Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales

Departamento de Computación

Algoritmos y Estructuras de Datos III

Trabajo Práctico 2

Autores:
Nicolás Chehebar, mail: *nicocheh@hotmail.com*, LU: 308/16
Matías Duran, mail: *mato_fede@live.com.ar*, LU: 400/16
Lucas Somacal, mail: *lsomacal@gmail.com*, LU: 249/16

Índice

1. Problema 1

1.1. El Problema

1.1.1. Descripción

Planteado de otra forma, el problema a resolver consiste en una situación en la que tenemos n trabajos $t_1, t_2, ..., t_n$ y dada cualquier división de los trabajos en dos secuencias $A = (t_{a_1}, t_{a_2}, ..., t_{a_{|A|}})$ y $B = (t_{b_1}, t_{b_2}, ..., t_{b_{|B|}})$ con $a_i < a_j \wedge b_i < b_j$ si $i < j$ (cada secuencia representa los trabajos que realizó una máquina) tiene asociado un costo; donde este viene dado por la suma del costo de A y el de B. El costo de A es $\sum_{i=1}^{|A|} costo(t_{a_i}, t_{a_{i-1}})$ donde *costo* es una función que toma valores en $\mathbb{N}_0$ y $costo(t_i, t_j)$ esta definidio si $i > j$ con $i \in [1, 2, .., n] \wedge j \in [0, 1, .., n-1]$ y represnta el costo de poner el trabajo i sobre el j (el costo de poner sobre el trabajo t_0 es el de ponerlo sobre la máquina vacía y $a_0 = 0$). El costo de B se calcula análogamente.

El problema pide dados los trabajos y la funcion de costo, dar A o B que minimice el costo y decir cuánto es este costo (basta dar uno de los dos ya que el otro se deduce por ser el complemento -en el conjunto de trabajos-)

1.1.2. Ejemplos

- En el caso en que la entrada es
  4
  2
  300 3
  300 3 3
  300 3 3 3
  tenemos 4 trabajos que sacando el primero son excesivamente caros de poner por primera vez en una maquina, luego si ponemos todos en la misma, el costo sera $2+3+3+3 = 11$ y una máquina tendrá todos los trabajos (si todos no estan en la misma, en algun momento pagamos 300 y el costo ya sería mayor a 11).

- En el caso en que la entrada es
  4
  2
  4 1
  300 3 300
  300 300 300 3
  tenemos 4 trabajos, ponemos el primero en una maquina y nos da costo 2, si bien en el proximo paso lo mejor es poner el nuevo trabajo encima (si hicieramos un algoritmo goloso), en ese caso el siguiente trabajo costará 300 haciendo el total > 299, y si no hubieramos puesto el segundo encima, si bien costaba mas en ese paso, reducía el costo del próximo, dando un costo total de 12 estando los trabajos 1, 3 y 4 en una máquina.

1.2. El Algoritmo

1.2.1. La función de dinámica

Para resolver el problema, utilizaremos programación dinámica. La idea de esto se basa en que la solución de nuestro problema es calculable en base a la solución de subproblemas (utilizamos optimalidad de subproblemas). Definimos así $f(q, h)$ como la función que asigna el mínimo costo posible para llegar al trabajo q-ésimo hecho (habiendo hecho del 1 hasta el q inclusive) con el trabajo h como el último que se hizo en algunas de las máquinas. Tomamos como dominio de f a los $q \in [1, 2, ..., t] \wedge h \in [0, 1, ..., q-1]$. donde $h = 0$ significa que hay una maquina vacía. Es clave notar que siempre que luego de que realizamos el trabajo q en una de las máquinas estará en el tope dicho trabajo, por ende basta definir qué hay en la

otra. Definimos a continuación la función para los valores en el dominio ya mencionado:

$$f(q,h) = \begin{cases} costo(1,0) & \text{si } q = 1 \wedge h = 0 \\ f(q-1,h) + costo(q,q-1) & \text{si } h < q-1 \\ \min\limits_{0 \leq h \leq q-2} f(q-1,h) + precios[q][h] & \text{caso contrario } (h = q-1) \end{cases} \quad (1)$$

Esta función hace efectivamente lo que queremos:

- En el primer caso lo hace pues si $q = 1$ esto implica $h = 0$ por restricciones de dominio y es la minima cantidad dado que coloqué solo el primer trabajo, pues sí o sí el costo será el de colocar el primero sobre la maquina vacía, por ende será el mínimo.
- En el segundo caso también lo hace pues si está un trabajo $h < q-1$ en una máquina es porque el último trabajo colocado (el q) se colocó en la otra, por ende previo a finalizar el trabajo q, estaba en una máquina el $q-1$ y en otra el h. Más aún sabemos que el q lo colocamos sobre el $q-1$. Supongamos $f(q,h)$ el costo mínimo dado el trabajo q hecho y el trabajo h en alguna impresora (análogo para $f(q-1,h)$), si es $f(q,h) < f(q-1,h) + costo(q,q-1)$ luego es absurdo pues $f(q-1,h)$ no es el mínimo, ya que hago la secuencia que da el mínimo en q trabajos hechos con h en una máquina sin el último paso (resto su costo, o sea el de poner a q sobre $q-1$) y me queda que tengo una forma de tener $q-1$ trabajos hechos con h en una máquina con costo $f(q,h) - costo(q,q-1) < f(q-1,h)$ lo que es absurdo pues $f(q-1,h)$ era el mínimo. Luego debe ser $f(q,h) \geq f(q-1,h) + costo(q,q-1)$ y por ende es un mínimo (quizás no el único).
- En el tercer caso también sucede pues, si está el trabajo $q-1$ en una máquina con la impresión q ya hecha, es porque la impresión q se colocó sobre alguna impresión h con $0 \leq h \leq q-2$. Dado dicho trabajo (de forma totalmente análoga al caso de arriba) debe ser el mínimo buscado con q trabajos hechos y el $q-1$ en una máquina $f(q,q-1) = f(q-1,h) + costo(q,h)$. Luego como no sé cuál trabajo de todos pudo haber sido, me quedo con el mínimo moviendo los h en el rango dado.

Así, definimos una función que resuelve el problema pedido si hallamos el $\min\limits_{0 \leq h \leq t-1} f(n,h)$ pues es el mínimo costo de realizar hasta el trabajo n (o sea todos) con el trabajo h en alguna máquina (me fijo todos los escenarios posibles como puede terminar la otra máquina, o sea todos los posibles h y me quedo con alguno que minimice el costo).

Así, podemos implementar la f dada, donde podemos ir recordando los valores que toma f y evitar calcularlos varias veces. Más aún podríamos mantener una lista (ordenada) de cuáles son los elementos que hay en alguna máquina y cada vez que agregamos un trabajo, chequeamos si lo agregamos sobre el último de la lista y en ese caso lo incluimos al final de esta (si no, es porque fue a la otra máquina).

Lo que sucede es que tenemos varios subproblemas y en este caso siempre resolvemos todos, por lo que no parece tener una clara ventaja hacerlo top-down. Más aún, hacerlo bottom-up nos permitirá solo guardarnos los subproblemas relativos a tener hecho exactamente hasta el anterior trabajo (con $q-1$ y para todos los h, notar los menores no los utilizo en el calculo de $f(q,h)$), o sea, nos reduce la complejidad espacial. Esto es así pues incialmente debíamos guardar el valor de $f\ \forall q \in [1,2,...,t] \wedge h \in [0,1,...,q-1]$ lo que sería $\mathcal{O}(n^2)$, y de esta forma solamente guardamos los valores para $q-1$ lo que es $\mathcal{O}(n)$.

Veamos todo esto en un pseudocódigo.

1.2.2. El Pseudocódigo

Cabe aclarar que en el pseudcódigo (como también en la implementación) numeramos los trabajos desde 0 a excepción del segundo índice de *costos* (que es una matriz) donde los trabajos están numerados

desde 1 (ya que el 0 se reserva para el costo de poner sobre la máquina vacía).

Algorithm 1: Devuelve el mínimo costo de entre todas las formas posibles de realizar todos los trabajos y una lista de trabajos realizados por alguna máquina que logra dicho costo

```
1  Dinámica (trabajos, costo);
   Input  : trabajos ∈ ℕ0; costo ∈ ℕ0^(trabajos×trabajos)
   Output: costo ∈ ℕ0, lista vector de enteros
2  Inicializo en 0 actualCosto y anteriorCosto vectores de enteros (de tamaño trabajos);
3  Inicializo en vectores vacíos actualLista y anteriorLista vectores de vectores de enteros (de tamaño
     trabajos);
4  for q ∈ [0, 1, .., trabajos) do
5      for h ∈ [0, 1, .., q) do
6          actualCosto[h] = anteriorCosto[h] + costo[q][q];
7          actualLista[h] = anteriorLista[h];
8          if Estaba q − 1 en anteriorLista[h] then
9              Agrego q a anteriorLista[h];
10         end
11         actualCosto[q] = min(actualCosto[q], anteriorCosto[h] + costos[q][h]);
12         Recuerdo en elegido el h que consiguió el minimo;
13     end
14     actualLista[q] = anteriorLista[elegido];
15     if NO Estaba q − 1 en anteriorLista[elegido] then
16         Agrego q a anteriorLista[elegido];
17     end
18     anteriorCosto=actualCosto;
19     anteriorLista=actualLista;
20 end
21 costo = mínimo de actualCosto (se alcanza en actualCosto[posicion]);
22 lista = actualLista[posicion];
23 return costo, lista;
```

En el pseudocódigo básicamente lo que hacemos es aplicar la f pero en orden, es importante esto ya que hay que tener cuidado en el orden en que resolvemos las dependencias (es porque estamos haciendo bottom-up). Es claro que cada fila, usa la fila anterior, o sea para calcular $f(q, h)$ para todo h uso todos los valores de $f(q-1, h)$ para todo h. Por esto es que ambos **for** se anidan de dicha manera. Al principio del **for** actualizamos el costo según nos dice la f y también actualizamos la lista de los trabajos que hay en alguna máquina, que se modifica sólo si era el de la máquina que tenía a $q-1$ (ya que es a la que le agrego el trabajo q). Además, como voy a recorrer todos los h, voy actualizando el $actualCosto[q]$ si tengo un menor $anteriorCosto[h] + costos[q][h]$; una vez que iteré en todos los h calculé el mínimo que es $f(q, q-1)$. Ahí salimos del primer **for** y (como hacía con $actualLista[h]$) actualizo si corresponde la $actualLista[q]$. Finalmente, antes de pasar a la siguiente iteración pongo en $anteriorCosto$ el $actualCosto$ y en $anteriorLista$ la $actualLista$, ya que en la próxima iteración los actuales serán anteriores y sobre lo que ahora pasó a ser actual pisaré y guardaré nuevos resultados. Finalmente, se devuelve el mínimo buscado y su lista asociada (lo que nos pedían era $\min_{0 \le h \le t-1} f(t, h)$ que en nuestro caso es el mínimo de $actualCosto$.

1.3. Complejidad

Cabe aclarar que para el análisis de complejidad tomaremos n como la cantidad de trabajos. Como pudimos ver en la explicación de la función de dinámica, tenemos n^2 subproblemas y cada uno se resuelve en $\mathcal{O}(1)$ salvo los subproblemas donde $h = q-1$ que toman $\mathcal{O}(n)$. Luego tengo $\mathcal{O}(n^2)$ subproblemas (son

menos de n^2 en total y siguen siendo menos si le saco los que no se resuelven en $\mathcal{O}(1)$ que son n) resueltos en $\mathcal{O}(1)$ cada uno y $\mathcal{O}(n)$ subproblemas (hay uno por cada q, son n) resueltos en $\mathcal{O}(n)$ cada uno. Luego se deduce que la complejidad será $\mathcal{O}(n * n) + \mathcal{O}(n^2 * 1) = \mathcal{O}(n^2)$

Más aún esto se ratifica si miramos el pseudocódigo ya que realizamos todas operaciones que son $\mathcal{O}(n)$ u $\mathcal{O}(n)$ fuera del ciclo (inicialización o recorrido de vectores de tamaño a lo sumo n). Veamos qué sucede dentro del ciclo: tenemos dos ciclos anidados que se ejecuta cada uno a lo sumo n veces, por ende todo se ejecuta a lo sumo n^2 veces y todo lo de adentro son operaciones $\mathcal{O}(1)$ (chequear si $q - 1$ está en $anteriorLista[h]$ es $\mathcal{O}(1)$ porque inserto siempre ordenado y si es un elemento, es el último; lo mismo vale para checkear si $q - 1$ está en $anteriorLista[elegido]$).Luego salimos del segundo **for** (el anidado) y cabe aclarar que copiar el vector $actualCosto$ y $actualLista$ no es $\mathcal{O}(1)$ sino $\mathcal{O}(n)$, pero está solo en uno de los ciclos, por lo que se repite n veces y aporta una complejidad de $n * \mathcal{O}(n) = \mathcal{O}(n^2)$ en total. Por ende, en el ciclo tenemos $n^2 * \mathcal{O}(1) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$ y sumado a lo que está fuera del ciclo nos da $\mathcal{O}(n) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$. Así, nuestro algoritmo logra la complejidad pedida.

1.4. Experimentación

1.4.1. Contexto

La experimentacion se realizó toda en la misma computadora, cuyo procesador era Intel Atom™ CPU N2600 @ 1.60GHz, de 36 bits physical, 48 bits virtual, con una memoria RAM de 2048 MB. Para experimentar, se calculó el tiempo que tardaba el algoritmo sin considerar el tiempo de lectura y escritura ni el tiempo que llevaba armar la matriz (ya que se leía un dato, se escribía la matriz y luego se leía el siguiente). El tiempo se medía no como tiempo global sino como tiempo de proceso, calculando la cantidad de ticks del reloj (con el tipo clock_t de C++) y luego se dividía el delta de ticks sobre CLOCKS_PER_SEC. En todos los experimentos el tiempo se mide en segundos.

1.4.2. Experimentos

Para empezar a experimentar, se corrió el programa con una serie de 2000 instancias generadas aleatoriamente con una cantidad de trabajos aleatoria entre 1 y 10^3 con una distribución uniforme [1] en dicho intervalo. Con un número entre 1 y 10^6 elegido aleatoriamente con distribución uniforme (de la misma manera) se eligió cada costo (cada elemento de la matriz de costos). En la Figura 1 se graficaron estas instancias.

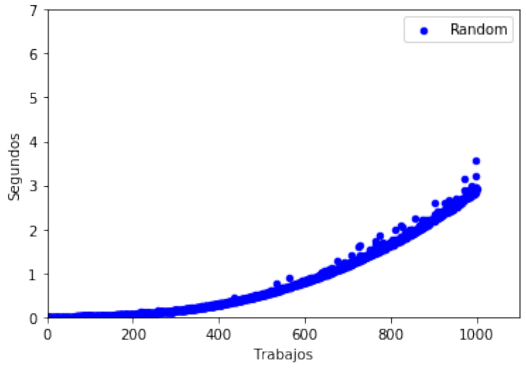


Figura 1: Gráfico de segundos de ejecución en función de cantidad de trabajos para instancias aleatorias.

Como se puede ver en dicha Figura, pareciera haber un gráfico semejante a una parábola lo que ratificaría la relación cuadrática que propusimos entre la cantidad de trabajos y la cantidad de operaciones realizadas. Sin embargo, no otorga información del todo completa para aseverar eso, ya que podría tratarse de alguna función con crecimiento similar. Es por esto que se realizó un gráfico de la relación entre tiempo de ejecución y $trabajos^2$ y si la relación es efectivamente cuadrática (o menor), este gráfico debería ser constante .

Como se puede ver en el gráfico de la Figura 2 efectivamente se trata de una constante, lo que verifica nuestra hipótesis y complejidad teórica de la relación cuadrática de dependencia entre la cantidad de operaciones y de trabajos. Podemos observar también (en la Figura 1) que no pareciera haber prácticamente dispersión, ni mejores ni peores casos, lo que analizaremos luego.

[1] se utilizó la función rand() de librerías de C++ en el rango correspondiente, para más detalle ver http://en.cppreference.com/w/cpp/numeric/random/rand

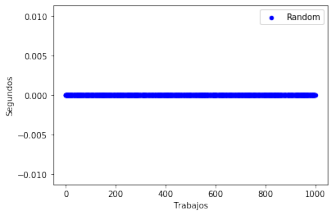
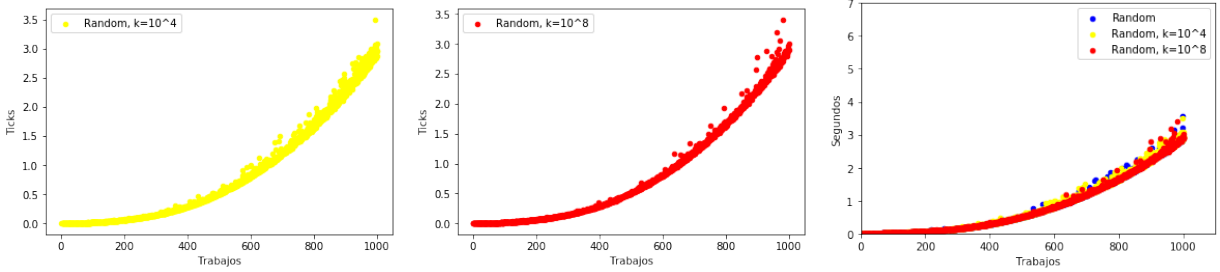


Figura 2: Gráfico de segundos de ejecución en función de cantidad de trabajos al cuadrado para instancias aleatorias.

Primero, debemos notar que según el análisis realizado, en el tiempo de ejecución solo influye la cantidad de trabajos, más alla del costo que pueda tener cada trabajo ya que en lo único que influye es en la suma (y como trabajamos con enteros acotados, no influye considerablemente en el tiempo de ejecución la suma). Por esto se ejecutaron las mismas 2000 instancias que sumando a todo costo una constante k que se movió entre 10^i con $i = 1, ..., 9$. En este caso, no cambió considerablemente el tiempo de ejecución[2] en todos los i como vemos a continuación en los casos $i = 4$ e $i = 8$ en las figuras 3.a, 3.b y 3.c.



(a) sumando $k = 10^4$ en costos

(b) sumando $k = 10^8$ en costos

(c) costos sin modificar (en azul), sumando $k = 10^4$ (amarillo) y sumando $k = 10^8$ (rojo)

Figura 3: Gráfico de segundos de ejecución en función de cantidad de trabajos para instancias aleatorias con distintos costos

.

Podemos ver que en todos los casos la dependencia sigue siendo, en rasgos generales la misma, cuadrática (se verificó haciendo el gráfico de $segundos/trabajos^2$ para cada i, los excluimos por una cuestión de espacio, pero todos resultaron constantes). Más aún, al comparar instancias de diversos i podemos ver que tienen similar tiempo de ejecución lo que nos indica que (sumado a que tomamos costos aleatorios) no hay influencia de los costos en el tiempo de ejecución, lo que tiene sentido por lo que hace el algoritmo y la complejidad teórica calculada.

Como decidimos implementar el algortimo de forma Bottom-Up siempre calculamos todos los subproblemas, esto es una ventaja en el sentido de que siempre todas las instancias de igual cantidad de trabajos tardan lo mismo, como se vio a lo largo de esta experimentacion, por lo que no hay mejores ni peores casos. Al ver la implementación y el pseudocódigo podemos ver que lo que realizamos depende exclusivamente de la cantidad de trabajos total (los costos solo cambian el resultado de cada cuenta, pero no la cantidad de operaciones ni su orden). También tomar esta decisión de implementar Bottom-Up nos permitió ahorrar en memoria ya que no requeríamos memorizar todos los subproblemas, sino que solo utilizábamos la información del subproblema anterior. La única desventaja es que a veces respecto de Top-Down, se calculan todos los subproblemas y no solo los necesarios. Pero si nos detenemos a ver la f que definimos al explicar el algoritmo (como ya también explicamos antes) siempre se van a calcular todos los subprobemas pues son todos necesarios, por ende esa tampoco es una ventaja del Top-Down en este caso. Todo esto se pudo ver experimentalmente ya que todas las instancias tuvieron un tiempo de ejecución muy similar y la dispersión fue practicamente nula.

[2]Notar que no cambia el resultado de cuáles trabajos quedan en una máquina respecto de la ejecución en la que no se sumó la constante

1.5. Conclusiones

Concluimos entonces que la complejidad es de $\mathcal{O}(n^2)$ como se vio teóricamente y además se pudo verificar de forma experimental. Como se analizó al implementar bottom-up se resolvían todos los subproblemas siempre, por lo que (como también se vió experimentalmente) no había diferencia entre casos, no había ni peores ni mejores casos, todas las instancias de igual cantidad de trabajos tomaban, prácticamente, el mismo tiempo de ejecución. Más aún se vio también experimentalmente que (como se esperaba y se deducía del algoritmo) no había influencia alguna de los valores de los costos en el tiempo de ejecución.

2. Problema 2

2.1. El Problema

2.1.1. Descripción

Si nos abstraemos de los detalles del problema, este nos describe una situación en la cual tenemos un grafo G (no orientado) conexo con pesos no negativos. Lo que nos piden en la parte 1 del problema es encontrar un conjunto de aristas $E' \subseteq X(G)$ del grafo que cumpla que la suma de sus costos sea la mínima posible (minimizar $\sum_{e \in E'} peso(e)$ y que el subgrafo H con nodos $V(G)$ y aristas E' sea conexo. En la parte 2 del problema nos piden, dado un E' que cumple lo antes descripto, elegir un nodo $v \in V(G)$ tal que si consideramos el subgrafo H sin pesos, minimice la máxima distancia de v a otro nodo (minimice $\max_{w \in V(H)} distancia(v, w)$).

2.1.2. Ejemplos

- Si consideramos K_n (el grafo completo de n vertices) con pesos constantes, todos 1 por ejemplo, la solución será cualquier conjunto de aristas que conecte todos los vértices (son al menos $n-1$ aristas) y con exactamente $n-1$ aristas se alcanza el mínimo (pues cada arista es de peso positivo, si no tuviese la mínima cantidad de aristas saco una y disminuye el peso). Así, la solución tendrá peso $(n-1) * 1 = n-1$ y podemos elegir tales aristas que cumplan que el subgrafo sea conexo (tomo la arista $(i, i+1)$ con $i = 1, 2, ..., n-1$ donde los nodos están numerados $1, ..., n$). Esta sería entonces una solución posible.

- Si consideramos C_n (el ciclo simple de n vertices) con pesos todos distintos positivos, la solución debe tener la mínima cantidad de aristas posibles (pues cada arista es de peso positivo, si no tuviese la mínima cantidad de aristas saco una y disminuye el peso) y para que sea conexo, estas son $n-1$. Luego basta excluir solo una arista y como quiero minimizar el peso y saque cual saque queda conexo, saco la de mayor peso y ya (es única la solución en este caso, pues son todos distintos y la arista de peso máximo es única). Las aristas buscadas serán todas menos la excluida y el peso, la suma de sus pesos.

2.2. Consultora 1

2.2.1. El algoritmo

Si nos detenemos a evaluar lo que pide la primera parte del problema, notamos que la solución debe permitir que sea conexo el grafo (debe tener $n-1$ aristas al menos) y debe minimizar la cantidad de aristas (pues cada arista es de peso positivo, si no tuviese la mínima cantidad de aristas saco una y disminuye el peso), luego debe tener exactamente $n-1$ aristas. Y además **debe ser conexo**, luego se trata de un árbol, y como debe tener como nodos a $V(G)$ es un árbol generador. Pero buscamos la solución de peso mínimo (o una de ellas), por ende la solución es un AGM (árbol generador minimo).

Para esto utilizamos el algoritmo de Prim (no pondremos su pseudocódigo por ser un algoritmo ya visto en clase y muy conocido, igual se puede ver el pseudocódigo en el problema 3, es cuestión de cambiarle las únicas dos modificaciones que están claramente marcadas y comentadas). Tomamos la opción de Prim en la que se utiliza un vector para implementar la cola de prioridad que tiene las distancias al AGM de los nodos no incluidos. Un breve resumen y descripción de lo que hace es que va construyendo un AGM, agregando un nodo (y una arista) en cada iteración. Itera n veces donde en cada una toma al nodo más cercano que no esté en el AGM (recorre linealmente todos los nodos de G). El nodo más cercano es aquel tal que la arista necesaria para incluirlo es la menos pesada de todas las que agreguen a algún nodo. Esto se hace recorriendo un vector en el que se guardan dos valores para cada nodo, si ya está incluido en el AGM, y el vecino suyo más cercano que esté incluido en el AGM (que es como guardar su distancia al AGM porque chequear la distancia entre dos vecinos es $\mathcal{O}(1)$ con nuestra representación). Una vez que se sabe qué nodo agregar al AGM, se actualizan todos sus vecinos. Donde al actualizar lo que se hace es chequear todos sus vecinos en su lista de adyacencia, y si él está más cerca que el nodo que ya teníamos registrado lo marcamos como el nuevo nodo más cercano.

Utilizamos como representación del grafo de entrada una matriz de adyacencia, para justamente poder acceder al peso de la arista que une dos nodos en particular en $\mathcal{O}(1)$. A la vez, como en cada iteración se necesita revisar todos los vecinos de un nodo particular que agregamos, nos tomamos el tiempo al principio de construir las listas de adyacencia para facilitar esta operación. Por último, para facilitar la escritura del output de la forma pedida, una vez que tenemos el AGM construido lo devolvemos en forma de matriz de incidencia. Sabemos que esto no empeora la complejidad porque como mucho un árbol tiene $n-1$ aristas, por lo que armarla cuesta $\mathcal{O}(n^2)$. Es importante aclarar que la consultora 1 devuelve el AGM en forma de listas de adyacencia, para que la consultora 2 pueda cumplir fácilmente con la complejidad pedida. Todas estas representaciones se construyen en $\mathcal{O}(n^2)$ por lo que no cambian la complejidad teórica del algoritmo.

2.2.2. Complejidad

Como bien sabemos, la complejidad del algoritmo de Prim puede ser o bien $\mathcal{O}(n^2)$ si se utiliza un vector para implementar la cola de prioridad que tiene las distancias al AGM de los nodos no incluidos (tomar el mínimo es $\mathcal{O}(n)$, pero actualizar una distancia es $\mathcal{O}(1)$) o bien $\mathcal{O}((m+n)log(n))$ si se utiliza un heap (tomar el mínimo y actualizar son ambos $\mathcal{O}(log(n))$. Ambas cumplen la complejidad pedida, pero en nuestro caso lo implementamos de la primera forma, por lo que la complejidad es $\mathcal{O}(n^2)$ que cumple lo pedido.

2.3. Consultora 2

2.3.1. El algoritmo

El algoritmo en sí es muy simple, la idea es encontrar el camino máximo del arbol que nos devuelve el algoritmo de la consultora 1 y tomar el nodo que está en la mitad del camino (o alguno de los dos si tiene una cantidad par de nodos el camino). Y para tomar el camino más largo, lanzamos BFS desde un nodo cualquiera v para medir los caminos mínimos (notar que los caminos son únicos, pues es un árbol) a todos los demas nodos (BFS es aplicable pues todas las aristas tienen el mismo peso en este caso) y sea w el que esta mas lejos. Luego lanzamos BFS desde w y sea z el que esté a mayor distancia de w. Luego el único camino entre z y w (único pues es un árbol) es el camino de máxima longitud que buscamos.

Lo que es quizás más complejo es entender por qué efectivamente esto funciona. Lo que nos piden es dado el árbol que devuelve la consultora 1, encontrar un nodo tal que si lo elegimos como raíz, la altura del arbol sea mínima (i.e, minimizar la máxima de las distancias). Veamos primero que esta distancia tiene que ser $\geq x/2$ donde $x =$ *longitud del camino simple maximo*. Supongamos que no, luego es $< x/2$ y por ende el camino entre dos nodos siempre sera $< x$ ya que un camino posible entre dos nodos a y b (no necesariamente simple, por ende de mayor longitud que el simple) es ir desde a hasta el nodo que elegimos como raíz y luego ir desde la raíz al b, como ambos caminos son de longitud $< x/2$ (es ir desde un nodo a

la raíz y el arbol tiene altura $< x/2$), se deduce que el camino de unir ambos tiene longitud $< x$. Luego, finalmente todo camino entre un par de nodos tiene longitud $< x$, luego x no era camino simple de longitud máxima (notar que el camino entre dos nodos es único), pues todo camino simple tiene longitud menor. Hemos visto que la distancia debe ser $\geq x/2$, por ende demostramos que encontrar el camino máximo y tomar como raíz un nodo de la mitad del camino, minimiza la altura.

Falta ver entonces que usar dos veces BFS como dijimos nos da efectivamente los dos nodos que dan el camino maximo. Sean a, b los dos nodos que son extremos del camino máximo. Y sea v el nodo desde el que inicialmente lanzamos BFS y w sobre el segundo que lanzamos BFS (el más lejano de v). Si quitamos v del árbol, este se nos divide en c componentes conexas. Si a y b pertenecen a distintas componentes conexas, luego el camino (es un árbol, luego es único) que los une pasa por v. Supongamos, ahora, sin pérdida de generalidad que w no está en la misma componente conexa que a (si no lo tomamos respecto a b, siempre hay uno con el que no está en la misma componente conexa, pues no puede estar en dos componentes conexas a la vez). Luego, si consideramos el camino desde w a v es de longitud mayor (o igual quizas) que el camino de a a v (pues w es el más lejano de v). Luego el camino de w a v unido con el de v a b tiene longitud mayor (o igual), lo que nos dice que necesariamente uno de esos nodos debe ser w. Luego tomamos el más lejano a w lanzando nuevamente BFS y obtenemos el camino máximo. Ya lo probamos si a y b están en diferentes componentes conexas, veamos qué sucede si a y b quedan en la misma componente conexa, pero en ese caso, repetimos el mismo argumento en la componente conexa desde el único elemento que estaba conectado con v como la raíz. Luego, w sigue siendo el más lejano a este (si algún w' fuese el nuevo más lejano, estaría más lejos que w de v pues solo sumo uno más en ambas distancias para llegar desde la nueva raíz a v y debo pasar sí o sí por ella pues es lo que une a la componente conexa con v, absurdo). Iteramos así y a cada paso reducimos en uno la altura del arbol que nos va quedando, si en algun momento a y b quedan en componentes conexas distintas, ya está por lo que probamos antes, sino, repetimos el argumento, hasta que en un momento (cuando la altura del arbol sea 2) al sacar un nodo nos quedan componentes conexas triviales y forzosamente a y b deben estar en componentes conexas distintas y vale lo que dijimos.

Luego hemos probado que hacer BFS dos veces de esta forma nos da el camino simple máximo, en realidad nos da sus extremos, pero como el camino es único, si el BFS además nos devuelve un vector en el que se aclara la distancia de cada nodo al nodo inicial del BFS, se puede reconstruir el camino máximo de la siguiente forma: Empezando por el nodo más lejano (el otro extremo que devuelve el BFS) se recorren todos sus vecinos y se agarra uno cuya distancia al original sea exactamente 1 menos que el actual (sabemos que existe porque de alguna forma se llego a este). Se repite este proceso tantas veces como nodos hay en el camino máximo, y como siempre la distancia al nodo original disminuye en 1, se llega al nodo con distancia cero, es decir el otro extremo del camino máximo. Como estamos en un árbol, sabemos que chequear cada vez todos los vecinos no trae problemas, porque aunque este método sea $\mathcal{O}(n+m)$, en un árbol, $m = n-1$ y las complejidades no cambian. Hemos probado ademas que un nodo de la mitad del camino simple máximo realiza el mínimo buscado (probamos que ninguna otra distancia menor funciona, por ende este es el mínimo). Luego, demostramos que nuestro algoritmo es correcto y hace lo que efectivamente queremos.

2.3.2. Complejidad

Ejecutamos dos veces BFS, que como bien sabemos es $\mathcal{O}(n+m)$ (ejecutarlo dos veces lo sigue siendo), pero como estamos en un árbol, $m = n-1$; luego $\mathcal{O}(n+m) = \mathcal{O}(n+n-1) = \mathcal{O}(n)$. Luego, una vez que tenemos los extremos del camino máximo, recorremos el grafo buscando las aristas que nos llevan entre ambos extremos, que lo hacemos en $\mathcal{O}(n)$. Nuevamente aclaramos que lo que se hace es empezar por un extremo del camino (el nodo más lejano que encontró la segunda llamada a BFS) y viendo todos sus vecinos se agarra alguno cuya distancia al otro extremo sea exactamente 1 menos. Esto se repite hasta llegar al nodo de distancia cero y llegado este punto recorrimos el camino máximo, guardando todos los nodos en un vector. Esto se puede realizar en $\mathcal{O}(n)$ porque como mucho se pasa por todos los nodos y se chequea todas las aristas, pero en un árbol $m = n-1$ y, entonces, la complejidad queda como se dijo. Finalmente tomamos el nodo de la mitad de la lista de nodos que nos dio este recorrido. Como solo hicimos

tres cosas que son $\mathcal{O}(n)$, la complejidad total es esa y cumple lo pedido.

2.4. Experimentación

2.4.1. Generación de instancias

Para llevar a cabo la experimentación, en primer lugar se generaron instancias, con tamaños entre $1 <= n <= 100$ y para cada uno de estos tamaños, se construyeron 100 casos. Cada caso tiene como mínimo $n-1$ aristas para que el grafo sea conexo pero a esto se agregaban una cantidad de aristas que se tomaban de forma aleatoria entre 0 y $n*(n-1)/2-(n-1)$ (o sea que el grafo podría ser un árbol -mínima cantidad de aristas-, K_n -máxima cantidad de aristas- o cualquier grafo con una cantidad de aristas intermedia). Se tuvo cuidado a la hora de generar los grafos para asegurarse no solo la correctitud de las instancias de prueba, sino además, su variedad, evitando casos como todos grafos conexos pero en los que siempre hay un nodo conectado a todos los otros.

La forma de generarlos fue ir agregando los nodos uno a uno, y al agregar al nodo i, siempre incluír una arista (i,t) con $1 <= t < i$ para asegurar que el grafo hasta ahora construido sea conexo.

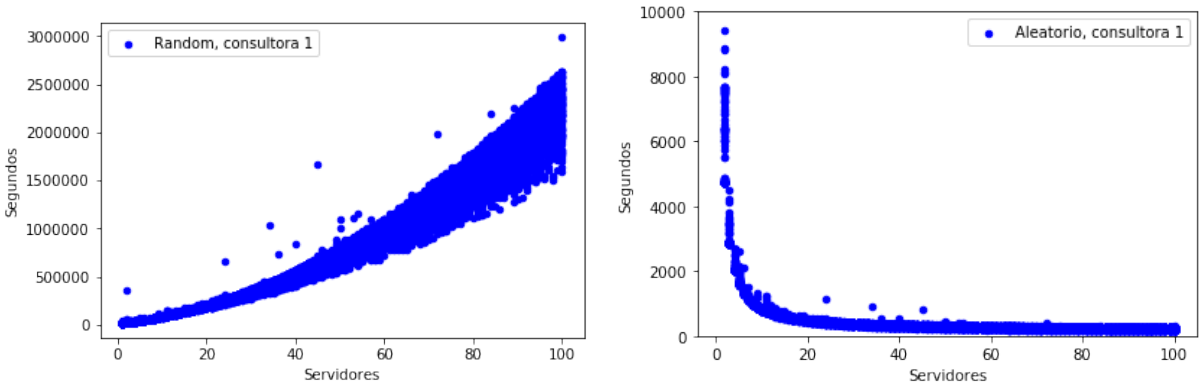
A los pesos de las aristas se les dio un rango amplio de valores posibles, desde 0 hasta n^2 para que puedan darse todas las opciones posibles (una arista que pese más que todas las otras, todas aristas de distinto peso, etc. -aunque no necesariamente equiprobablemente-).

Se tuvo cuidado también de no generar multigrafos: en la generación de cada caso se construyó una matriz de adyacencia a la que se agregaban aristas nuevas si y solamente si sus extremos no estaban ya conectados en la matriz de adyacencia.

Cada vez que se tomaron numeros aleatorios en la generación de instancias se utilizó una distribución uniforme [3].

2.4.2. Consultora 1

Al correr las instancias generadas se esparaban ver resultados que reforzaran las complejidades calculadas teóricamente. Se esperaba que se viera un tiempo de ejecución dependiente de la cantidad de servidores del grafo de entrada, con una relación de forma polinomial, más específicamente de n^2 (n es la cantidad de servidores). Estos son los resultados obtenidos.



(a) tiempo según n

(b) tiempo dividido n^2

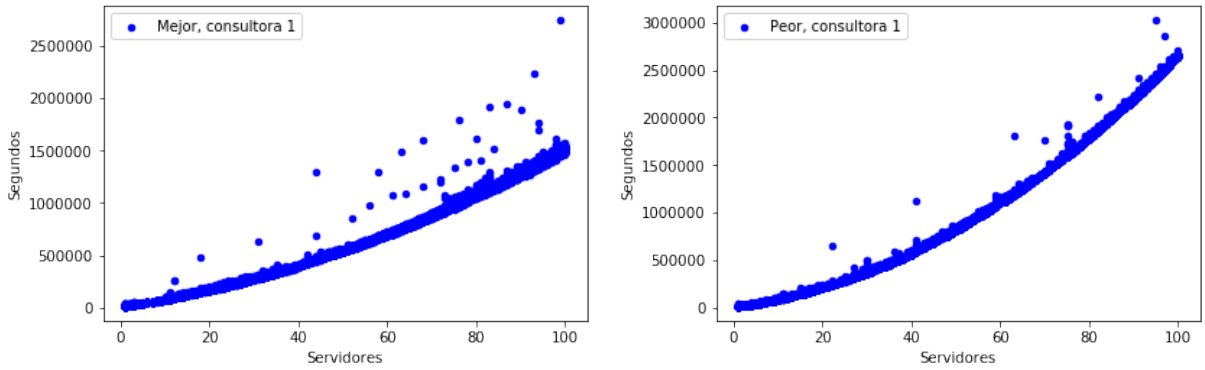
Figura 4: Tiempo de ejecución de la consultora 1 para instancias aleatorias

En la figura 4.a se puede observar un crecimiento que se adecúa bastante bien a la complejidad cuadrática esperada. Incluso más, en la figura 4.b se graficó el tiempo de ejecución dividido la cantidad de servidores

[3] Se utilizó la función rand() de librerías de C++ en el rango correspondiente, para mas detalle ver http://en.cppreference.com/w/cpp/numeric/random/rand

al cuadrado, esperando que, como la complejidad era cuadrática, se viera en el gráfico resultante una constante. Vemos que exceptuando los casos con un tamaño muy chico, en el que el término constante de la complejidad toma más importancia (ya que la división por pequeños valores de n no afecta mucho), todos los casos con un tamaño mayor a 20 se ubican en una recta horizontal.

Por otro lado, para entender los contextos de uso en los que este algoritmo mostraría un buen desempeño se buscaron peores y mejores casos. Se pensó que la cantidad de aristas, si bien están acotadas en la complejidad teórica, son relevantes para el desempeño empírico de nuestro algoritmo. Como hay que recorrer todas las aristas para ir encontrando los nodos más cercanos al AGM, el peor caso sería un grafo completo, en el que hay que recorrer muchas aristas. En cambio, en el grafo que se chequean pocas aristas sería un mejor caso. Se generaron entonces instancias de la forma antes detallada, pero fijando la cantidad de aristas totales a $n-1$ y $n*(n-1)/2$ para el mejor y peor caso respectivamente (la cantidad de aristas agregadas a las que aseguran conexión son 0 y $n*(n-1)/2-(n-1)$ respectivamente).



(a) mejor caso

(b) peor caso

Figura 5: Tiempo de ejecución de la consultora 1 para el mejor y el peor caso

Se puede observar en las figuras 5.a y 5.b como, si bien se mantiene a grandes rasgos la dependencia de la cantidad de servidores ya confirmada en la figura 4, se ve un rendimiento peor en la figura 5.b (los grafos completos) que en la 5.a (los árboles). Se mostrará en la Figura 6, cómo estas instancias construidas para resultar en los tiempos de ejecución más extremos, muestran tiempos de ejecución que acotan a los tiempos de los casos aleatorios.

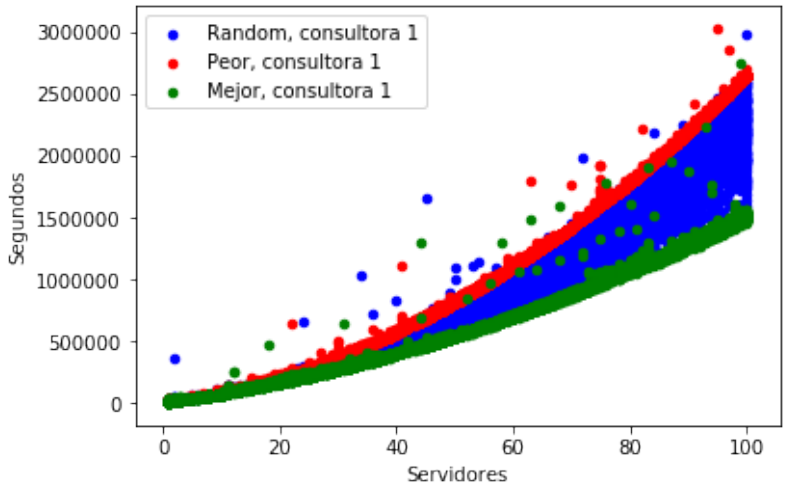
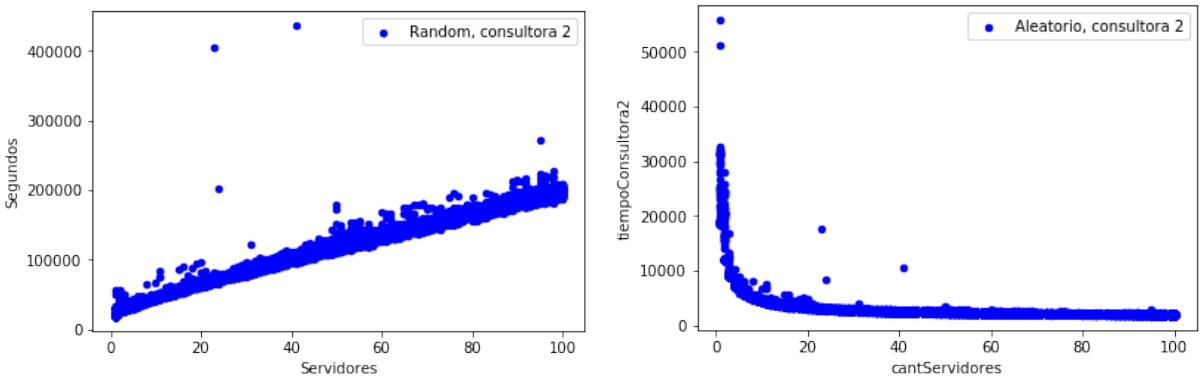


Figura 6: Comparación de los tiempos de ejecución para la consultora 1

Se puede apreciar en la Figura 6, que si bien hay casos excepcionales dados por las incertezas de la medición empírica, las instancias aleatorias presentan una eficiencia acotada por los mejores y peores casos predichos. Con esto se concluye que efectivamente se trataban de mejores y peores casos y que el tiempo de ejecución que requiere la consultora 1 depende de la cantidad de aristas del grafo de entrada. Si bien no aumenta la complejidad teórica, si puede causar una diferencia considerable en el tiempo de ejecución, como se ve entre los peores y los mejores casos.

2.4.3. Consultora 2

Al igual que con la consultora 1, el estudio teórico de la complejidad del algoritmo lleva a esperar ciertos resultados en la etapa experimental. En este caso particular, se verá que el tiempo de ejecución depende de la cantidad de servidores del árbol a analizar por la consultora 2. La relación entre el tiempo y el tamaño de entrada es de forma lineal, y no puede variar según la cantidad de aristas ya que el árbol siempre tiene la misma cantidad $(n-1)$. Lo que sí afectará ligeramente el tiempo de ejecución es el largo del camino máximo del arbol, como se verá en el estudio de los peores y mejores casos del algoritmo.



(a) tiempo según n

(b) tiempo dividido n

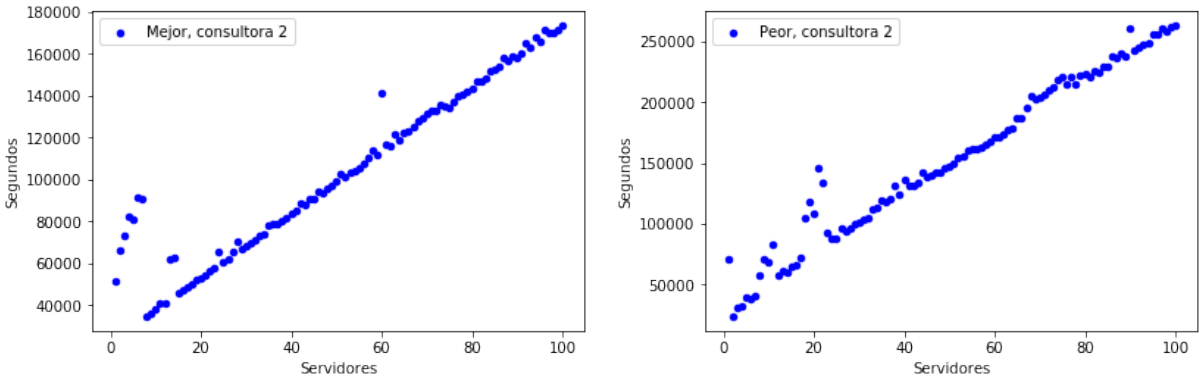
Figura 7: Tiempo de ejecución de la consultora 2 para instancias aleatorias

En la figura 7.a se ve cómo el tiempo necesario para la resolución de la instancia de entrada depende directamente del tamaño de la misma. Se ve que presenta la forma de una recta, una función lineal. En la figura 7.b se repitió la idea de dividir a la función por el tamaño de entrada, buscando como resultado un gráfico de una recta horizontal (constante) que confirme la complejidad lineal de nuestro algoritmo. Desechando los casos pequeños, y analizando los tamaños mayores (que son los que nos interesan al estudiar la complejidad teórica de los algoritmos) vemos que tienden a una constante tal como se había predicho.

A continuación, una vez confirmada la complejidad para los casos genéricos, se buscó encontrar los casos en los cuales el algoritmo mejoraba o empeoraba su eficiencia. Como en la primera parte (en el BFS) siempre se recorre todo el árbol, el énfasis se puso en el final, la reconstrucción del camino máximo para luego encontrar su medio. Si este camino fuera corto este proceso sería rápido, mientras que si fuera más largo el tiempo de ejecución sería mayor.

Se decidió generar mejores casos en los que el camino máximo del árbol fuera de tamaño mínimo (2), para esto se generaron grafos en los que un nodo estuviera conectado a todos los demás (de este se "colgará" el árbol al terminar el algoritmo) sin ninguna otra arista extra, para asegurarnos que este sea el árbol devuelto por la consultora 1.

Para los peores casos se debían generar grafos en los que el AGM resultante tuviera un camino máximo de longitud $n-1$ (contando la cantidad de aristas -ya que cada arista tiene el mismo peso-), esto se logra generando instancias en las que cada servidor tiene como vecinos sólamente al servidor inmediatamente anterior y al inmediatamente posterior.



(a) mejor caso (b) peor caso

Figura 8: Tiempo de ejecución de la consultora 2 para el mejor y el peor caso

En ambos gráficos de la Figura 8 se puede apreciar claramente la forma lineal de la complejidad del algoritmo de la consultora 2. Lamentablemente no es claro que los peores casos sean efectivamente peores, para esto se recurrirá a la Figura 9 en la que se comparan los mejores y peores casos con las instancias aleatorias descriptas previamente.

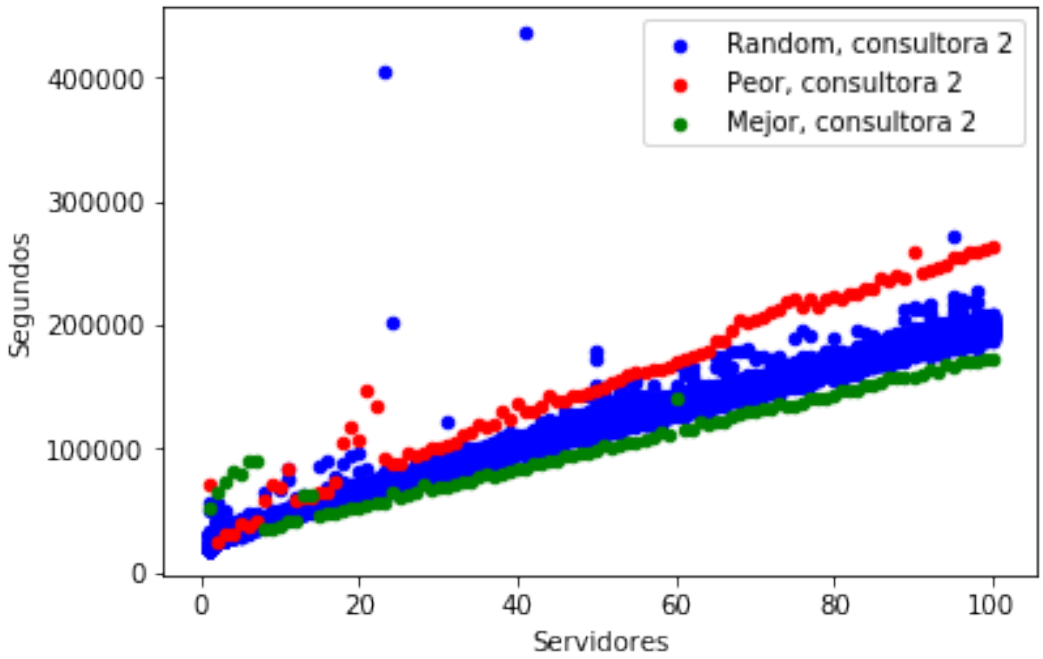


Figura 9: Comparación de los tiempos de ejecución para la consultora 2

Ahora sí, en la Figura 9 se puede ver cómo los casos con un camino máximo de longitud 2 son resueltos por el algoritmo de forma más rápida que aquellos que tienen un camino máximo más largo, especialmente más rápida que los peores casos con un camino máximo que recorra todo el árbol. Aquí se ve también que efectivamente se trataban de peores y mejores casos ya que acotan superior e inferiormente (respectivamente) los tiempos de ejecución de los generados de forma aleatoria.

2.5. Conclusiones

Se pudo verificar a partir de la experimentación computacional que las complejidades teóricas predichas ($\mathcal{O}(n^2)$ y $\mathcal{O}(n)$) son efectivamente ciertas. Además, con el estudio de los peores y mejores casos comprobados en las figuras 6 y 9 se obtuvo información valiosa sobre el desempeño de los algoritmos bajo distintas condiciones de uso. Con este tipo de estudios se puede diferenciar entre distintas implementaciones de algoritmos que si bien cumplen con la misma complejidad teórica, presentan un desempeño distinto bajo diferentes condiciones de las instancias de entrada. También pudimos conlcuir que efectivamente había dependencia respecto de la cantidad de aristas del grafo para el tiempo de ejecución de la consultora 1 y hallamos una dependencia respecto del la longitud del cámino máximo del AGM (devuelto por la consultora 1) para el tiempo de ejecución de la consultora 2. Sabemos, entonces, que las consultoras en cuestión presentarán tiempos de ejecución menores para grafos con una menor cantidad de aristas y tales que todo AGM presente un camino máximo lo más corto posible. Recíprocamente los tiempos serían mayores a mayor cantidad de aristas y un cámino maximo lo más largo posible en todo AGM.

3. Problema 3

3.1. El Problema

3.1.1. Descripción

Planteado de otra forma, la situación que tenemos es un grafo (no orientado) con pesos positivos en las aristas G en el cual tenemos una partición de $V(G) = F, C$ dada (con $|C| \geq |F|$) y queremos hallar un subconjunto de aristas $E' \subseteq X(G)$ que tenga peso mínimo (o sea minimizar $\sum_{e \in E'} peso(e)$) y que cumpla que para todo nodo en C existe un camino a algun nodo en F (o sea, para toda componente conexa W del grafo $H = (V(G), E')$, $\exists\ v \in F$). Nos piden hallar ese costo mínimo (i.e $\sum_{e \in E'} peso(e)$) cuántas y qué aristas lo logran.

3.1.2. Ejemplos

- Consideremos C_n (supongamos n par) con pesos asociados todos iguales y bipartito con F y C los elementos de la partición (o sea, en el ciclo, si lo recorremos en algun sentido, hay un nodo de F, luego uno de C, luego uno de F y así sucesivamente). Es claro que debemos minimizar la cantidad de aristas (todas pesan lo mismo y tienen costo positivo) y como tenemos $|C| = n/2$ necesitaremos al menos $n/2$ aristas y consideramos alguna de las que la unen con alguno de sus vecinos para cada elemento de C. Así tenemos nuestro conjunto de aristas que minimizan la suma (será $k * (n/2)$ si k es el peso de cada arista), de hecho es claro en este ejemplo que el conjunto de aristas que minimiza la suma no es único (de hecho hay $2^n/2$ -pues para cada elemento de C elijo una de las dos aristas que inciden en él).

- Consideremos ahora un grafo compuesto de q componentes conexas donde cada una es un C_n como mostramos en el ejemplo anterior (con n par para toda componente conexa y con un elemento de C y uno de F alternadamente y pesos iguales k). Para minimizar la cantidad de aristas, debemos resolver el problema en cada una de las componentes conexas, pues la única forma de llegar a un elemento de una componente conexa es desde alguna fábrica que esté en esa componente. Luego, como vimos cada componente se resuelve con $k*(n/2)$ de peso total, por ende la solución total tendrá $q*k*(n/2)$

3.2. El Algoritmo

3.2.1. Resumen

Lo que tenemos que hacer es algo bastante parecido a encontrar un AGM, pero como hemos visto incluso en los ejemplos, la solución no tiene por qué ser un árbol. Más aún ni siquiera tiene que generar (puede que haya un nodo que no sea alcanzable, en ese caso sería uno de F -por ejemplo uno que tiene un costo altísimo cada arista que lo une con cualquier otro y siempre es más barato llegar a sus vecinos desde otro elemento de F-). Pero si nos detenemos a pensar, no tiene sentido que haya un ciclo, ya que sacamos una arista y (como todas tienen peso positivo) disminuye el peso. Luego nuestro grafo solución no es un AGM, pero sí es un bosque que tenga a todos los elementos de C y sea de peso mínimo. Y un bosque es un conjunto de árboles, queremos hacer prácticamente lo mismo que en un AGM, pero sin mantener necesariamente la conexión en el grafo que vamos generando (al que le agregamos un nodo y una arista en cada iteración). De hecho, sabemos que este grafo tendrá exactamente $|C|$ aristas, una por cada iteración ya que en cada iteración agregaremos al cliente "más cercano". Así surge la idea de lo que realizamos: hacer Prim pero comenzando en vez de con un nodo, con todos los nodos de F.

3.2.2. El Pseudocódigo

Como se trata efectivamente de una variación del algoritmo de Prim, incluiremos un pseudocodigo

Algorithm 2: Devuelve un conjunto de aristas que conectan a todo elemento de C con alguno de F con menor costo y su costo

```
1  PrimModificado (G);
   Input  : GrafoG, F ⊆ V(G)
   Output: costo ∈ ℕ0, lista vector de aristas
2  for v en V(G) do
3  |  distancia[u] = ∞;
4  |  padre[u] = NULL;
5  |  Añadir a la cola (u, distancia[u]));
6  end
7  for f en F do
8  |  distancia[f]=0;                                  ▷ Cambio respecto de Prim
9  end
10 while NO esta vacia la cola do
11 |  for v adyacente a u do
12 |  |  u = extraer el de menor distancia de la cola que ∉ F;   ▷ Cambio respecto de Prim
13 |  |  if v ∈ cola ∧ distancia[v] > peso(u, v) then
14 |  |  |  padre[v] = u;
15 |  |  |  distancia[v] = peso(u, v);
16 |  |  |  Actualizar la cola (v, distancia[v]);
17 |  |  end
18 |  end
19 end
```

Como se ve en el pseudocódigo, para resolver el problema usamos el mismo algoritmo que usa Prim, solo que en vez de empezar con algun nodo (cualquiera) marcado como el primer elemento del AGM, empezaremos con todos los elementos de F marcados. Además, cuando tomemos el nodo de menor distancia, tomamos el de menor distancia y que $\in C$. Así en cada iteración incluimos al nodo de C que está a menor distancia del grafo hasta entonces generado, cada vez aumentamos el grafo inicial en un nodo hasta que no queden más elementos en C (a diferencia de prim en que este grafo siempre era un árbol), y en cada paso siempre incluimos al que suma menor costo (y le actualizamos la distancia a todos sus vecinos). Así,

cuando no queden más elementos de C por incluir, tendremos el conjunto de aristas que buscamos y serán las de peso mínimo (el argumento es exactamente el mismo que el de correctitud de Prim, siempre a cada paso agregamos el más cercano -si hubiera una arista e que convenía ser incluida en vez de otra f porque disminuiría el peso, no habría elegido a f en ningun momento pues siempre habría algun elemento a incluir con un costo menor al de f-).

3.3. Complejidad

Como ya analizamos en el problema 2, la complejidad de prim, por como lo implementamos, es $\mathcal{O}(n^2)$ y por ende ahora será $\mathcal{O}((|F| + |C|)^2)$ pero si recordamos, el enunciado nos asegura que $|C| \geq |F|$, luego $(|F| + |C|) \leq 2 * |C| = \mathcal{O}(|C|$ y por lo tanto, $\mathcal{O}((|F| + |C|)^2) \leq \mathcal{O}(|C|^2)$. Mostramos así que el algoritmo cumple la complejidad propuesta. Es claro que nuestra variación no afecta en absoluto la complejidad de prim, pues como implementamos la cola como un arreglo y buscar el mínimo nos tomaba $\mathcal{O}(n)$, encontrar el mínimo dentro de los que pertenecen a C sigue siendo $\mathcal{O}(n)$ (donde n es el tamaño de la cola). Agregamos sí un ciclo que inicializa todas las distancias de los elementos que estan en F, eso es $|F|$ operaciones $\mathcal{O}(1)$, lo que es $\mathcal{O}(|F|) \leq \mathcal{O}(|C|)$ y por ende no suma complejidad. Así, hemos probado que la modificación del algoritmo de Prim, mantiene la misma complejidad teórica, por lo tanto hemos probado que nuestro algoritmo es complejidad $\mathcal{O}(|C|^2)$.

3.4. Experimentación

3.4.1. Contexto

La experimentacion se realizó toda en la misma computadora, cuyo procesador era Intel Atom ™ CPU N2600 @ 1.60GHz, de 36 bits physical, 48 bits virtual, con una memoria RAM de 2048 MB. Para experimentar, se calculó el tiempo que tardaba el algoritmo sin considerar el tiempo de lectura y escritura ni el tiempo que llevaba armar la matriz (ya que se leía un dato, se escribía la matriz y luego se leía el siguiente). El tiempo se medía no como tiempo global sino como tiempo de proceso, calculando la cantidad de ticks del reloj (con el tipo clock_t de C++). En todos los experimentos se medirá en Ticks el tiempo de ejecución.

3.4.2. Experimentos

Primero, se genero una serie de casos aleatorios, generados de la misma forma que en el Problema 2. La única diferencia radicó en que en vez de a partir del segundo nodo conectarlo con alguno de los anteriores para asegurar conexidad, esto se hizo a partir del nodo $F + 1$ (los nodos mayores a F serán los clientes, y los menores o iguales las fábricas). Se corrieron casos con C entre 1 y 60 y para cada uno de ellos, se movió el F entre 1 y C (para respetar que siempre $C > F$) y para cada uno de esos valores de C y F se ejecutaron 10 casos aleatorios (en donde todo se realizo como se describió en el problema 2).

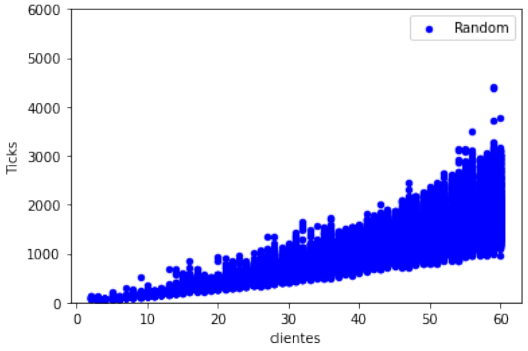
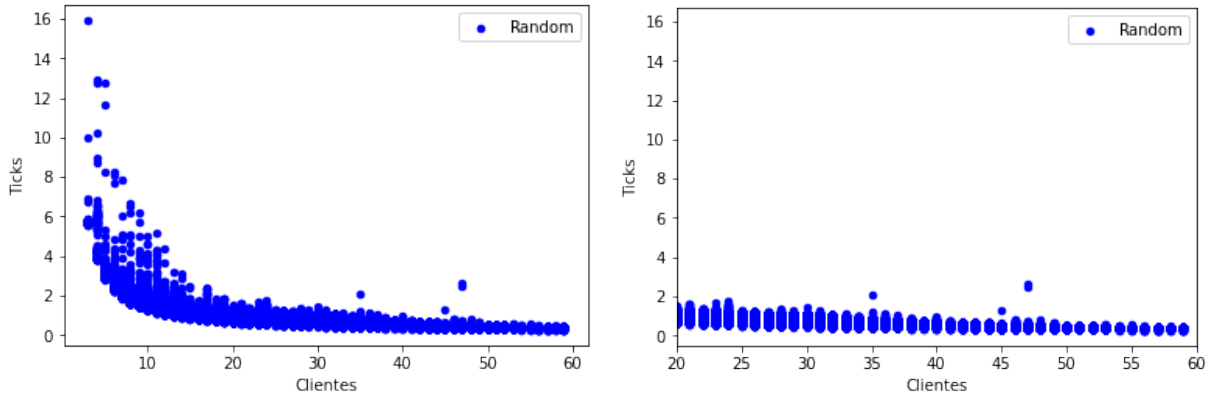


Figura 10: Gráfico de segundos de ejecución en función de cantidad de clientes para instancias aleatorias.

Como vemos en el gráfico de la Figura 10, parece haber un crecimiento del tiempo de ejecución cuando crece la cantidad de clientes. Para verificar que este crecimiento hace que efectivamente estemos resolviendo el problema en $\mathcal{O}(|C|^2)$, hicimos un gráfico de Ticks en función de clientes al cuadrado, donde se puede ver (salvo para pocos clientes) que el gráfico es acotable por una constante (ver Figura 11.a). En los primeros casos de clientes no sucede ya que al ser pequeña la cantidad de clientes, toma mucha más importancia el termino constante. Más aún, como la notación $\mathcal{O}$ solo nos habla de la complejidad asintótica, esto tiene

sentido. Pero dejando de lado estas instancias de $|C|$ pequeño, se ve que se puede acotar por una constante como se esperaba teóricamente (ver Figura 11.b).



(a) eje x entre 0 y 60 (todas las instancias).

(b) eje x acotado entre 20 y 60.

Figura 11: Gráfico de tiempo de ejecución en función de cantidad de clientes al cuadrado para instancias aleatorias

Sabemos que todos los pesos son enteros (del tipo `int` de C++) y el costo de hacer cualquier operación sobre los mismos no cambia (al menos para el modelo de complejidades que utilizamos, en que suponemos de costo básico todas estas operaciones). Como en el algoritmo, lo que se ve claramente en el pseudocódigo, lo único que hacemos con los pesos es compararlos de a pares para decidir cuál es el menor e incluirlo y además lo sumamos al costo total. Como la comparación y la suma no dependen de qué entero sea, podemos concluir que no hay influencia alguna del peso de las aristas, o sea el costo de reparar cada ruta. Por ende esta variable siempre se tomó aleatoria, teniendo la certeza de que no influiría en las otras.

Por otro lado, si bien la cantidad de fábricas no aparece en la complejidad teórica, al empezar el algoritmo de PrimModificado las "pintamos"todas y revisamos todos sus vecinos para actualizar las distancias al AGM. Pero si recordamos el enunciado, este nos da la condición de que $|F| < |C|$, razón por la que no influye $|F|$ en la complejidad teórica explicada previamente. Pero a la hora de efectivamente ejecutar el algoritmo, estas operaciones se realizan y, aunque estén acotados teóricamente, con una mayor cantidad de fábricas habrá que chequear más vecinos en el paso inicial, lo que aumenta el tiempo de ejecución. Para comprobar esta correlación realizamos el siguiente gráfico:

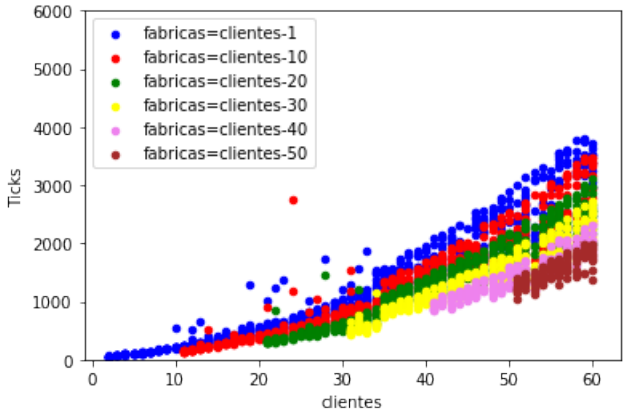


Figura 12: *Gráfico de tiempo de ejecución en función de cantidad de clientes para instancias aleatorias con* $|fabricas| = |clientes| - k$ *con* $k = 1, 10, 20, 30, 40, 50$.

Como vemos en la Figura 12, cuanta mayor cantidad de fábricas tengamos, en relación con la cantidad de clientes, el tiempo de ejecución es mayor ($fabricas = clientes - 1$ es el de mayor y $fabricas = clientes - 50$ es el de menor). Como se explicó anteriormente, esto tiene sentido ya que al principio del algortimo chequeamos todos los vecinos de las fábricas (al ser los primeros nodos pintados en Prim).

Queda por analizar entonces la dependencia del tiempo de ejecución en función de la variable R. Sabemos que la complejidad del algoritmo es $\mathcal{O}(|C|^2)$, pero si recordamos la complejidad teórica cuando calculamos la del algoritmo de

Prim, en un momento recorremos todos los vecinos de un nodo, y como repetimos esto para todos los nodos, en total lo hacemos $X(G)$ veces que en este caso es R. Luego, como $|V(G)|^2 \geq |X(G)|$ para todo grafo G, acotamos el R por $|C|^2$, pero este R influye en el tiempo de ejecución, lo que nos lleva a pensar que dentro de instancias de igual tamaño, las que tengan mayor cantidad de aristas tendrán un mayor tiempo de ejecución y las que tengan menos aristas un menor tiempo de ejecución.

Para ver esto, realizamos un gráfico de tiempo de ejecución en funcion de R. Efectivamente en el gráfico parecía verse una correlación, lo que verificamos utilizando el índice de pearson, como se puede ver en la Figura 13.

Efectivamente, hay correlación (ya que el p-value es 0) y como el índice de pearson es positivo, nos indica que al crecer la cantidad de rutas, crece el tiempo de ejecución.

Trataremos de entender si esto es porque al tomar valores mayores de R, deben ser mayores los de $|F|+|C|$ (puesto que $R \leq (|C|+|F|)(|C|+|F|-1)/2$ ya que la máxima cantidad de aristas -rutas- se da en el completo $K_{|C|+|F|}$, que es el que logra la igualdad) y por ende, como hemos visto, mayor el tiempo de ejecución. Para esto, fijamos la cantidad de fábricas y clientes y analizamos la relación del tiempo de ejecución respecto de R siendo esta la única variable libre.

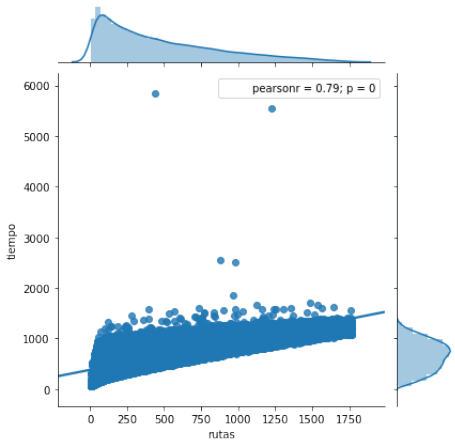
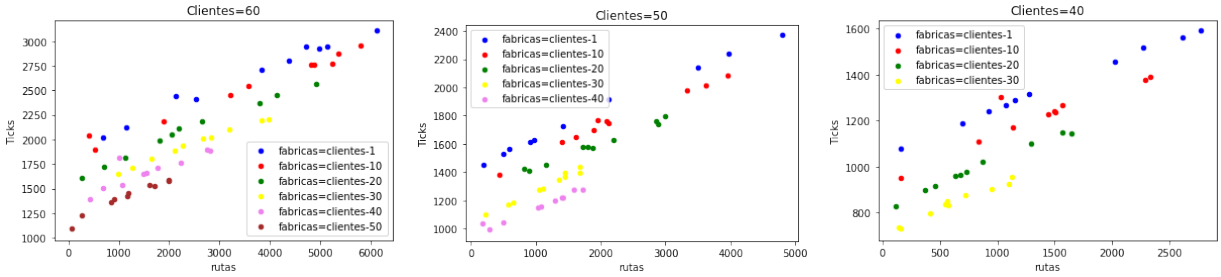


Figura 13: Gráfico de tiempo de ejecución en función de cantidad de rutas para instancias aleatorias con el índice de pearson.



(a) $|C|$=60 (b) $|C|$=50 (c) $|C|$=40

Figura 14: Gráfico de tiempo de ejecución en función de $|R|$ para valores fijos de $|C|$ y $|F|$

En la Figura 14, se tiene en cada subfigura, una cantidad de clientes fijada (en 60, 50 y 40) y para cada una de esas subfiguras, tenemos en distintos colores distintas cantidades de fábricas dada esa cantidad de clientes. Lo que se graficó es el tiempo de ejecución en función de la cantidad de rutas y se puede ver que en todos los valores de c, a mayor cantidad de rutas mayor tiempo de ejecución (ya que en todos los colores es creciente). Más aún, volvemos a verificar la dependencia según la cantidad de fábricas ya que los colores que representan mayor cantidad de fábricas tienen un mayor tiempo de ejecución.

Como ya se dijo, se observó que hay dependendencia en la cantidad de rutas que tiene el grafo, por esto se trabajo con esto en búsqueda de generar mejores y peores casos. Además, por los gráficos 11 y 13 se puede ver que la cantidad de fábricas afecta también al rendimiento del algoritmo, lo que también se tuvo en cuenta al generar estos casos.

En este sentido, los peores casos ocurren cuando $f = c - 1$ y $r = \frac{(c+f)(c+f-1)}{2}$. La influencia de f se debe a que en la instancia inicial, el conjunto de rutas que el algoritmo chequeará si incluir o no en el grafo que esta generando será mucho mayor que si solo hubiera una fábrica. De este modo, mientras mayor sea f, mayor será la cantidad de aristas que el algoritmo comenzará a examinar. Por otro lado, en un grafo de

n nodos, la mayor cantidad de aristas posibles es $\frac{n(n-1)}{2}$ (el caso de un grafo completo) y en nuestro caso, $n = c + f$, por ende un grafo completo de nuestro problema tiene $\frac{(c+f)(c+f-1)}{2}$ aristas. Ahora bien, como ya se analizó, mientras más aristas hay en el grafo, más tarda en ejecutarse el algoritmo.

De este modo, se experimentaron los mejores y peores casos para observar su comportamiento. Para ello se minimizaron y maximizaron los valores de f y r, es decir, se fijaron los valores de $f = 1$ y $r = c + f - 1$ para el mejor caso y $f = c - 1$ y $r = \frac{(c+f)(c+f-1)}{2}$ para el peor caso. De este modo, se pueden observar estos resultados en la Figura 15 donde, en verde están los mejores casos y en rojo los peores.

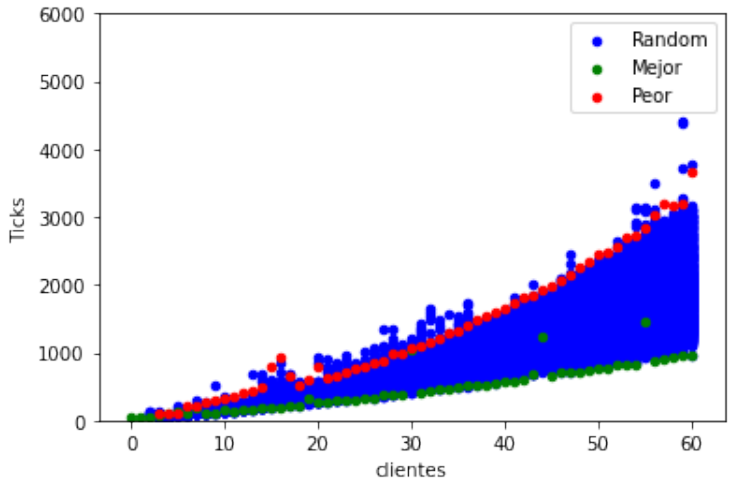


Figura 15: Gráfico de tiempo en función de cantidad de trabajos al cuadrado para instancias aleatorias.

Como se puede observar en el gráfico, los mejores casos acotan por debajo al resto de los puntos mientras que los peores los acotan superiormente. Esto quiere decir que los mejores casos acotan inferiormente los tiempos de ejecución de los generados aleatoriamente y los peores lo hacen superiormente. De este modo, se puede verificar que, efectivamente,estos son mejores y peores casos.

3.5. Conclusiones

Salvo una pequeña modificación en el código, sabemos que es un problema muy parecido al 2 (más aún porque ambos se implementaron con Prim y con un vecto como estructura para la cola de prioridad). Por lo tanto, los resultados observados son similares. Sin embargo, una importante diferencia a considerar es que en este problema son dos las variables que determinan el número total de nodos. De tal modo, se pudieron analizar por separado las influencias de estas variables en la complejidad y se pudo observar que además de que c influía (lo que era esperable pues determinaba la complejidad teórica), f también lo hacía (si crecía, el tiempo de ejecución también crecía), lo que fue de gran relevancia en la determinación de mejores y peores casos. Concluimos así que la modificación del algoritmo de Prim, si bien pequeña, era muy relevante ya que daba importancia a esta nueva variable respecto del problema 2. Sin embargo, como f está acotada por c, es sólo esta última la que define la complejidad teórica que es $\mathcal{O}(c^2)$. Cabe destacar que esta complejidad se pudo validar experimentalmente. udimos ver además que hay influencia de la cantidad de rutas donde al crecer esta, crecía el tiempo de ejecución. Teniendo en cuenta esto (y que el tiempo de ejecución también crecía cuando f lo hacía), pudimos generar mejores y peores casos.