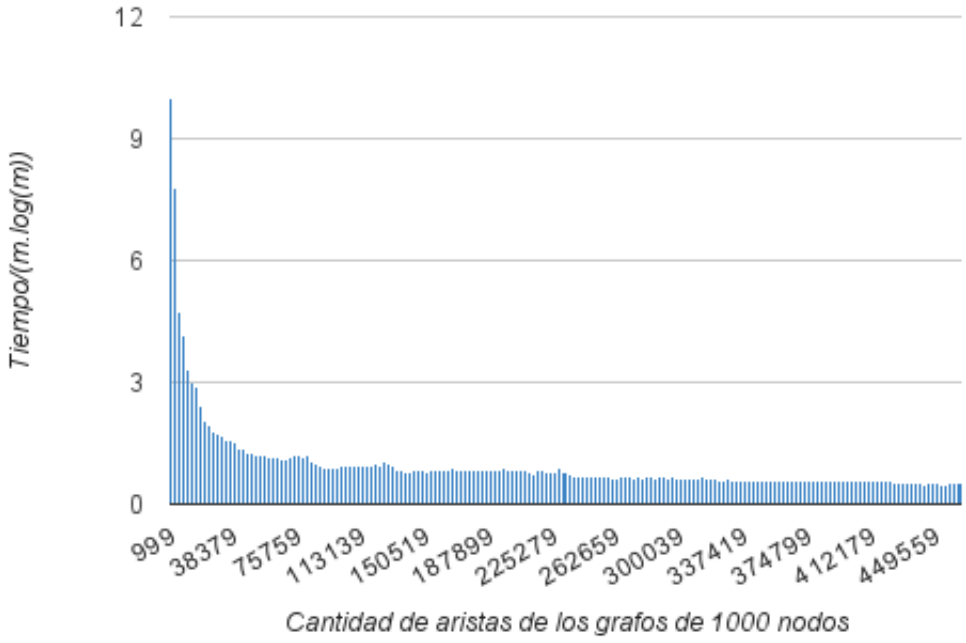


Se hicieron estos gráficos fijando el n a modo de mostrar que efectivamente la cantidad de nodos no influye en la complejidad final del algoritmo y como se puede observar las curvas son similares. Los dos últimos grafos tienen menos instancias dado que generar grafos mas grandes y luego correr el programa resultaba en un problema de falta de poder de procesamiento y de memoria física que soportara las estructuras de datos de mas de 200 MB.





Para finalizar con este análisis tomamos las muestras de los grafos de 1000 nodos y las dividimos por $m \log m$ para corroborar que la complejidad teórica se condice con la realidad y concluímos dadas estas muestras, el gráfico parece tender a una constante por lo que decimos que nuestro algoritmo cumple con la cota de complejidad propuesta.


12
9
6
3
0
Tiempo/(m.log(m))
999
38379
75759
113139
150519
187899
225279
262659
300039
337419
374799
412179
449559
Cantidad de aristas de los grafos de 1000 nodos