Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Informe 2

Integrante	LU	Correo electrónico
Jazmín Alvazer Vico	75/15	jazminalvarezvico@gmail.com
Marcelo Pedraza	393/14	marcelopedraza314@gmail.com
Uriel Jonathan Rozenberg	838/12	rozenberguriel@gmail.com
Javier María Cortés Conde Titó	252/15	javiercortescondetito@gmail.com

Reservado para la cdra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Problema 1: Laberinto

1.1. Introducción

En este problema los viajeros encuentran un mapa de un laberinto señalando algún lugar con una x, sin embargo no todos los puntos están conectados. Ellos quisieran llegar a ese lugar caminando lo menos posible. Además, pueden esforzarse para romper una cantidad determinada de paredes. Nos piden que, de ser posible, encontremos la manera que caminen lo mínimo posible

Formalmente podemos modelar el problema utilzando un grafo que contiene nodos "especiales"(las paredes). Se busca la distancia entre dos nodos, pasando como mucho por p nodos especiales.

1.2. Explicación de la solución

La entrada de nuestro problema es una matriz con "." , "#" , una "o" marcando el origen y una "x" marcando el destino.

Podemos dividir la resolución en tres partes: la primera, donde interpretamos la entrada (una matriz) y generamos un primer grafo, que tiene cada coordenada de la matriz como nodo. Nuestra clase nodo tiene un booleano para distinguir si un nodo es pared o no. Entre dos nodos (sean de tipo pared o no) consecutivos, siempre habra un eje. Ademas los nodos cuentan con un entero que es indicador de a que nivel pertencen. Al inciar todos los nodos son de nivel 0.

En la segunda parte copiaremos el primer grafo tantas veces como paredes se puedan romper, de manera que la acción de romper una pared esta representada por un cambio de nivel en el grafo. Así, cada nivel muestra cuántas paredes fueron atravesadas, y los $p + 1$ nodos finales, los finales con p paredes atravesadas.

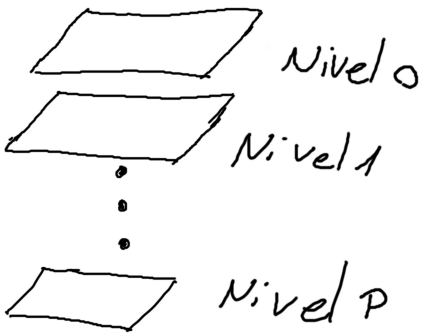


Figura 1: imagen explicativa

En la tercera parte, vamos a aplicar un bfs modificado para calcular las distancias del nodo origen a todos los demás nodos. Entre las modificaciones se incluyen unos "movimientos ilegales" . Por un lado, no está permitido pasar a un nivel menor al que uno se encuentra. Además, no es correcto mover de un nodo "habitación" a un nodo "pared" en el mismo nivel. Estas son acciones que en nuestro

modelo no tienen correlación con ninguna situación del problema, y que podrían generar camínos mínimos incorrectos.

Por último, se calcula la distancia mínima entre todos los nodos finales copiados. Si no existe un camino posible a ninguno de los nodos finales en todos los niveles, el algoritmo devuelve -1

1.3. Pseudocódico

1.3.1. Pseudocódigo

solucion(vector<Nodo>nodos, vector<Eje> ejes, int p)

```
1:  vector<Nodos> aux←nodos
2:  Desde i = 0 hasta i = p
3:     aux← ClonarUltimoNivel(aux,ejes)
4:  finDesde
5:  res← Bfs(aux)
6:  devolver res
```

Este algoritmo llama a ClonarUltimoNivel la cantidad de paredes que se tiene permitido romper, y luego hace un bfs sobre el nuevo grafo.

```
Bfs(vector<Nodos> nodos)
 1:  Para todo nodo i
 2:    pred(i)← −1, order(i)← −1
 3:  finPara
 4:  final ← −1
 5:  next← 0
 6:  list←(0,0)
 7:  Mientras list ≠ ∅ hacer
 8:    Sacar un elemento (i,padre)
 9:    pred(i,padre)=padre
10:    Si order(i=-1) hacer
11:      order(i)←next
12:      incrementar next
13:      Si i es un nodo final
14:        final ←i
15:        cortar el mientras
16:      Para cada nodo j vecino al nodo i hacer:
17:        Si(order(j)= −1 y nivel(j) ≥ nivel(i) y (nivel(j)≠nivel(i) o j no es pared ) ) hacer
18:          list∪(j,i)
19:        finSi
20:      finPara
21:    finSi
22:  finMientras
23:  res←2
24:  aux←0
25:  Si final = −1
26:    devolver −1
27:  finSi
28:  Mientras final ≠ 0 y final ≠ −1 hacer
29:    final←pred(final)
30:    incrementar res
31:    incrementar aux
32:  finMientras
33:  devolver res
```

Este algoritmo es un bfs clásico, con dos agregados. Por un lado, en la línea 13 preguntamos si el nodo en el que estamos es un nodo final (la habitación a la que queremos llegar. Recordemos que vamos a copiar el nodo p veces, entonces existirán p+1 nodos finales). Por otro lado la guarda de la línea 17 no sólo contempla que no haya visitado el nodo vecino antes en el bfs (order(j)=−1), si no que suma las condiciones mencionadas anteriormente: que esté en el mismo nivel o, si está en uno menor, que no sea pared (nivel(j) ≥ nivel(i) y (nivel(j)≠nivel(i)).

```
ClonarUltimoNivel(aux,ejes)
 1:  n ←último nodo de aux
 2:  nivel ← el nivel del n
 3:  tamanioDelUltimoNivel ← 0
 4:  mientras exista n y el nivel de n = nivel
 5:    retrocedo un nodo
 6:    incremento tamanioDelUltimoNivel
 7:  finMientras
 8:  Para i desde 0 hasta tamanioDelUltimoNivel
 9:    nuevoNodo ←copia de n+i
10:    indice del nuevoNodo← indice de n + tamanioDelUltimoNivel
11:    agrego nuevoNodo al final de aux
12:  Fin Para
13:  Para e en ejes
14:    Si e tiene ambos ejes incidentes a nodos en el ultimonivel
15:      nuevoEje←copia del eje, pero incidente a los nodos del nuevo nivel
16:      Agrego nuevoEje al final de ejes
17:    fin Si
18:  Fin Para
19:  para n en nodos
20:    Si n es pared
21:      Para e en los ejes de n
22:        nuevoEje←copia del otro eje que conecta n con su clon del nuevo nivel
23:        Agrego nuevoEje al final de ejes
24:      Fin Para
25:    Fin Si
26:  Fin Para
```

El objetivo de este algoritmo es clonar el grafo de manera particular, de modo de terminar con un grafo que simule la acción de romper una pared, pasando a otro nivel. El primer ciclo (línea 4) cuenta cuántos nodos tiene el grafo original. Luego arma un nuevo nivel copiando los nodos del nivel anterior, y agregándolos a la lista de nodos. En los siguientes ciclos se copian los ejes. Si en el nivel anterior dos nodos estaba conectados entre si, lo estarán e el nuevo nivel. Si uno de ellos es pared, se conecta la habitación del nivel anterior con la pared del nuevo nivel.

1.4. Demostración de Correctitud

Para demostrar que este algoritmo devuelve lo pedido, podemos ver que todos los movimientos entre habitaciones están representados por la posibilidad(o no) de visitar a un vecino. De esta manera, todos los caminos posibles(y sólo esos) están considerados a la hora de hacer bfs.
Movimientos legales:

Moverse de una habitación a otra; corresponde a 2 nodos "habitación" adyacentes. Como en todos los niveles creados esta adyacencia se mantiene, este movimiento siempre es posible.

Romper una pared; corresponde a un "cambio de nivel". Se pasa de un nodo "habitación" de nivel i a un nodo "pared" de nivel i+1. Como existen P niveles, se pueden observar todos los caminos dis-

tintos, rompiendo las paredes en distintas situaciones. Notar que el algoritmo explícitamente prohíbe moverse a una pared en el mismo nivel, para evitar falsos resultados.

Moverse de una pared a otro lado; este caso está siempre conectado en un mismo nivel, y con una guarda en el bfs para evitar la vuelta meniconada en el caso anterior.

1.5. Demostración de Complejidad

Para poder probar que este algoritmo cumple la complejidad pedida, vamos a demostrar que en el grafo final(luego de la clonación) los ejes son, como mucho, ocho veces la cantidad de nodos. Podemos ver que, por un lado, en el grafo sin clonar, cada nodo puede tener dos(si está en una de las cuatro esquinas del tablero) tres(en los bordes) o cuatro(en el medio) ejes. Además, a la hora de clonar, creas un nuevo eje si y sólo si los adyacentes son distintos. Es decir, una habitación y una pared. De esta manera, el caso máximo de vecinos que puede tener un nodo es si todos sus vecinos son del otro tipo, tanto si es una habitación rodeada de paredes, o una pared rodeada de habitaciones.

Una vez pasado por eso, vamos a analizar la función *clonarUltimoNivel*. Como todos los elementos de la matriz menos los bordes son nodos, es lo mismo decir que la complejidad es $\mathcal{O}$(F*C). En la solución, por un lado se aplica *ClonarUltimoNivel* p veces (dando una complejidad de $\mathcal{O}$(F*C*P)), y luego se aplica bfs sobre el grafo clonado. El grafo clonado tiene $n*p$ nodos, con una cantidad de ejes acotado por lo mencionado anteriormente, llegando a una complejidad de $\mathcal{O}$(F*C*P). En conclusión, el algoritmo propuesto tiene una complejidad de $\mathcal{O}$(F*C*P), cumpliendo con la cota pedida.

1.6. Experimentación, resultado y análisis

En los primeros dos gráficos utilizamos instancias muy similares. En el primero fuimos variando la cantidad de columnas, manteniendo constante la cantidad de filas, empezando con 10 filas, terminando con 250 y aumentando de 10 en 10. En el segundo realizamos la instancia análoga manteniendo contante las columnas. La "X" siempre se encuentra en el extremo opuesto a la "o".

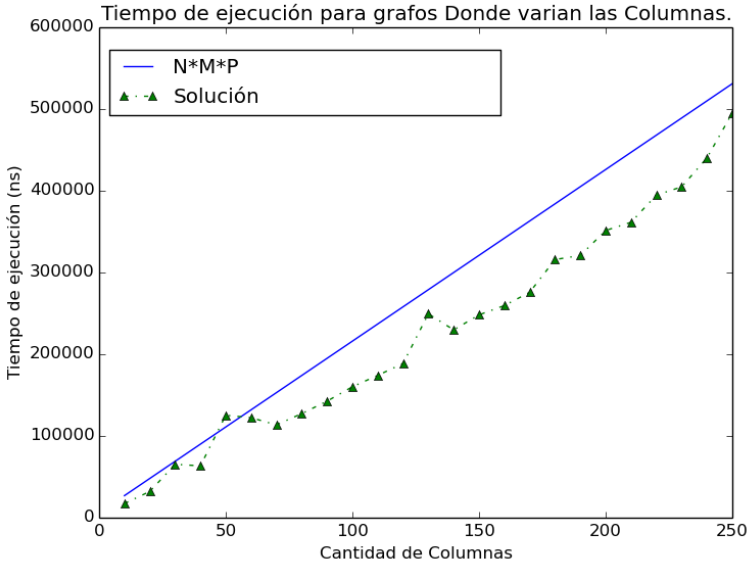


Figura 2

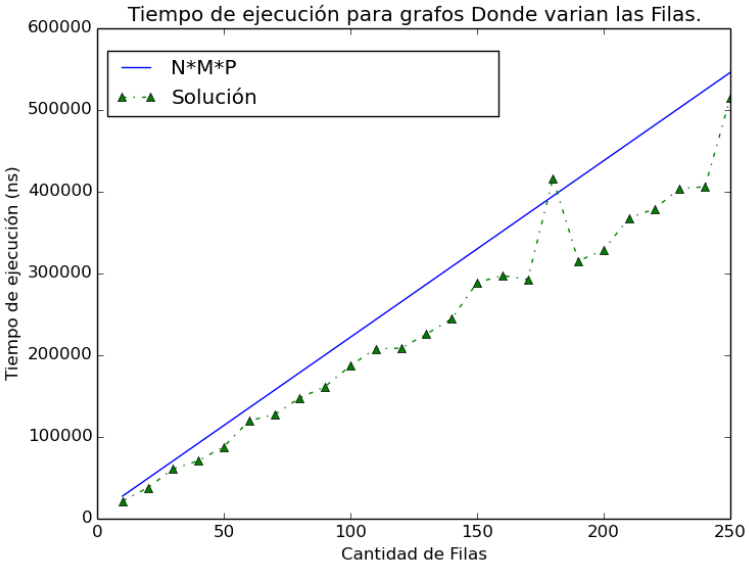


Figura 3

Podemos observar que ambos gráficos tienen tendencia lineal. También notamos que tienen el mismo rango de tiempo de ejecución. Podemos concluir que el algoritmo no es afectado por la cantidad de filas y columnas en sí, si no por la cantidad de nodos.

En los siguientes dos gráficos utilizamos una instancia similar a la anterior. La única diferencia es que aumentamos la cantidad de componentes conexas, alternando filas o columnas de "#" con las de ".". Decimos que son componentes conexas, ya que tenemos p=0, o sea, no podemos romper ninguna pared, entonces esos nodos nunca podrán ser conectados.

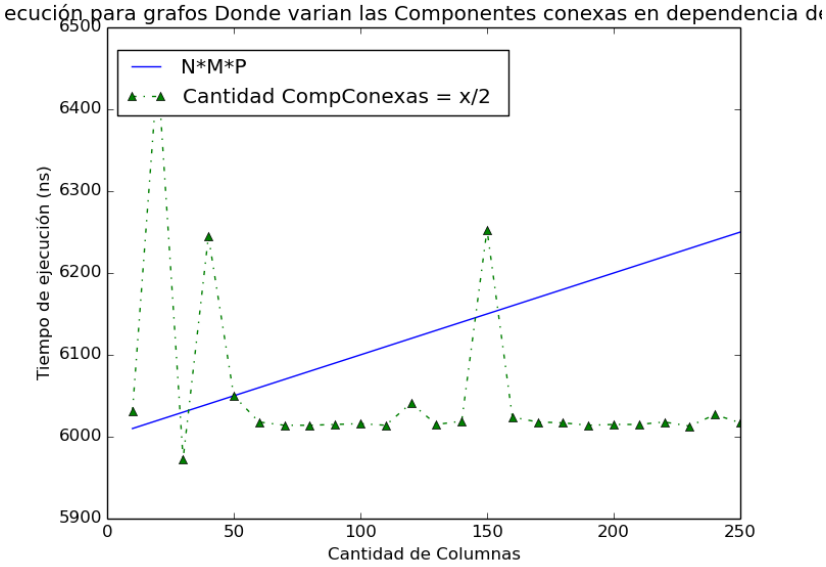


Figura 4

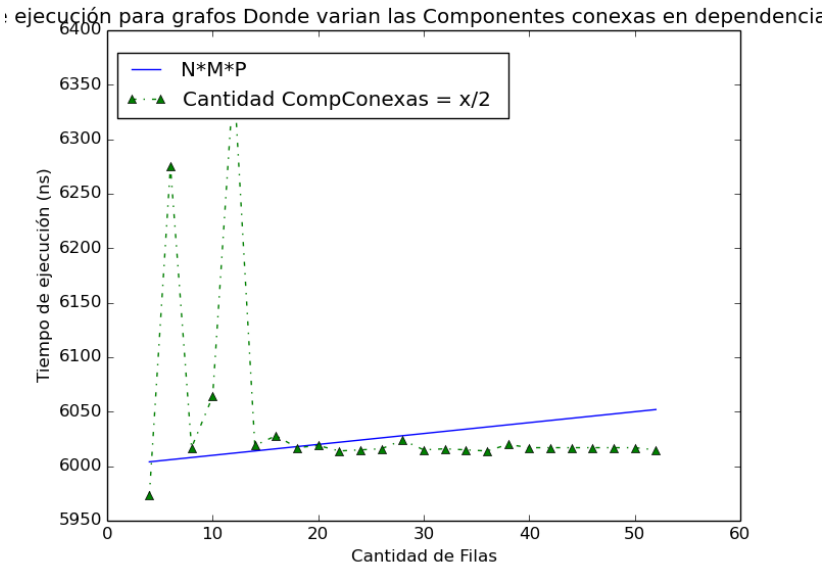


Figura 5

Igual que antes podemos observar que ambos casos tienen comportamientos similares. Sin embargo a diferencia de los casos sin mas de una componente conexa,

Para el análisis de este algortimo lo que hicimos fue separar las mediciones en dos casos, teniendo fc y p variable.Esto significa que tomamos laberintos cuyo tamaño estaba fijo y fuimos midiendo el algoritmo aumentando el valor de P paulativamente.

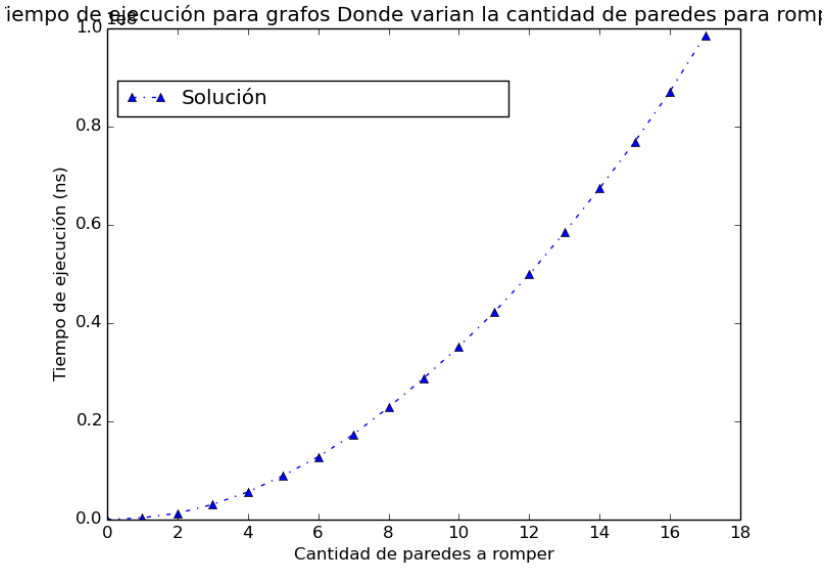


Figura 6

Se aprecia en este gráfico que si solo varía P, obtenemos un comportamiento lineal respecto a FC, dado que el cambio es la clonación de un nivel.

2. Problema 2: Juntando piezas

2.1. Introducción

En este problema nuestros arqueólogos quieren recorrer todas las salas, para ello pueden romper las paredes. Sin embargo, romperlas no cuesta el mismo esfuerzo para todas. Teniendo un mapa con los esfuerzos cuantificados del 1 al 9 desean saber cuál es el mínimo esfuerzo que deben hacer para cumplir su objetivo.

Formalmente, podemos modelar este problema con grafos. Teniendo un grafo con peso se desea encontrar el árbol generador mínimo y devolver la sumatoria del peso de todos sus ejes.

2.2. Explicación de la solución

Nuestra entrada es un "mapa"que contiene " . ", "#" y números, es decir una matriz que en cada posición tiene alguno de esos simbolos. los puntos representan lugares caminables. Cada punto lo representamos con un nodo. los números son las paredes que se pueden romper y los numerales son las paredes irrompibles. Los nodos pueden conectarse a otros nodos que estén vertical o horizontalmente al mismo sí son continuos o tienen una pared rompible en el medio.

Definimos una clase eje que tiene como propiedad punteros a los nodos de cada extremo y un peso que representa cuanto cuesta romper la pared que existe entre ellos. Si dos salas son continuas, es decir no tienen un número o numeral en el medio el eje que las une tendrá peso cero. Sí entre dos nodos hay un número, ese será el peso del eje que los une. De haber un numeral, como esa pared es irrompible, no habrá un eje que una a esos nodos.

Ahora, teniendo el conjunto de todos los ejes y utilizamos el algoritmo de Kruskal para obtener el árbol generador mínimo, y retornar el valor total del mismo.
Para implementar Kruskal, utilizamos la implementación de dsu vista en clase. De esta forma agregamos dos modificaciones: guardamos un entero en el cual ir sumando el peso de cada arista elegida para devolver como resultado y utilizamos las funciones de DSU para poder identificar cuando no hay resultado y devolver -1. Presentaremos ese fragmento del algoritmo en el pseudocodigo ya que lo consideramos relevante.

2.2.1. Pseudocódigo

---

solucion(int n, vecor<ejes> arist)

1: ColaDePrioidad<ejes> aristas ▷ $\mathcal{O}(1)$
2: Mientras aristas ! = $\emptyset$ ▷ $\mathcal{O}(m)$
3: Tomar cada eje de arist e insertarlo en aristas ▷ $\mathcal{O}(log(m))$ esta es la complejidad que toma en c++ el ordenamiento.
4: finMientras
5: int res= Kruskal(aristas,n) ▷ $\mathcal{O}(mlog(n)+n)$
6: devolver res ▷ $\mathcal{O}(1)$

Complejidad: $O(mlog(m)) + O(mlog(n)+n) = O(m(log(m)+log(n))+n)$

Fragmento Kruskal()

```
1:  j ← 1, b ← true, res←1, nod ← el padre del nodo 1
2:  Mientras j<n y b O(n)
3:    Si el padre del nodo j es distinto a nod                    ▷ O(1)
4:      b ← false                                                 ▷ O(1)
5:      res ← -1                                                  ▷ O(1)
6:    finSi
7:    j ← j+1                                                     ▷ O(1)
8:  finMientras
9:  devovler res;                                                 ▷ O(1)
```

Complejidad: $O(n)$ esto le aporta el "$+n$" a la complejidad del Kruskal

2.2.2. Demostración de Correctitud

Este problema lo modelamos con grafos de modo tal que cada habitación caminable es un nodo conectado a otros mediante sus ejes. Los mismos, de haber una pared en el medio tendrán peso igual al costo de destruirla y, de no haber pared, tendrán peso cero. Si entre dos nodos hay una pared indestructible esos nodos no podrán conectarse, es decir no existe una arista entre ellos.
Un arbol generador mínimo de un grafo es un subgrafo que contiene todos los nodos del mismo y es un árbol que cumple que la sumatoria de los pesos de sus ejes es la menor posible. Es decir, no existe otro arbol generador tal que la sumatoria del peso de sus ejes sea mayor que la del AGM. También podemos garantizar que es mejor que cualquier subgrafo no arbol, ya que implicaría que tiene circuitos, y esto siempre agregaría peso de más ya que habría aristas de más.
En este problema nos piden que demos el esfuerzo minimo que nos cuesta acceder a todas las habitaciones. Por como modelamos el problema y las definiciones previas, esto es equivalente a encontrar el conjunto de aristas que conectan todos los nodos tal que minimzen la sumatoria de sus pesos. Como los ejes que conectan nuestros nodos continuos tienen peso cero, entonces no gregan peso a nuestro resultado. Es decir, que el resultado será el mismo que si las consideraramos como una sola sala. Entonces podemos ver que lo que nos están pidiendo es un AGM. Si el problema tiene solución, para encontrar el AGM del grafo utilizamos el algoritmo Kruskal. La demostración de correctitud de este algoritmo puede encontrase en el paper de J.B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society, Volume 7, pp. 48-50, 1956".
En caso de que el problema no tenga solución utilizamos las funciones de dsu para devolver -1. Basicamente, si no existe un AGM posible es porque el grafo no es conexo. En este caso exisitrán un par de nodos que al finalizar el kruskal no tendrán el mismo nodo padre.

2.2.3. Demostración de Complejidad

Como podemos apreciar en el pseudocódigo nuestro algoritmo tiene complejidad $\mathcal{O}(m(log(m) + log(n)) + n)$ donde m es la cantidad de ejes y n la cantidad de nodos de nuestro grafo. La complejidad pedida para este algoritmo es $\mathcal{O}(FClog(FC))$ donde F es la cantidad de filas y C la cantidad de columnas del mapa.

Primero probaremos que podemos acotar la cantidad de ejes por la cantidad de nodos:
Sabemos que
$1/2 \sum_1^n d(v_i) = m$.
Sin embargo podemos acotar $d(v_i) \leq 4$ ya que como mucho los nodos del medio del mapa pueden

tener 4 ejes mientras que todos los que estén en los bordes podrán tener como mucho 2.
Luego $1/2 \sum_1^n d(v_i) \leq 1/2 \sum_1^n 4 = 2n$.
Entonces $m \leq 2n$ y $\mathcal{O}(m) \leq \mathcal{O}(2n)\mathcal{O}(n)$.
Ahora veremos que la complejidad de nuestro algoritmo es la pedida:
Por lo anterior tenemos que $\mathcal{O}(m(log(m) + log(n)) + n) = \mathcal{O}(n(2log(n) + 1)) = \mathcal{O}(nlog(n))$.
La cantidad de nodos que tengamos depende directamente de F y C. la máxima cantidad de nodos que podemos tener es F-1*C-1 (la matriz sin paredes descartando los bordes)
Luego $\mathcal{O}(nlog(n)) = \mathcal{O}(FClog(FC))$

2.3. Experimentación

La complejidad requerida para este algoritmo es $\mathcal{O}(FClog(FC))$ y en la sección anterior probamos que esto es equivalente a $\mathcal{O}(nlog(n))$. Realizamos una serie de experimentos para respaldar empíricamente este hecho.

Los mismos consisten principalmente en variar las dimensiones del mapa, es decir, modificar F y C. Esperamos que estas variaciones afecten la cantidad de nodos, aumentando o disminuyendo la complejidad a corde de $\mathcal{O}(FClog(FC))$ manteniendo la misma tendencia.

Creemos que nuestro algoritmo será un poco más rápido a la cota pedida ya que la cantidad de nodos es un poco menor a FC.

Nuestro algoritmo devuelve -1 cuando se tiene más de una componenete conexa, es decir cuando no se pueden recorrer todas las salas. Decimdimos hacer un test para ver que ocurría en estos casos. al disminuir las componentes conexas tambien disminuimos la cantidad de nodos, por esta razón suponemos que debe tardar menos. si no las disminuyeramos creemos que el tiempo sería el mismo. Cada instancia de los experimentos fue iteradada 3000 veces, y se calculó el promedio.

2.3.1. Resultados y análisis

En nuestro primer experimento fuimos aumentando la dimensión de nuestro mapa, con F=C. lo corrimos para F=10 hasta F=250 aumentando de diez en diez.

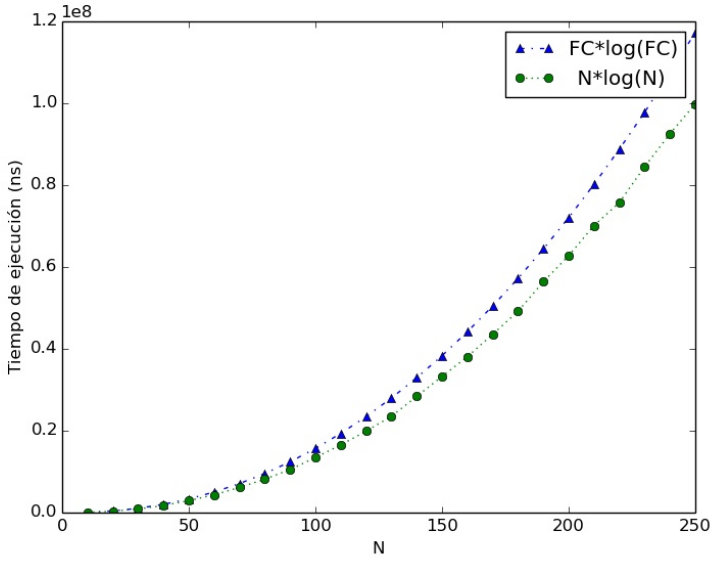
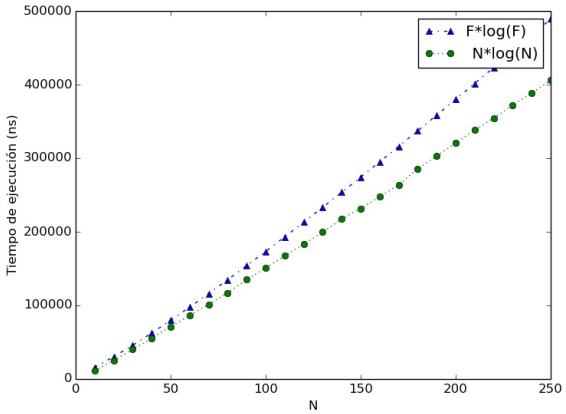


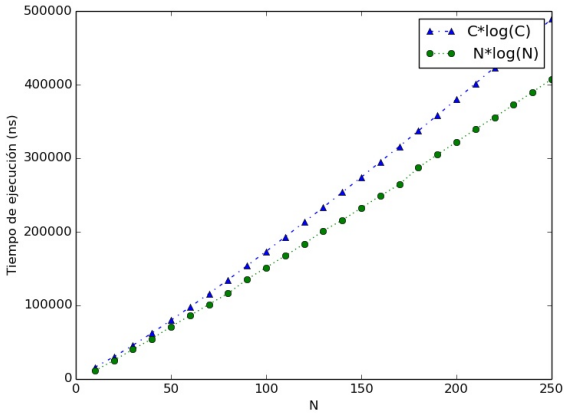
Figura 7: Aumento en la dimensión sel grafo

Podemos observar que el algoritmo tiene la misma tendencia logarítmica que nuestra cota teórica. También podemos ver que es mejor a medida que aumenta la cantidad de nodos (o filas y columnas).

En este experimento mantuvimos, o bien constante el número de filas, para aumentar de 10 en 10 las columnas hasta 250, o bien el número de columnas para aumentar las filas.



(a) Aumento en filas del mapa.



(b) Aumento en columnas del mapa.

Podemos ver que estos gráficos son practicamente iguales. con esto podemos mostrar que el algoritmo no depende en realidad de la cantidad de filas o columnas sino de los nodos. La razón por la cual nuestros gáficos son iguales es que mantienen la misma cantidad de nodos. la única diferencia en nustros mapas sería que uno es vertical y otro horizontal pero eso no afecta la cantidad de nodos.

Para este experimento creamos dos matrices manteniendo en una constante C=4 (habiendo entonces dos columnas de nodos) y en la otra C=6(habiendo 4 columnas de nodos). Fuimos aumentando las filas según como correspondiera en cada caso para mantener siempre la misma cantidad de nodos en ambos grafos.

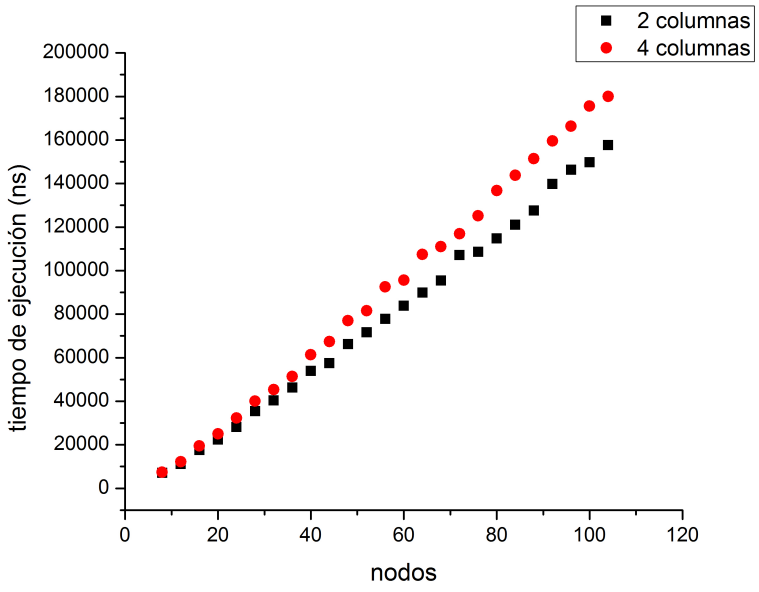


Figura 9: misma cantidad de nodos en mapas con distinta distribución

Podemos ver que el caso de 4 columnas tiene un tiempo de ejecución mayor. Esto se debe a que el algoritmo Kruskal tiene complejidad $\mathcal{O}(mlog(n))$ y aunque en nuestro algoritmo podamos acotar la cantidad de ejes por la cantidad de nodos, esta diferencia sigue siendo perceptible como podemos observar en la figura. El grafo con 4 columnas tiene más ejes que el grafo con 2 puesto que los nodos del medio contienen cuatro ejes mientras que en el de 2 columnas la máxima cantidad de ejes que se puede aspirar a tener es 3 en los nodos que no son vértices.

En este experimentos tomamos un mapa con F=C=31 y fuimos aumentando las componentes conexas, agregando alternadamente una columna de "#" en el mapa.

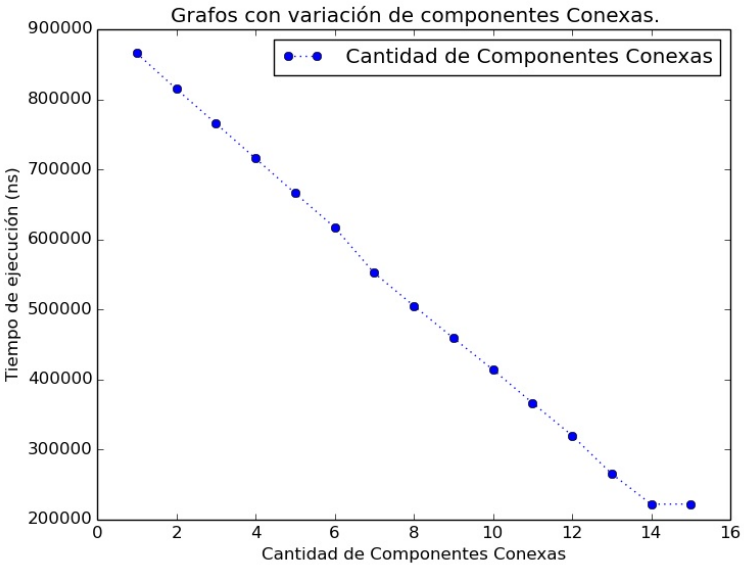


Figura 10: Aumento de las componentes conexas del grafo

Como podemos observar en esta figura el tiempo de ejecución disminuye a medida que aumentan las componentes conexas. Creemos que esto se debe principalmente a la disminusión de la cantidad de nodos.

3. Problema 3: Escapando

3.1. Introducción

En este problema, los exploradores se encuentran en un dilema, luego de romper varias paredes la fortaleza se esta derrumbando. Por suerte ellos se encuentran en una habitación que tiene varios carritos y un mapa que les indica que estaciones estan conectadas y cuanto tardan en llegar de estación a estación. Lo que quieren es la forma más rápida de llegar desde el lugar en donde estan hasta la salida, que sería la última estación.

Formalmente, tenemos un digrafo rotulado, con peso en los ejes, y cada nodo esta identificado por un número desde el uno hasta la cantidad de nodos. Nuestro objetivo es encontrar el camino mínimo, devolviendo el tiempo y su conjunto de nodos.

3.2. Explicación de la solución

En esta sección explicaremos por que el problema dado se puede adapatar al algoritmo de camino mínimo de Dijkstra. La precondición que el algoritmo pide es que no tenga ejes negativos. Como el peso de los ejes esta definido como el tiempo que se tarda de llegar del nodo de origen al nodo de llegada podemos asegurarnos que nunca vamos a tener una entrada, que nos importe, que tenga un eje con peso negativo.

3.2.1. Pseudocódigo

Solución(input: Arreglo[(Origen, Destino, Tiempo)])

```
1: LecturaDatos(input, matAdy)  Parte 1
2: valor ← CaminoMinimo(matAdy, estaciones)  Parte 2 y Parte 3
3: res ← valor ∧ estaciones
```

LecturaDatos(input: Arreglo[(Origen, Destino, Tiempo)], matAdy: Matriz(TiempoM, OrigenM))

```
1: Origen, Destino, Tiempo, TiempoM y OrigenM son Nat, simplifica la lectura
2: Incializa matAdy con (-1, -1);
3: ∀ x ∈ input
4:    aux ←(Tiempo(x), Origen(x))
5:    tiempoaux ← TiempoM(matAdy[Origen(x)][Destino(x)])
6:       if tiempoaux ≠ -1 ∨ tiempoaux ≥ Tiempo(x)
7:          matAdy[Origen(x)][Destino(x)] ← Tiempo(x)
8:       endif
```

Como no se sabe mucho de que input acepta el algoritmo, es decir, puede perfectamente un digrafo decidimos que era importante aclarar con el pseudocodigo de la lectura de datos como creamos la

matriz de adyacencia. Como es un digrafo la convencion que tomaremos es que dada una posición M_{ij} i es el origen y j es el destino. Luego en vez de solo representar el peso del eje tambien tenemos martcado cual es el predecesor, se inicializa los valores con valores que nunca vamos a tomar para saber si ese eje pertenece al grafo o no.

CaminoMinimo(MatrizAdy: Matriz(Nat, Nat) , estaciones: vector<Nat>)

```
1:  n ← tamaño(MatrizAdy)  Comienzo de Parte 2
2:  NodosSeguros ← 1
3:  nodosNoSeguros  conjunto {2, ..., n}
4:  mientras NodosSeguros ≠ G
5:     nodomin ← buscarMin(nodos, MatAdy[1])
6:     nodosNoSeguros - {nodomin}
7:     NodosSeguros ∪ {nodomin}
8:     ∀ e ∈ nodosNoSeguros ∧ [nodomin, e] ∈ X
9:        longi ← π1(matAdy[1][e])
10:       longmin ← π1(matAdy[1][nodomin])
11:       longimin ← π1(matAdy[nodomin][pos])
12:       if longi ≥ longmin +longimin
13:          matAdy[1][e] ← (longpmin + longimin, nodomin)
14:       endif
15: tiempo ← π1(MatAdy[1][n])  Fin de Parte 2 Comienzo de Parte 3
16: pred ← n
17: mientras pred ≠ 1  ∧  pred ≠ 0 (cuando no existe camino)
18:    estaciones ∪ {pred}
19:    pred ← π2(matAdy[1][pred])
20: res ← tiempo
```

CaminoMinimo en la parte 2 es dijkstra y en la parte 3 es recorrer la matriz para saber cuales son las estaciones que recorre el camino mínimo.

3.2.2. Demostración de Correctitud

Como podemos apreciar el pseudocódigo es el algoritmo Dijsktra, entonces su correctitud se desprende de la demostración de correctitud de dijkstra que se puede encontrar en varios libros de algoritmos, en nuestro caso vamos a referenciar al libro titulado "Introduction to Algorithms, Second Edition"de Thomas H Cormen, Charles E. Leiserson, entre otros. La demostracion se encuentra en el capítulo 24, subsección 3 bajo el título "Theorem 24.6: (Correctness of Dijkstra's algorithm)".

3.2.3. Demostración de Complejidad

Si analizamos con atención el pseudocódigo, tenemos tres secciones que se pueden analizar por separado y sumando sus complejidades obtendremos la complejidad total del algoritmo. La primer parte y la segunda parte combinadas son Dijkstra, la primera es la creación de la matriz y la segunda son los cálculos, la tercera parte es poner la información del camino mínimo. En los próximos párrafos nos vamos a referir a la cantidad de nodos en el gráfico como N.

Parte 1 La primera parte a analizar es la creación de la matriz, al ser una matriz de adyacencia, la cantidad de filas es N y la cantidad de columnas es N, actualizar todos los valores es recorrer toda la matriz haciendo que la complejidad sea $\Theta(N^2)$

Parte 2 En esta sección se encuentran dos ciclos anidados, podemos observar que el ciclo exterior hace N iteraciones ya que termina cuando el conjunto de nodos del grafo tiene el mismo cardinal que el conjunto de "nodosSeguros" y este último aumenta en uno por cada iteración. Dentro del ciclo principal tenemos dos operaciones que debemos tener en cuenta; sacar el nodo de la lista de "nodosNoSeguros" y el ciclo interno. Sacar un nodo de la lista nos va a costar encontrar el nodo y luego eliminarlo. Por la estructura que utlizamos eliminarlo no nos aporta complejidad, pero encontrar el nodo es una búsqueda lineal, es decir $\mathcal{O}(N)$. La última parte que nos falta analizar para poder determinar la complejidad de los ciclos anidados es el ciclo interno. Cada iteración recorre N posiciones de la matriz, aquellas que podrían ser un eje válido, aunque hace distintas cosas dependiendo de si es un eje válido o no, el interior del ciclo aporta una complejidad constante. Reuniendo toda la información, el interior del ciclo externo nos aporta una complejidad $\mathcal{O}(N)$ e itera N veces, es decir que la complejidad de la segunda parte es $\mathcal{O}(N^2)$.

Parte 3 Nos encontramos un ciclo que lee los datos de la matriz y guarda en un conjunto los nodos que tenemos que atravezar para tener el camino mínimo. La cantidad máxima de iteraciones que hace este ciclo es N, el razonamiento detrás de esta afirmación es que los ejes no tienen pesos negativos, si existe un camino mínimo, este no va a tener ciclos ya que pasar por un ciclo solo aumentaría el peso total del camino, y un camino sin ciclos en un grafo con n nodos tiene a lo sumo n-1 ejes, ya que el camino puede llegar a pasar por todos los nodos. Podemos concluir que el ciclo hace n iteraciones en el peor caso, es decir que la tercer parte es $\mathcal{O}(N)$

Ahora que analizamos las tres partes que podían llegar a dar complejidad al algoritmo sabemos que la complejidad algoritmica de la primer parte, la segunda parte y la tercera respectivamente son $\Theta(N^2)$, $\mathcal{O}(N^2)$, $\mathcal{O}(N)$. Entonces como todas las complejidades pertenecen al orden de $\mathcal{O}(N^2)$ y esta clase de complejidad es la mìnima a la que todas pertenecen a la vez, la complejidad total es $\mathcal{O}(N^2)$.

3.3. Experimentación

La cota de complejidad de nuestro algoritmo es $\mathcal{O}(N^2)$. Es decir que depende de la cantidad de nodos en un grafo. En esta sección trataremos de respaldar esta cota mediante el análisis de los datos empíricos que obtuvimos a través del testeo de nuestros algoritmos.
Tenemos dos algoritmos, los dos una variación del mismo pseudocódigo, el algoritmo sin modificaciones, A1, que busca el camino mínimo con todos los nodos y el algoritmo, A2, que se interrumpe cuando encuentra el camino mínimo que estamos buscando.
Decidimos testear sobre los dos algoritmos ya que al asignar peso aleatorio a los ejes en la mayoría de los casos teníamos la intuición de que los gráficos podrían quedar bastante mal. Pensamos en hacer test que nos aseguraran que se recorrían todos los nodos, pero al final decidimos usar dos variaciones del mismo algoritmo. Esto nos va a ayudar a demostrar que el algoritmo que nosotros elejimos en el peor caso tiene una complejidad igual al que realmente nos va a probar la cota N^2 y en el mejor caso tiene una complejidad lineal.
Con este objetivo a lo largo de los tests modificamos los grafos para observar su comportamiento y poder sacar conclusiones sobre las elecciones algoritmicas que tomamos. En cada test los valores se logran al promediar un tres mil iteraciones sobre el mismo input, sobre que forma tiene el input se va a hablar más adelante.
Tambien vale aclarar que no tenemos muchas restricciones sobre los parámetros del input, esto quiere decir que nada asegura que del input no podamos obtener un pseudo-digrafo. La lectura de datos selecciona las aristas de menor peso y las deja en una matriz. Entonces la creación de la matriz no se toma en cuenta en la ejecución de cada experimento ya que no tenemos que tomar mediciones que incluyan lectura de archivos.
Cada instancia de los experimentos fue iteradada 3000 veces, y se calculó el promedio.

3.3.1. Resultados y análisis

En nuestro primer experimento corrimos el algoritmo con diferentes grafos Kn, donde el n empieza en 10, se incrementa de a 10 terminando en 250. Los pesos de los ejes se generan de forma aleatoria. Creamos estos parámetros para tener una primera impresión de como variaba, dependiendo solamente de los nodos, ya que los ejes dependen de la cantidad de nodos. Nuestra expectativa era que A1 tenga un comportamiento cuadrático y que A2 tenga un comportamiento errático, pero parecido a una función cuadrática, ya que no sabíamos como iban a afectar la interrupciones que introducimos en el algoritmo, esperábamos mejoras en algunas iteraciones.

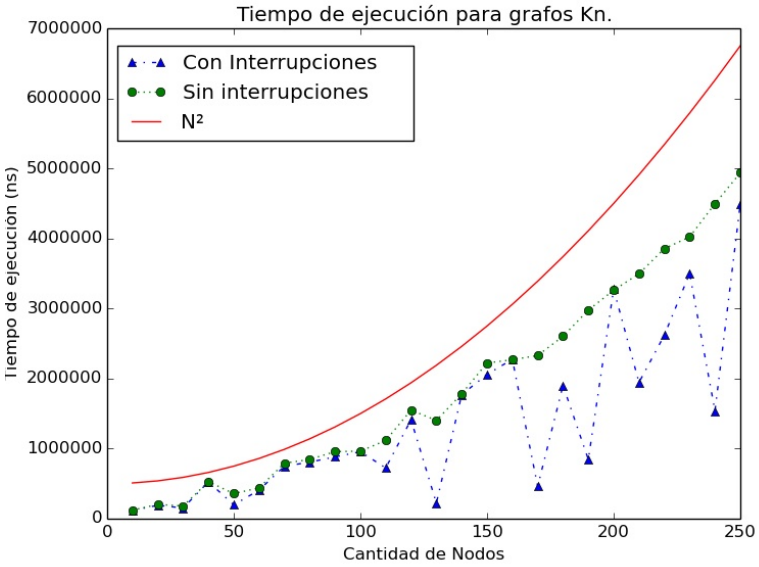


Figura 11

Como se puede ver en el gráfico, nuestras expectativas fueron cumplidas, A1 tiene un tendencia cuadrática y A2, aunque no muestra una tendencia clara, en muchos casos el tiempo de ejecución es menor que el de A1. Como explicamos anteriormente este comportamiento errático responde a que A2 termina cuando encuentra el camino mínimo para n y no sigue ejecutando para otros nodos, esto quiere decir que cuanto más cerca esta A1 de A2, el camino mínimo de n es uno de los últimos en computarse y análogamente si están lejos es que n es uno de los primeros en computarse.

Al encontar respaldo empírico sobre como nuestro algoritmo cumple con las complejidades teóricas, decidimos evaluar como respondía A1 y A2 sobre el mejor caso, este sería que el camino mínimo sea el eje que va desde 1 hasta n, ya que A2 corta en cuanto encuentra la solución para n. Nuestra complejidad, despues de leer el input es lineal. Lo que creamos es un test que nos crea grafos Kn y que tienen la particularidad de que el eje (1,n) pesa cero, haciendolo el camino mínimo. Debajo se encuentran dos gráficos que modelan este test.

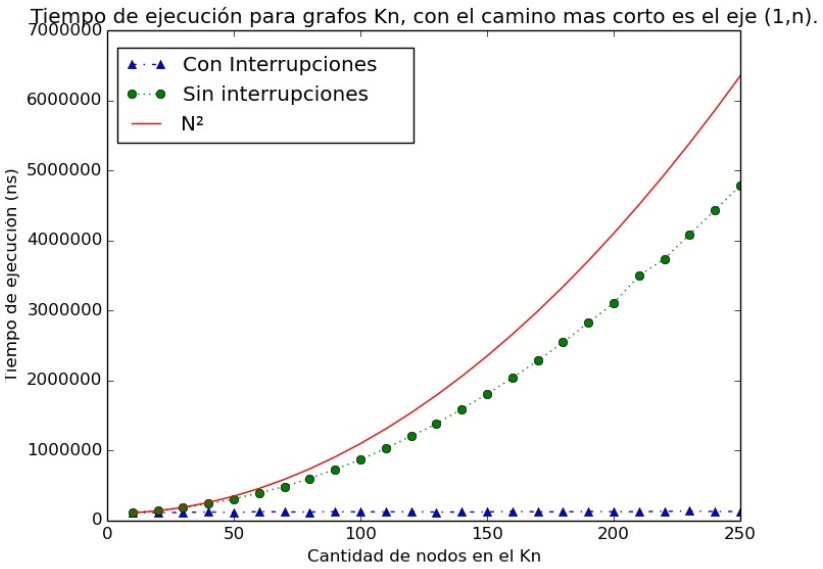


Figura 12

En este gráfico podemos ver como A1 mantiene su apariencia cuadratica, pero nos hace pensar que A2 tiene una forma constante. Esto se debe a la escala que estamos utilizando. El tiempo de ejecución de A2 es mucho menor que el de A1, al punto de que siempre se mantiene por debajo de 1000000 nanosegundos. Por eso decidimos mirar solo la línea de A2 para ver que a que función se asemejaba.

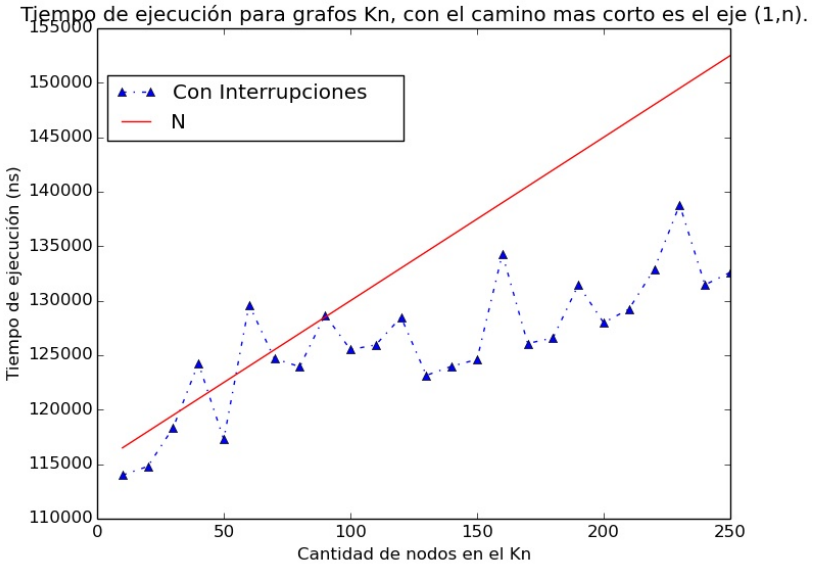


Figura 13

Como Podemos observar la apariencia constante de A2 era meramente una apariencia por las escalas que tenía la figura 10. Sin embargo no podemos percibir una tendencia lineal como esperabamos. Como parece estabilizarse un poco luego de los 100 nodos decidimos repetir el test con hasta 500 nodos para ver si efectivamente podíamos apreciar una tendencia un poco más lineal.

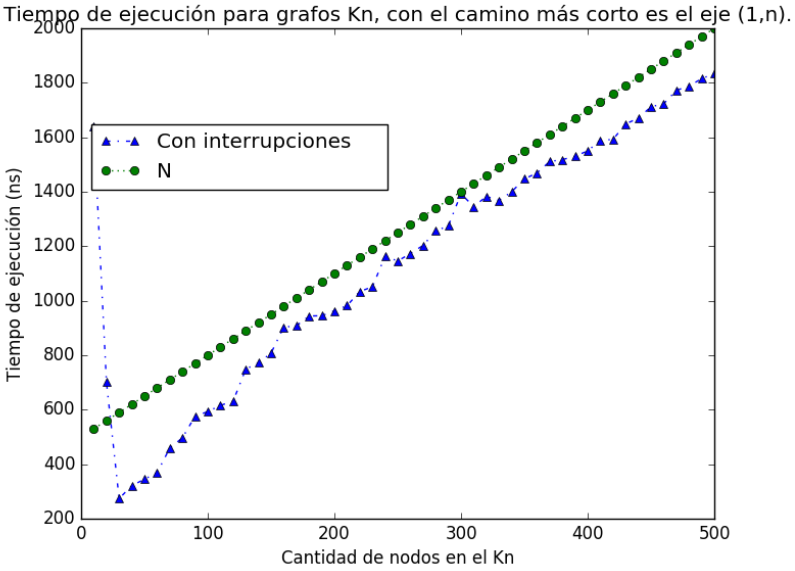


Figura 14

Efectivamente en este grafico se puede apreciar una semejanza con la funcion lineal. Las primeras instancias siguen siendo un poco conflictivas pero pasando los 100 nodos es mucho más estable.

Al finalizar estos experimentos, nos dimos cuenta que modificar la cantidad de nodos y que la cantidad de ejes está en función a la cantidad de nodos nos reducía el univierso de posibles grafos y probablemente habían dependencias en términos de complejidad que nosotros no cubríamos.
Entonces creamos este experimento, que crea grafos conexos con 200 ejes y varía los nodos desde 20 hasta 199 y los pesos están asignados aleatoriamente. Esperábamos un gráfico muy parecido a la figura 9.

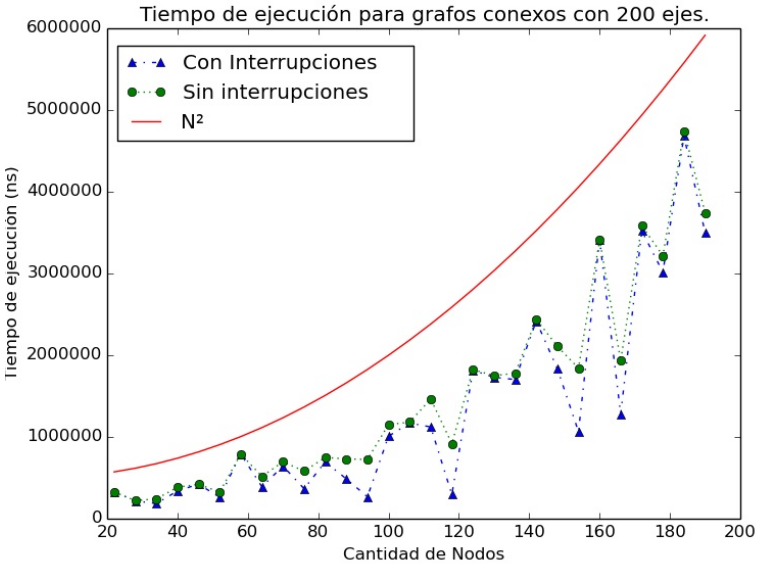


Figura 15

Aunque se pueden ver unas tendencias cuadráticas en el gráfico, y que está por debajo de la cota dada, también podemos ver que la figura 9 tiene los datos de A1 más regulares que en la figura 13.

Nuestras hipótesis es que, cuanto más denso es el grafo, más regular quedan las mediciones y cuanto menos denso las mediciones tienden a ser irregulares. Entonces decidimos experimentar sobre grafos menos densos y ver como quedaban las mediciones.
A fin de lo antes mencionado creamos una prueba, que dada una cierta cantidad de nodos, nos generaba cuatro grafos. Definimos el concepto D, como una símil densidad, donde la densidad está definida como la cantidad de ejes en un grafo dado, cuanto más denso el grafo más ejes tiene. D funciona sólo para grafos conexos, 0 es un árbol y 100 es el Kn.

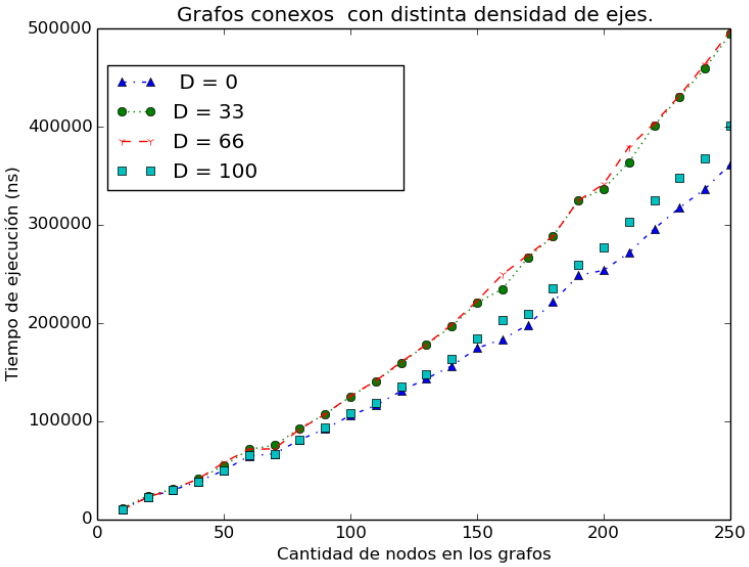


Figura 16

Como se puede observar en el gráfico no se respalda nuestra hipótesis, y nos genera una incertidumbre aún mayor, al encontrar la línea de Arboles muy por encima de las otras tres, que no era la idea intuitiva que nosotros teníamos, donde los grafos menos densos iban a estar acotados por grafos más densos. Aun así el próximo gráfico sin los arboles avala esta idea, y podríamos pensar que el caso de los arboles es una excepción a la regla.

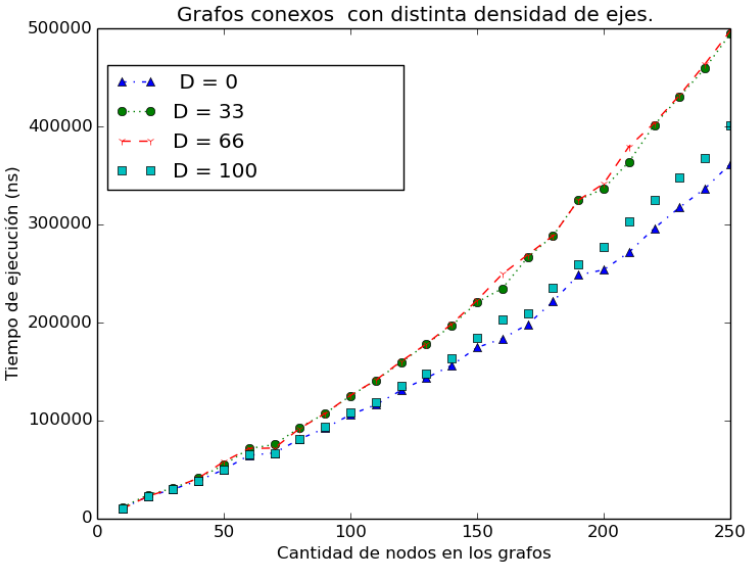


Figura 17

Luego de demostrar que la complejidad Teórica es respaldada por la experimentación, entramos en una series de pruebas que se generaron por la falta de entendimiento de nuestros gráficos y la busqueda de una hipótesis que satisfaga al lector. Lamentablemente no encontramos una, solo encontramos más preguntas sin resolver, que para no agobiar al lector dejamos como meras incógnitas para encarar en un futuro.¿Por qué los árboles tardan una cantidad significante más de tiempo que un grafo fuertemente conexo?¿Cómo se explica la variación de tiempos cuando la cantidad de ejes es la misma pero los nodos aumentan? Acaso fue que las 3000 iteraciones dieron unas mediciones poco estándares o hay una razón por la cual un grafo con 124 nodos tenga el mismo tiempo de ejecución que uno de 170, cuando estos tendrían que ser muy diferentes.