Algoritmos y Estructuras de Datos III

Primer cuatrimestre 2017

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

TP Nº 2 - Grafos

Camino Minimo, Arboles

Integrante	LU	Correo electrónico
Javier Petri	306/15	javierpetri2012@gmail.com
Jonathan Scherman	152/15	jonischerman@gmail.com
Lucas Adriel Figarola	953/13	lukas12_alfa56@hotmail.com
Ruben Adrian Castiglione	818/15	adriancastiglione@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

Delivery Optimo

Introducción al problema

En este problema nos situaremos en una provincia de Optilandia, donde las ciudades están conectadas por rutas bidireccionales. Algunas de estas rutas (en general las que representan caminos más cortos y/o directos entre las ciudades) han sido catalogadas con la categoría de premium, indicando que se encuentran en mejor estado y que reciben mejor mantenimiento que el resto. Para reducir los costos del mismo existen regulaciones que impiden que una empresa de transportes utilice más de k rutas premium en un mismo envío. Transportex es una de estas empresas y ha solicitado nuestros servicios para encontrar el camino de mínimo costo entre 2 ciudades (una de origen, otra de destino) cumpliendo con estas restricciones, siempre que esto sea posible.

Escribiremos un algoritmo que reciba los datos de las ciudades, las rutas y el k permitido y encuentre una solución si es que existe. Las ciudades pueden interpretarse como los nodos de un grafo, y las rutas que las conectan como las aristas del mismo, por lo que interpretaremos la entrada como un grafo; también se sabe que siempre hay una forma de llegar de una ciudad a otra (independientemente del uso de rutas premium), por lo que este grafo será conexo. Llamaremos n a la cantidad de nodos del grafo y m al número de sus aristas.

A partir del mismo construiremos un digrafo en el que los nodos serán de la forma $\langle c, p\rangle$, representando que se puede llegar a la ciudad c utilizando a lo sumo p rutas premium. Así, 2 nodos estarán conectados sii se puede pasar de un estado a otro, formalmente, si c_1 y c_2 están conectadas en el grafo de entrada, $\langle c_1, p\rangle$ y $\langle c_2, p\rangle$ estarán conectadas si las une una ruta normal, y $\langle c_1, p\rangle$ y $\langle c_1, p+1\rangle$ estarán conectadas si las une una ruta premium. También conectaremos $\langle c_1, p_1\rangle$ y $\langle c_1, p_2\rangle$ con $p_2 > p_1$, ya que si se ha llegado a una ciudad usando hasta cierta cantidad de rutas premium, el mismo camino sirve si la cantidad de rutas premium disponibles aumenta.

Se construye un digrafo y no un grafo ya que si c_1 y c_2 están conectadas por una ruta premium, tiene sentido poder avanzar desde $\langle c_1, p\rangle$ hacia $\langle c_2, p+1\rangle$ pero no al revés. Este digrafo tiene $n(k+1)$ nodos ya que se genera un nodo $\langle c_1, p\rangle$ para cada $c_1 \in [0; n)$ y para cada $p \in [0; k]$; en cuanto a las aristas, cada nodo $\langle c_1, p\rangle$ tendrá $k-p$ aristas conectándolo con los nodos que representan llegar a $c1$ usando más rutas premium, obteniendose $\frac{nk(k+1)}{2}$ aristas de este tipo; a su vez tendrá a lo sumo k aristas por cada arista incidente a c_1 en el grafo de entrada (este número cambiará en función de qué rutas sean premium y cuales no). En total, las aristas del digrafo son, a lo sumo, $\frac{nk(k+1)}{2} + mk < (n(k+1)+m)k$.

Finalmente, obtendremos el camino mínimo entre $\langle origen, 0\rangle$ y $\langle destino, k\rangle$, es decir, el camino mínimo desde el origen, sin haber usado rutas premium (en realidad no se han usado rutas de ningún tipo) y el destino, habiendo usado a lo sumo k rutas premium.

Representación

Representamos al grafo de entrada como una matriz simétrica $G \in \mathbb{Z}^{nxn}$, donde G_{ij} contiene la distancia entre i y j (0 si no están conectadas o si $i = j$). Para simplicidad, utilizamos la misma matriz para marcar si una ruta es premium o no de la siguiente manera: si d es la distancia (en valor absoluto) entre 2 ciudades, d será negativa si esas ciudades están unidas por una ruta premium, y positiva cuando las une una ruta normal. Notar que G es un grafo a pesar de esta representación, y que la ciudad número 0 se corresponde con aquella que estaba numerada como 1 en el archivo de entrada. El nuevo digrafo se representa por una lista de adyacencia ("lista de vecinos") donde cada vecino tiene el nodo correspondiente y la distancia. Dado que la información sobre si una ruta es premium o no está contenida en la conexión entre los nodos, las distancias se guardan positivas. Por lo tanto, un vecino es una tupla $\langle nodo, distancia\rangle$ (recordar que los nodos también son tuplas y que las distancias son ahora números naturales).

El algoritmo

El algoritmo que resuelve el problema consta de 3 partes: la primera es la lectura del archivo de entrada, la segunda parte construye el digrafo descrípto anteriormente y la tercera parte ejecuta el algoritmo de Dijkstra sobre el mismo. Dado que leer un archivo y construir una matriz es trivial y que el algoritmo de dijsktra puede ser consultado de diversas fuentes, mostramos a continuación el armado del digrafo:

```
1: function Problema1-armarDigrafo(grafoDeEntrada: int[n][n], n: unsigned, origen: nodo, destino: nodo, k)
2:     grafo = listaAdyacencia(n*(k+1), false)
3:     for i=0 to n*(k+1) do                          ▷ Generar el digrafo que representa el problema
4:         c_1 = ⌊i/(k+1)⌋
5:         p_1 = i % (k+1)
6:         for j=0 to n*(k+1) do
7:             c_2 = ⌊j/(k+1)⌋
8:             p_2 = j % (k+1)
```

```
9:             if c_1 == c_2 ∧ p_1 < p_2 then
10:                grafo.agregarVecino(i, ⟨⟨c_2, p_2⟩, 0⟩)
11:            else
12:                distancia = grafoDeEntrada[c_1][c_2]          ▷ distancia entre las ciudades c_1 y c_2
13:                if distancia == 0 then continuarConSiguienteIteracion end if    ▷ Si las ciudades no están conectadas no hay nada que hacer
14:                esRutaPremium = (distancia < 0)
15:                if ((p_2== (p_1 + 1) ∧ esRutaPremium) ∨ (p_1 == p_2 ∧ !esRutaPremium)) then
16:                    grafo.agregarVecino(i, ⟨⟨c_2, p_2⟩, |distancia|⟩)          ▷ Inserción en O(1)
17:                end if
18:            end if
19:        end for
20:    end for
21:    return caminoMinimoDijsktra(⟨0, 0⟩, ⟨n − 1, k⟩, grafo, k)
22: end function
```

Como solo nos interesa la distancia desde el origen hacia un nodo en particular, sería interesante poder cortar la ejecución de Dijsktra una vez que esa distancia ha sido hallada. Recordemos brevemente el algoritmo:

```
function AlgoDijsktra(Grafo G, nodo origen, nodo destino)
    Inicializar Π distancias y Q cola de prioridades
    while Q no está vacia do
        u ← min(Q)
        for each v vertice vecino de u do
            procesar v
        end for
    end while
    return Π(destino)
end function
```

Este algoritmo mantiene un invariante en el **while**: en el momento en el que un vértice u es extraido de la cola, se sabe que se ha obtenido su distancia mínima al origen. Aprovechando esta característica, nuestra implementación de Dijsktra deja de iterar cuando la cola no tiene más elementos, o cuando el nodo de destino ha sido extraido de la cola y por tanto se tiene su distancia mínima.

Complejidad

La complejidad de la solución estará dada por las complejidades de sus 3 partes.:

- La lectura del archivo se realiza en $O(m)$, dado que la entrada tiene $m+4$ líneas por cada instancia del problema y que los accesos a una matriz son $O(1)$ en C++.
- La generación del grafo es $O(n^2k^2)$ dado que consta de 2 ciclos anidados, cada uno con $n(k+1)$ iteraciones, y de operaciones en tiempo constante dentro de esos ciclos.
- El algoritmo de Dijsktra se implementa utilizando una cola de prioridad con actualización de prioridades en $O(1)$ e inicialización y búsqueda de mínimo en $O(t)$ (donde t es la cantidad de elementos de la cola, la cuál se corresponde con la cantidad de nodos del digráfo generado en la etapa anterior). Llamemos D al digrafo donde correrá este algoritmo, n' a la cantidad de sus nodos y m' a la cantidad de sus aristas. Se harán n' extracciones de mínimo y m' actualizaciones de prioridades, obteniendose una complejidad de $O((n')^2 + m')$. Dado que $n' = n(k+1)$ y $m' < (n(k+1)+m)k$, Disjkstra correrá en $O(n^2k^2 + m')$, donde $m' \in O(n^2k^2)$.

Juntandose todas las partes, el algoritmo correrá en $O(n^2k^2)$ (recordar que $m < n^2$ y que $m' \in O(n^2k^2)$.

Correctitud

TODO!!!

Experimentación

Para la experimentación realizamos un generador de instancias aleatorias para este problema, que genera grafos de n nodos, m aristas (de las cuales p son premium) y busca un camino mínimo de *origen* a *destino* usando k rutas premium ($n, m, p, origen, destino, k$ son parámetros para el generador) . El algoritmo del generador puede ser simplificado en 3 pasos:

- Generar un camino de n nodos de manera aleatoria
- Generar aristas entre nodos aleatorios (siempre y cuando no estuvieran conectados previamente ni sean el mismo nodo) hasta completar a m (m se establece entre $n-1$ y $\frac{n(n-1)}{2}$).
- Seleccionar p aristas aleatoriamente y catalogarlas como rutas premium.

Para generar el camino se selecciona un nodo aleatoriamente como inicial, y se lo marca como el nodo actual. Luego, sobre los nodos que no han sido seleccionados se selecciona un nuevo nodo aleatoriamente, se lo convierte en vecino del nodo actual y se lo marca como el nuevo nodo actual. Este proceso se repite hasta que no queden nodos que no hayan sido seleccionados. Para el segundo y tercer paso simplemente se seleccionan nodos aleatorios y se los une o marca su ruta como premium si las condiciones están dadas; en ambos casos se descartan aquellos nodos seleccionados y que no cumplan las condiciones para asegurar que el algoritmo termine. En cuanto a las condiciones, se unen nodos distintos que no hayan sido unidos previamente y se marca como premium el camino entre dos nodos sii estos están unidos y su ruta no es premium. Para la selección aleatoria se utiliza la función RAND de C++.

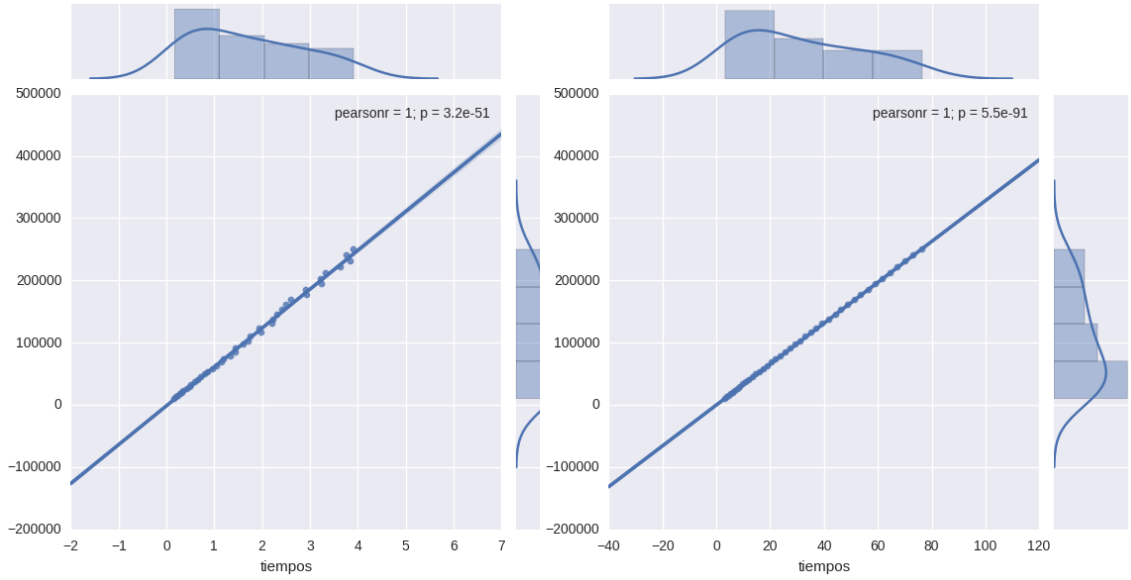
A continuación se harán experimentaciones para medir complejidades y para analizar casos buenos y malos para esta solución (en términos de complejidad)

Mediciones de complejidad

El objetivo de esta experimetación es el de comprobar que las complejidades teóricas se cumplan en la implementación. Dado que la misma depende de las variables n, k y $m'n$, se realizarán experimentaciones para medir cada una de ellas por separado, fijando las otras 2. Recordemos que la complejidad teórica analizada es de $O(n^2k^2 + m')$, con $m' \in O(n^2k^2)$.

Complejidad sobre n

Para este primer experimento k y m' serán fijadas en ?? y ?? respectivamente, mientras que n aumentará de 100 hasta 500 dando saltos de a 10. Dicho de otro modo, se probarán instancias de tamaño $n = 100, 110, 120, ..., 500$, midiendo sus tiempos de ejecución. Para cada caso se harán 5 repeticiones y se tomará la mediana. Se espera que el crecimiento sea cuadrático, por lo que una vez obtenidos los resultados se calculará el coeficiente de pearson junto con las funciones n, n^2, n^3 y n^4 para medir la correlación de estas con los tiempos de corrida. Dado que todos son polinomios la correlación debería ser alta en todos los casos, pero n^2 debería tener la mayor correlación de acuerdo al análisis teórico.



Complejidad sobre k

En este caso se fijarán n y m' en ?? y ?? mientras que k aumentará desde 100 hasta 500 dando saltos de a 10. Nuevamente se miden tiempos de ejecución, se realizan 5 repeticiones para caso (tomando la mediana de las 5), y se mide la correlación con n, n^2, n^3 y n^4 mediante el coeficiente de pearson. También en este caso se espera que el crecimiento sea cuadrático.

Complejidad sobre m'

Se fijarán n y k en ?? y ?? y m' se moverá entre 1000 y 4950 dando saltos de a 50. Nuevamente se realizarán 5 repeticiones y se tomará la mediana de los tiempos obtenidos. En cuanto al resultado esperado, recordemos que $m' \in O(n^2k^2)$ y que n y k permanecen fijos, por lo que es razonable esperar que los cambios en m' no aporten diferencias significativas en el tiempo de ejecución.

Subsidiando el transporte

Representación

Representaremos este problema con un grafo G que cumpla lo siguiente:

- G es un grafo orientado: Nos dicen que las rutas son de mano única por lo que representaremos estas rutas como aristas orientadas.
- $V = \{v \in \mathbb{N}_0 : v < n\}$: Esto es, los vértices de G serán números, en donde cada uno de ellos representa cada una de las n ciudades del problema.
- $E = \{(v_1, v_2) \in V^2 : \exists \text{ ruta que va de } v_1 \text{ a } v_2\}$: De esta forma, los ejes de G representarán las rutas del problema.
- $p : E \to \mathbb{Z}$: Representaremos los precios de los peajes como pesos en los ejes (si son positivos, serán la suma que se cobra al cliente. Si son negativos, la suma que se le paga). Para ello, la función p asignará enteros a cada uno de estos. Notar que nos dicen que hay un peaje por cada ruta por lo que $dom(p) = E$ ya que cada peaje debe tener un costo.
- $\forall v \in V, d_{out}(v) \geq 1$ (no existen ciudades aisladas)

Entonces, G será el modelo que representa a la ciudad de nuestro problema. Puntualmente, lo que queremos es encontrar el máximo s tq si consideramos $p'(e) = p(e) - s$ $(\forall e \in E)$ en lugar de p, entonces no se forman ciclos negativos, es decir, que la suma de los pesos de sus ejes no sea menor a 0.

Para construír G a partir de la entrada, lo que haremos será ir agregando eje por eje, que va a tomar $O(m)$. Representaremos así a G como una lista de incidencias.

Resolución

La idea para la resolución del problema va a ser la siguiente:

1. Unir un vértice *centinela* a cada uno de los nodos: En el siguiente punto vamos a necesitar usar el algoritmo de *Bellman-Ford* en G. Como no sabemos si G es fuertemente conexo o no, agregamos este centinela que alcance a todos los $v \in V$, para luego correr el algoritmo tomando al centinela como origen.
2. Variar subsidio $0 \leq s \leq C$ binariamente hasta encontrar el máximo:
    - Se va a ir probando con subsidios que van a variar como en una búsqueda binaria tradicional. La diferencia es que, en cada iteración, se va a fijar si tal subsidio genera algún ciclo negativo en lugar de verificar si el elemento actual es el buscado.
    - Para encontrar ciclos negativos, se utiliza el algoritmo de *Bellman-Ford*, tomando como origen al centinela agregado en el paso 1 que, como alcanza a todos los vértices de G, nos asegura que encuentra todos los ciclos negativos en caso que los hubiera.

De esta forma, una vez que tengamos maximizado el s del paso 2, sabremos que ese es el valor máximo para el cual no se generan ciclos negativos que, en terminos del problema, corresponde al máximo subsidio que se puede aplicar tal que no genere aprovechamientos, lo cual es la solución óptima al problema que pretendemos resolver.

Implementación

Para la implementación del algoritmo, vamos a usar la función *hayCicloNegativo?* que, mediante el algoritmo de *Bellman-Ford*, detecta si hay ciclos negativos en el grafo. Para ello, le pasamos como parámetro el grafo con sus dimensiones y el nodo origen, que en este caso será el centinela puesto que es

el único vértice para el cual podemos asegurar que está conectado a todos los nodos del grafo y así no se va a perder ningún ciclo negativo en caso que los haya.
Además, usaremos las función *aumentarSubsidio* que recibe las rutas y una constante por parámetro y, entonces, para cada ruta, le resta la constante a su peso. De forma similar tendremos la función *disminuírSubsidio* que, en lugar de restar la constante para cada ruta, la suma.

```
1:  function MAXIMOSUBSIDIO(rutas, n, m, C)
2:      centinela ← n
3:      for i=1 to n do                          ▷ Unir centinela a todos los nodos
4:          e ← Eje(centinela, n, C)
5:          rutas.push(e)
6:          m++
7:      end for
8:      n++
9:      j ← C
10:     i ← 0
11:     s, sAnterior ← 0
12:     while i < j do                           ▷ Mover binariamente s
13:         s ← ⌊(j+i)/2⌋
14:         if s = sAnterior then return s end if   ▷ Evita ciclo infinito en caso borde
15:         aumentarSubsidio(rutas, s)
16:         if hayCicloNegativo?(rutas, n, m, centinela) then
17:             j ← s
18:         else
19:             i ← s
20:         end if
21:         disminuírSubsidio(rutas, s)
22:         sAnterior = s
23:     end while
24:     return s
25: end function
```

Complejidad

En el primer ciclo (L3-7) se ejecuta n veces operaciones que son $O(1)$ por lo que la complejidad de éste resulta $\Theta(n)O(1)$ que, en particular, es $O(n)O(1) = O(n)$

En el segundo ciclo (L12-23), estamos moviendo al s como en una búsqueda binaria, en donde $0 \leq s \leq C$. En este caso, a diferencia de la búsqueda binaria, se va a tomar $\Theta(log(C))$ (que es $O(log(c))$) en todos lo casos puesto que buscamos maximizar C hasta no poder más, a diferencia de la otra que termina cuando encuentra el elemento en un buen caso. Además, como mencionamos anteriormente, la función *hayCicloNegativo?* ejecuta el algoritmo de $Bellman - Ford$, que toma $O(nm)$, junto a las operaciones *aumentarSubsidio* y *disminuírSubsidio* que toman $O(m)$ puesto que recorren todas las aristas, junto a otras operaciones que toman $O(1)$. Por lo que la complejidad de este ciclo resulta $O(log(C)(nm + m)) = O(log(C)nm + log(C)m) = O(log(C)nm)$ dado que $nm \geq m (n \in \mathbb{N})$

Finalmente, falta sumar O(1) por algunas otras operaciones que se realizan como asignaciones, incrementaciones y demás, por lo que la complejidad final resulta entonces $O(m + log(c)nm)O(1) = O(nmlog(c))$ en todos los casos.

En cuanto a la complejidad espacial, el algoritmo utiliza la lista de ejes ($O(m)$) sumado la complejidad espacial que necesita *Bellman-Ford*, que es $O(n)$ (necesita una arreglo de distancias de cada nodo al origen). Entonces, la complejidad espacial final resulta $O(m + n)$

Correctitud

Sea s el subsidio máximo de G obtenido mediante el algoritmo. Veamos que:

(I) $0 \leq s \leq C$

(II) s es la solución óptima

dem:(i) Sea s' una solución del problema cualquiera. Por hipótesis del enunciado, sabemos que $d_{out}(v) \geq 1(\forall v \in V) \Rightarrow \exists P$ ciclo en G. Tenemos por un lado que $s' \geq 0$, ya que si no fuera así, s' no sería un subsidio sino un impuesto. Por otro lado, $s' \leq C$, porque sino $s' > C \Rightarrow p(v) \leq C < s'(\forall v \in P) \Rightarrow p(v) - s' < 0(\forall v \in P) \Rightarrow$ P es un ciclo negativo, que es absurdo, por lo que debe ser entonces que $s' \leq C$. Por lo tanto, debe ser que $0 \leq s' \leq C$. Ahora, en el algoritmo, s varía 'binariamente' entre exactamente ese rango, por lo que por correctitud de la busqueda binaria debe valer esta propiedad.

dem:(ii) Supongamos que existe otra solución s' tq $s' < s$. Debe ser entonces que se evaluó incorrectamente la solución $s = s'$: Esto es, se detecto un ciclo negativo cuando no lo había, lo cual sucede si el algoritmo de *Bellman-Ford* detecta tal cosa. Ahora, que s' haya sido invalidada implica que la respuesta de $Bellman - Ford$ sea incorrecta, pero dado que

- Se tomó como origen al *centinela* que agregamos y que alcanza a todos los nodos de G
- Agregar el centinela no agregó ningún ciclo (pues ningun vértice incide sobre centinela)

por correctitud de $Bellman - Ford$ sabemos que esto no puede suceder. Por lo cual, es absurdo que exista tal solución s', lo cual implica que s sea la óptima.

Experimentación

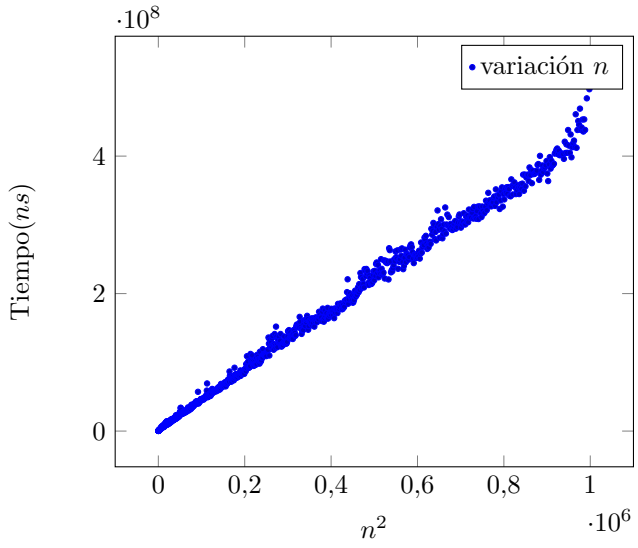
El objetivo de esta experimentación fue de concluír que la complejidad teórica calculada se condice con los tiempos de ejecución del algoritmo. Éste, según el análisis realizado, depende de tres variables: n, m y c. Por lo cual, la idea de la experimentación fue fijar de a pares y variar el tercero para determinar la influencia de esta variable en los tiempos a medida que aumenta. Para cada valor de la variable de cada experimento, se tomó el promedio del tiempo de ejecución de 5 instancias de test.

Para generar las instancias de test, se siguieron los siguientes pasos: dados $n, m, cotaSupC$

1. Para todo vértice v, se creó una ruta hacia otro vértice v' elegido de forma aleatoria (para satisfacer la hipótesis de que no haya ciudades aisladas). Para determinar su peso, se eligió también de forma aleatoria tal que el resultado sea menor o igual a $cotaSupC$
2. Mientras se tenga que seguir agregando rutas, seguir construyéndolas de forma aleatoria.

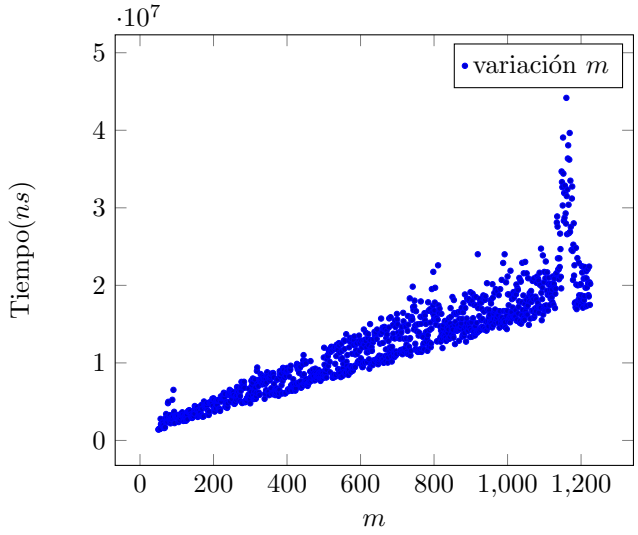
Los parámetros dependen del experimento realizado que se procederá a detallar en el experimento correspondiente.

Para el primer experimento, queremos ver como varía n a medida que crece. Para el experimento, se fijó $C = 5000$ y $m = n$ (la cantidad de rutas mínima para satisfacer la hipótesis). Teníamos que la complejidad teórica era era $O(nmlog(C)) \Rightarrow O(n^2log(C))$, por lo que se espera que los tiempos crezcan de forma cuadrática. Entonces, tomando al eje $x = nm = n^2$, se espera que la curva sea lineal. Veamos los resultados obtenidos:

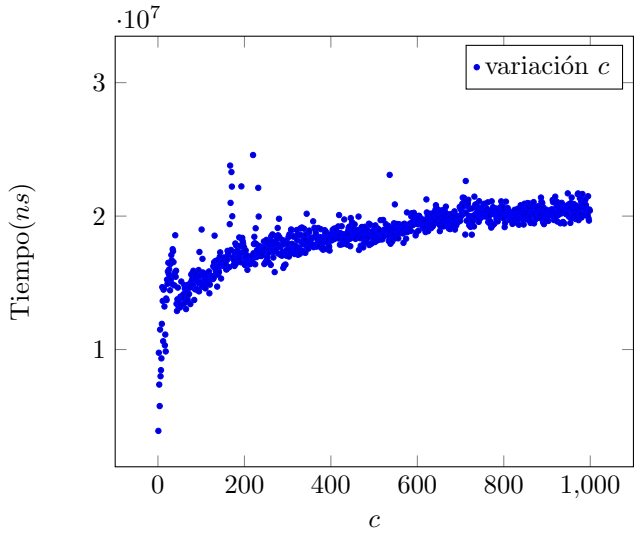


Efectivamente, se puede ver que los resultados obtenidos fueron los esperados.

Para el siguiente experimento, se quiere ver como varía m. Para ello, se fijó $C = 5000, n = 50$. Esperamos que este factor crezca de forma lineal acorde a la complejidad teórica. Los resultados son:



Observamos que, de nuevo, m varía linealmente como esperábamos. Veamos como varía C en nuestro último experimento. Fijamos $n = 200, m = 300$. Esperamos que tenga un comportamiento logarítmico. Los resultados:



Podemos concluír que C se comporta efectivamente de forma logarítmica. De todas formas, se esperaba un gráfico más regular para valores relativamente grandes dado que siempre se toma $log(C)$. En particular, se esperaba que todo valor $log(512) \leq x < log(1024)$ tome un tiempo similar.

Reconfiguracion de Rutas

Representación

Representaremos este problema con un grafo G_i que cumpla lo siguiente:

- G_i es un grafo simple: Dado que todas las rutas son doblemano, podemos abstraernos de las direcciones y pensarlos como si cada par de rutas fueran una única.
- $V = \{v \in \mathbb{N}_0 : v < n\}$: Esto es, los vértices de G_i serán números, en donde cada uno de ellos representa cada una de las n ciudades del problema.
- $E = \{(v_1, v_2) \in V^2 : \exists \text{ ruta que une } v_1 \text{ con } v_2\}$: De esta forma, los ejes de G_i representarán las rutas existentes del problema.
- $p : V^2 \to \mathbb{N}_0$: Representaremos como los precios de construcción o destrucción de las rutas como pesos en los ejes de G_i. Por lo cual, p asignará un costo (mayor o igual a 0) a cada uno de los ejes. Observar que $E^c \subseteq dom(p)$ dado que esta función también asigna costos a los ejes inexistentes de G_i. En definitiva, p asigna precios de destrucción a rutas que existen y de construcción a aquellas que no.

Entonces, G_i será el modelo que representa a las ciudades con sus rutas y los precios de construcción o destrucción correspondientes.

Resolución

La idea del algoritmo que usaremos para resolver este problema será la siguiente:

1. Encontrar $c_1, c_2, ..., c_k$ componentes conexas de G_i. No sabemos si G_i es conexo o no, pero sabemos que podemos tomar todas las componentes conexas C_i y entonces, para cada par de ciudades $v_1, v_2 \in V(C_i)$ habrá, por lo menos, un camino para llegar de una a la otra.
2. Destruír las rutas innecesarias con menor costo de destrucción. Esto lo haremos obteniendo los árboles generadores máximos A_i correspondientes a cada C_i, con $1 \leq i \leq k$. Podemos lograrlo invirtiendo el peso de los ejes y corriendo el algoritmo de $Kruskal$, dado que $codom(p) \geq 0 \Rightarrow$ los ejes son positivos (ó 0). Llamemos E_d a estas rutas destruídas, P_d al costo total de destruírlas.
3. Construír las rutas mas baratas entre cada par de componentes conexas. Para elegir que rutas deberán ser construídas, usaremos nuevamente el algoritmo de $Kruskal$, tomando E^c como conjunto de ejes a agregar y G_i como grafo base. Llamemos E_c a estas nuevas rutas, P_c al costo total de construírlas.

Así, si G_f es el grafo resultante, entonces $V(G_f) = V(G_i)$, $E(G_f) = E(G_i) - E_d \cup E_c$. Entonces, el resultado será $E(G_f)$ y el costo total $P = P_c + P_d$

Implementación

Para la implementación vamos a necesitar, en primer lugar, la estructura *Disjoint Set* para trabajar con las componentes conexas. Además, necesitaremos hacerle una serie de modificaciones:

- Para su estructura, necesitaremos saber cuáles son los ejes contenidos en cada componente conexa.
- En la función *union*, además de su funcionalidad tradicional, tendrá que unir los ejes de la componente con menor rank a la de mayor rank, que puede hacerse en $\Theta(1)$ puesto que se representan con listas enlazadas. Además, *makeSet* tendrá que inicializar la lista de ejes vacía correspondiente. Entonces, las complejidades de las funciones *makeSet, union, find* de esta estructura no se ven alteradas con estas. modificaciónes

- Se agrega la función *sets*, que devuelve la lista de componentes conexas distintas que contiene la estructura. Para ello, tendrá que recorrer todos los nodos y agregar (por referencia) las componentes conexas diferentes a las cuales estos pertenecen en una lista, cuyo costo será $\Theta(n)$.

De esta forma, para obtener las componentes conexas tendremos que, luego de inicializar las componentes conexas triviales usando *makeSet* para cada vértice, recorrer todas las aristas y, para cada una de ellas, llamar a la función *unify* entre los extremos de la misma. Así, las obtenemos utilizando la función *sets*.

Por otro lado, se usa la funcion *kruskal-AGMax* sobre cada componente conexa C_i, que toma $E(C_i)$ por referencia y genera un árbol generador máximo, devolviendo una tupla con el peso de todos los ejes destruídos y los ejes del árbol generado. Para ello, se usará el algoritmo de Kruskal pero invirtiendo los pesos de los ejes para realizar el ordenamiento inicial de ejes para que Kruskal ubique las aristas con mayor costo de destrucción al principio. Además, irá acumulando el peso de los ejes que se destruyen (o sea, aquellos ejes que Kruskal fue descartando). Por lo cual, estos cambios no alteran el objetivo del algoritmo y el costo temporal sumando junto al costo de invertir los ejes y crear una copia de ellos para devolver, que se resuelve en $O(2m(C_i))$. Luego el total es $O(m(C_i)\log(m(C_i))+2m(C_i)) = O(m(C_i)\log(m(C_i))$.

Por último, la función *kruskal-AGMin*, que toma una lista de ejes de entrada *rutasResultantes*, que corresponde a $m' = \bigcup_{i=1}^{k} E(A_i)$ (todas las rutas restantes después de la destrucción de rutas innecesarias), y las rutas que todavia no existen, $m^c{}_{Gi}$, y set ejecuta el algoritmo de Kruskal tomando como ejes de inserción a *rutasNoExistentes* y como grafo base a *rutasResultantes*. Notar que en esta implementación, ya tenemos una ciudad parcial con componentes no conectadas. Por lo cual, en lugar de construír el AGM entero (partiendo desde la lista *vacía*), comenzaremos tomando la lista *rutasResultantes* de esas componentes y se agregarán los ejes(rutas) con menor peso. Por lo cual, la idea del algoritmo es la misma, pero con la única diferencia de que se parte de una lista de aristas no vacía, contrariamente al algoritmo tradicional. Por lo tanto, seguirá valiendo que esos ejes que se agreguen serán minimos y entonces nos ayuda para resolver el paso 3 (construír las rutas necesarias). Entonces, la complejidad para lograr esto permanece y sabemos que es $O(m^c{}_{Gi} log(m^c{}_{Gi}))$

Input: n: Cantidad de ciudades (int)
rutasExistentes: Lista de ejes correspondientes a las rutas existentes
rutasNoExistentes: Lista de ejes correspondientes a las rutas que no existen.
Donde eje tiene tiene un origen, destino y peso.

```
1:  function RECONSTRUIRRUTAS(n, rutasExistentes, rutasNoExistentes)
2:      uds ← DisjointSetVacio
3:      for i=0 to n - 1 do                          ▷ Inicializar vertices como C.C.
4:          uds.makeSet(i)
5:      end for
6:      for each e ∈ rutasExistentes do              ▷ Paso 1: Separar componentes conexas
7:          uds.union(e)
8:      end for
9:      rutasResultantes ← ListaVacía
10:     costoTotal ← 0
11:     componentesConexas ← uds.sets()
12:     for each C_i ∈ componentesConexas do         ▷ Paso 2: Destruír rutas innecesarias
13:         <costo, rutas> ← kruskal-AGMax(C_i.ejes())
14:         rutasResultantes.unir(rutas)
15:         costoTotal ← costoTotal + costo
16:     end for
17:                                                  ▷ Paso 3: Construir rutas nuevas
18:     <costo, rutas> ← kruskal-AGMin(rutasResultantes, rutasNoExistentes)
19:     costoTotal ← costoTotal + costo
20:     return <rutas, costoTotal>
21: end function
```

Complejidad

Para inicializar el *Disjoint Set*, se puede ver que el ciclo itera siempre exactamente n veces y ejecuta la operación *makeSet*, que como dijimos antes, su complejidad no se ve afectada por los cambios por lo que sabemos que su complejidad es O(1). Por lo cual, este ciclo toma $O(n)$

Para el ciclo del paso 1, podemos pensar que en un peor caso las rutasExistentes van a ser todas las posibles, en cuyo caso el ciclo iteraría O(m) veces, ejecutando la operación *union* que, dado que la complejidad original permanece, toma O($\alpha(n)$) amortizado, en donde α es la inversa de la función de *Ackermann*. Luego, la complejidad del ciclo es $O(m\alpha(n)) \subseteq O(\frac{n(n-1)}{2}\alpha(n)) = O(n^2\alpha(n))$

En la línea 11, como se mencionó anteriormente, tomará Θ(n).

Para el ciclo del paso 2, sabemos que se va a recorrer exactamente k veces, donde k es la cantidad de componentes conexas distintas que hay. Por cada una de ellas, se va a realizar *kruskal-AGMax*, que toma O(m(C_i)log(m(C_i))). Además tendrá que hacer uníon de listas que puede hacerse en O(1), y sumas y asignaciones que también son O(1). Entonces, lo que sabemos es que la complejidad temporal de este ciclo será $O(\sum_{i=1}^{k} m(C_i)log(m(C_i)))$. Ahora, como $m(C_i) \leq m$, tenemos

$$\begin{aligned}
&\sum_{i=1}^{k} m(C_i)log(m(C_i)) \\
&\leq \sum_{i=1}^{k} m(C_i)log(m) \\
&= log(m)\sum_{i=1}^{k} m(C_i) \\
&= log(m)m \\
&\leq log(\frac{n(n-1)}{2})\frac{n(n-1)}{2} \\
&\leq log(n^2)n^2 \\
&= 2log(n)n^2
\end{aligned}$$

Por lo cual, la complejidad temporal de este ciclo es $O(n^2log(n))$.

Por último, vimos antes que la complejidad de la función *kruskal-AGMin* no cambia con respecto a la original. Por lo tanto, sabemos que esto tomará $O(|\text{rutasResultantes}|)$ para armar el *Disjoint set*, y $O(|\text{rutasNoExistentes}|log(|\text{rutasNoExistentes}|))$. Como rutasResultantes $\leq$ rutasExistentes y además $|\text{rutasExistentes}| + |\text{rutasNoExistentes}| = m$, vale que $|\text{rutasResultantes}|, |\text{rutasNoExistentes}| \leq m$. Por lo que

$$\begin{aligned}
&|\text{rutasResultantes}| + |\text{rutasNoExistentes}|log(|\text{rutasNoExistentes}|) \\
&\leq m + mlog(m)
\end{aligned}$$

Por lo cual, esto toma $O(m + mlog(m)) = O(mlog(m)) \subseteq o(n^2log(n))$ (por lo visto en el caso anterior)

Sumado a algunas otras operaciones O(1), la complejidad temporal final del algoritmo en un peor caso es $O(n\alpha(n)+n^2\alpha(n)+n+2n^2log(n)) = O(n^2log(n))$ en un peor caso, que cumple con la complejidad pedida.

En cuanto a complejidad espacial, se necesitará O(n+m) para el *Disjoint Set* y O(m) para las listas de rutas existentes, no existentes y resultantes. Por lo cual, resulta $O(n + m) \subseteq O(n + \frac{n(n-1)}{2}) = O(n^2)$,

Correctitud

Sea G_f el grafo resultante, P_{Gf}[1] la inversión efectuada. Para probar la correctitud de este algoritmo con respecto a lo pedido, queremos ver que se verifican:

(I) G_f es un árbol: Si esto se cumple, entonces por definición de árbol, G_f es conexo y no tiene ciclos. Como es conexo, $\forall v1, v2 \in V(G_f), v1$ y $v2$ estan conectadas por algún camino, lo cual implica que todas las ciudades estan conectadas Además, como no hay ciclos, este camino es único.

(II) P_{Gf} es mínimo.

dem(i): Por correctitud de Kruskal, sabemos que el resultado de ejecutar dicho algoritmo sobre las componentes conexas $C_1, C_2, ..., C_k$ serán k árboles distintos $A_1, A_2, ..., A_k$ (paso 2). Por otro lado, sabemos que Kruskal solo agrega aristas que no generan ciclos, por lo que es imposible que después de terminar el paso 3 haya alguno. Además, en este paso se contemplan todas las aristas inexistentes por lo que tampoco puede pasar que algún par de componentes conexas de G_i queden disconexas entre sí. Luego, G_f es conexo y no tiene ciclos $\Rightarrow G_f$ es un árbol.

dem(ii): Supongamos que P_{Gf} no es mínimo. Entonces, $\exists G'_f$ árbol tq $P_{G'f} < P_{Gf}$ ($\Rightarrow G'_f \neq G_f$).
Veamos que:

- La destrucción de rutas de G_f fue óptima:
  Sea C_i una componente conexa de G_i. Si tomamos $A_i \subset G_f, A'_i \subset G'_f$ los subgrafos que corresponden a C_i en estas soluciones (más precisamente, $A_i = G_f|_{V(C_i)}, A'_i = G'_f|_{V(C_i)}$[2]), por correctitud de Kruskal, sabemos que $P_{Ai} \leq P_{A'i}$ (pues A_i va a a destruír las rutas más baratas).
  Pero esto vale para todas las componentes conexas de G_i, por lo tanto debe ser que

  $$P_A \leq P'_A$$

  donde A, A' son grafos tales que A es la union de todos los A_i y A' es la union de todos los A'_i.

- La construcción de rutas de G_f fue óptima:

  (a) Se construyeron las rutas necesarias más baratas:
  En primer lugar, notemos nunca va a ser necesario construír en una componente conexa pues si así fuera se generaría entonces un ciclo (en términos del problema, se crearían nuevos caminos y se aumentaría el costo de la solución). Por esto, solo tiene sentido construír rutas entre las componentes conexas del grafo inicial G_i. Para esto, podemos asegurar por correctitud de Kruskal, que se van a construír las rutas mas baratas entre todas las posibles y, además, esta construcción nunca va a generar ciclos, es decir, nunca se construirán rutas innecesarias para conectar estas componentes. Si consideramos los grafos $A^c = G_f - A, A'^c = G'_f - A'$, tenemos que entonces vale

  $$P_{A^C} \leq P_{A'^C}$$

  (b) No se construyó ninguna ruta innecesaria:
  De nuevo, es imposible que pase que A^c tenga alguna ruta innecesaria por correctitud de Kruskal. Por otro lado, en A, no se construyó ninguna ruta nueva, solo se destruyeron las que eran innecesarias.

Finalmente, si sumamos ambas desigualdades, llegamos a que

$$\begin{aligned} P_A + P_{A^C} &\leq P_{A'} + P_{A'^C} \\ &\Leftrightarrow P_{Gf} \leq P_{G'f} \end{aligned}$$

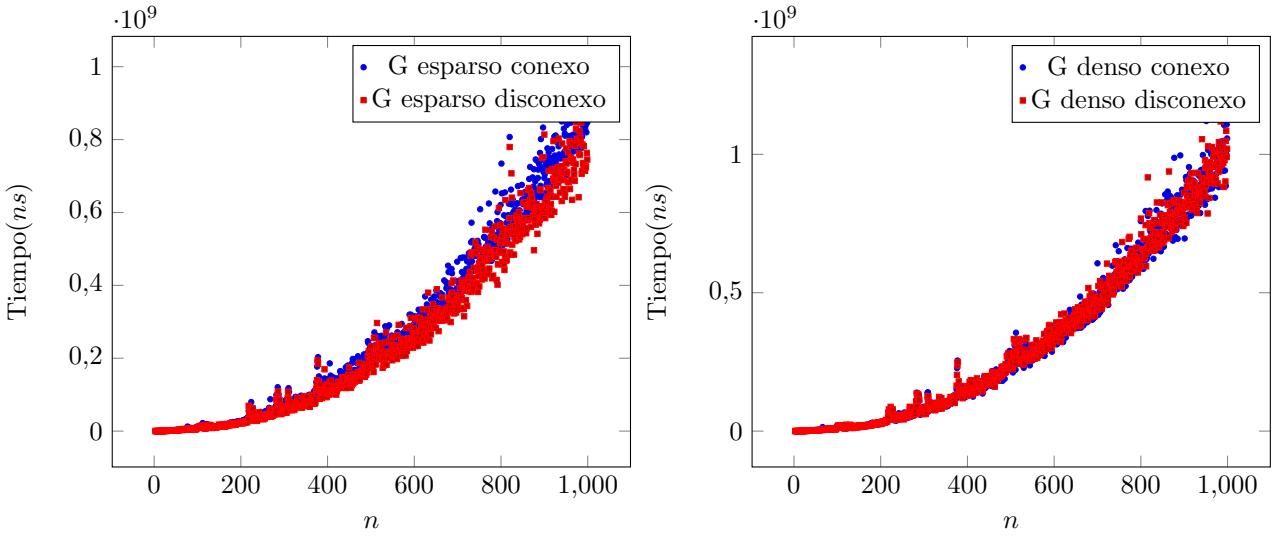
Lo cual es absurdo puesto que se contradice con la hipótesis inicial. Debe ser entonces que G_f sea la mejor solución. □

[1] Notar que P corresponde a la inversión efectuada, que es diferente a la función p que asigna pesos
[2] Restricción (abuso de notación)

Experimentación

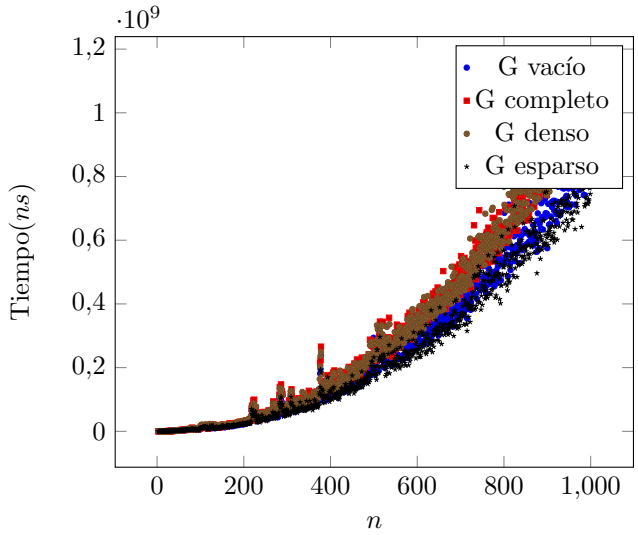
Para todos los experimentos, se tomaron, para cada n, el tiempo promedio que tardaron 5 instancias de n elementos y luego se tomo el tiempo promedio para representarlo en el gráfico. Además, el costo de los tiempos es el producto de construír el grafo a partir de las listas de rutas existentes/inexistentes y ejecutar la lógica de reconfiguración de rutas en sí.

En primer lugar, veamos como se comporta el algoritmo frente a grafos conexos o disconexos. Para ello, vamos a comparar los grafos esparsos generados con $m = n$ (ya que es $O(n)$) y a los densos con $m = \frac{(n-1)^2}{2}$ (ya que es $O(n^2)$). Por cada tipo, queremos ver cuanto cambia en base a si son conexos o no. Los resultados son:



Podemos notar que los tiempos de ejecución son relativamente parecidos para grafos conexos o disconexos.

En el próximo experimento, veamos como se comportan los tiempos de ejecución entre los grafos esparsos, densos, vacíos y completos. Los primeros no son necesariamente conexos (de todas formas vimos en el primer experimento que se comportan de forma similar en términos de tiempo).

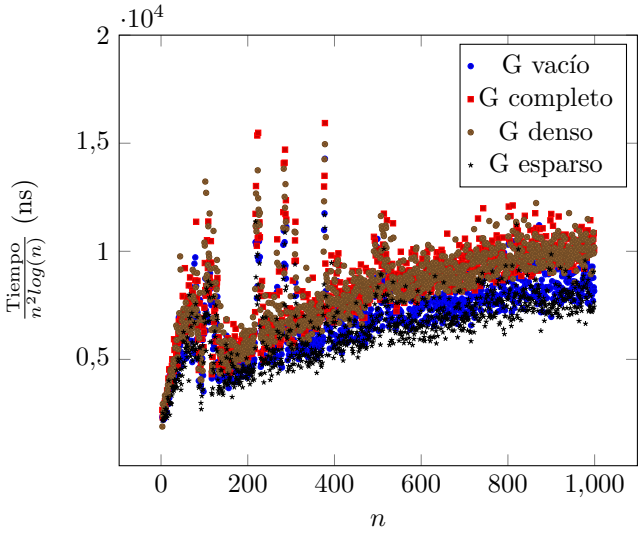


Lo que podemos notar es que en todos los casos, nuevamente, el tiempo tomado en cada caso es similar. Esto puede entenderse porque el algoritmo trabaja con las rutas (E) y las rutas no existentes (E^c).

Cuanto más tiempo tome el trabajo en alguno de estos conjuntos menos tiempo tomará en su complemento. En particular, para un grafo esparso su complemento es denso, por lo que en ese caso, el costo de destruír rutas va a ser menor que el costo de construír las nuevas. Equivalentemente vale la inversa y, además, esto aplica para los grafos vacíos y completos, puesto que estos son los grafos mas esparsos y densos respectivamente.

Otra observación es que, mirandolo con un poco de detalle, se puede ver que los grafos completos y densos tardan menos que los vacíos y esparsos. Esto es por los costos de trabajar con las diferentes componentes conexas. Cuantas más componentes conexas hayan, más aumenta la cantidad de operaciones que se realizan (se realizan mas copias de ejes, por ejemplo)

En el próximo experimento, se pretende ver que la complejidad teórica calculada se condice con los tiempos a medida que n crece. Para ello, tomamos el cociente entre el tiempo de ejecución y la complejidad teórica. Los resultados fueron los siguientes:



Sorpresivamente, no se puede asegurar a simple vista que los tiempos tiendan a una constante como esperábamos. Parecería ser que los tiempos crecen logarítmicamente más rápido que la complejidad teórica calculada.