DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nro. 2

08/05/2009 Algoritmos y Estructuras de Datos III

Integrante	LU	Correo electrónico
Dinota, Matías	076/07	matiasgd@gmail.com
Huel, Federico Ariel	329/07	federico.huel@gmail.com
Leveroni, Luciano	360/07	lucianolev@gmail.com
Mosteiro, Agustín	125/07	agustinmosteiro@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Aclaraciones generales

Antes de comenzar el análisis de cada ejercicio, cabe mencionar lo siguiente:

- La implementación de los 3 algoritmos se realizó en **lenguaje Java**, haciendo uso de las librerías estándar del mismo.
- Para el cálculo de tiempo de los algoritmos se utilizó la función **nanoTime()** de la clase System de Java. Con el fin de aumentar la precisión de las mediciones, se utilizó el comando **nice** para darle máxima prioridad a la tarea.
- El código fuente de los algoritmos aqui analizados se encuentran en los archivos *Dengue.java* (Ej 1), *Diamante.java* (Ej 2) y *RedAstor.java* (Ej 3).
- El código fuente de los programas encargados de hacer uso de los algoritmos y necesarios para compilar las aplicaciones son:
  - Ej 1: MainDengue.java, Dengue.java e InstanciaDengue.java.
  - Ej 2: MainDiamante.java, Diamante.java, InstanciaDiamante.java.
  - Ej 3: MainRedAstor.java, RedAstor.java, InstanciaRedAstor.java, Arista.java, AristaComparator.java, Dupla.java, DuplaComparator.java.
- Para la lectura y escritura de los datos se utilizaron clases provistas por el lenguaje Java. No se hará referencia a estos algoritmos ya que no resultan de interés para el trabajo aqui presentado.
- Los gráficos se realizaron con **GNUPlot**. En los casos considerados pertinentes, se utilizó una escala logarítmica con el fin de poder visualizar mejor los resultados.

Ejercicio 1: Dengue

Introducción

El primer ejercicio plantea el problema de optimizar el uso de litros de "Raid" para la fumigación de mosquitos óptima, requiriendo cubrir una determinada cantidad de zonas. Es relevante comentar que al utilizar una mayor cantidad de litros en una zona, esto no implica matar más mosquitos.

El ovjetivo diseñar un algoritmo que resuelva el problema planteado de manera polinomial. Para esto se utilizó la técnica de progamación dinámica, lo que implicó la realización de diversos pasos. Primero se corroboró que el problema cumpla con el principio de optimalidad. Una vez comprobado esto (condición necesaria para poder aplicar programación dinámica) y utilizando las ideas vistas en dicha demostración, se procedió a crear una función recursiva que retorne el resultado deseado. Luego se modificó dicha version por un algoritmo iterativo, siendo este un paso intermedio requerido para aplicar la técnica mencionada. El paso siguiente consistió en modificar el algoritmo de manera de evitar la repetición de cálculos innecesarios (aquí queda implicita la idea de programación dinámica), mejorando notablemente la complejidad de la función. Por último, como veremos más adelante, se optó por un cambio de técnica algoritmica pasando de un algoritmo que originalmente era *top-down* a uno *bottom-up*.

Algoritmo

En la presente sección presentaremos todos los pasos que se siguieron para lograr la implementación del algoritmo que resuelve el problema. Como se mencionó anteriormente, el problema será resuelto por medio de la técnica de programación dinámica por lo que presentaremos las demostraciones de todos los pasos que implica construir un algoritmo de esa manera. En primer lugar, se demostrará cómo aplica el principio de optimalidad en el problema dado. En segundo lugar, plantearemos una función recursiva que resuelva el problema y demostraremos su correctitud. Finalmente, mostraremos el algoritmo construido con programación dinámica, explicando detalladamente por qué resuelve el problema de manera más eficiente que la solución recursiva.

Principio de optimalidad

Demostraremos por qué vale el principio de optimilidad en este problema por el absurdo.

Sean n y l la cantidad de zonas y litros disponibles respectivamente y que las zonas se numeran de 1 a n. Suponemos que la cantidad de mosquitos muertos con l litros hasta la zona n es óptima y llamaremos P a dicha cantidad. Sea k la cantidad de litros usados en la zona n, suponemos que hasta la zona $n-1$ usando $l-k$ litros la cantidad de mosquitos muertos (Q) no es óptima y se intenta llegar a un absurdo.

Si hasta la zona $n-1$ la cantidad de mosquitos muertos no es óptima (para $l-k$ litros), existe otra forma de distribuir los $l-k$ litros entre las zonas 1 a $n-1$ que hace que la nueva cantidad, a la que llamaremos R, sea óptima (es decir, $R > Q$). Entonces tenemos que

$R + cantMM(n,k) > Q + cantMM(n,k) = P \Rightarrow R + cantMM(n,k) > P$

Esto es un absurdo ya que supusimos que P era la cantidad óptima. Este absurdo surge de suponer que la cantidad de mosquitos muertos hasta la zona $n-1$ con $l-k$ litros no era óptima. Entonces queda demostrado que vale el principio de optimalidad para el problema planteado.

Solución recursiva y demostración de correctitud

A partir de la idea presentada en la sección anterior se puede plantear la siguiente función recursiva para resolver el problema.

$f(0,l) = 0$

$$f(i,l) = max \begin{cases} f(i-1,l) \\ f(i-1,l-k) + MM[i][k] \quad \text{con } 1 \leq k \leq l \end{cases}$$

Siendo $MM[i][j]$ la matriz que contiene la cantidad de mosquitos muertos por litro, de cada zona. La solución al problema sería $f(n,l)$ con n la cantidad de zonas y l la cantidad de litros disponibles.

Para demostrar que la función presentada es correcta utilizaremos inducción en la cantidad de zonas manteniendo la cantidad l de litros fija. Es decir, queremos probar la siguiente proposición.

P(n): $f(n,i)$ es la cantidad máxima de mosquitos muertos hasta la zona n con i litros. $(0 \leq i \leq l)$

Caso Base ($P(1)$)

$f(1,i) = max_{0 \leq j \leq i}(MM[1][j])$ con $i \leq 1$

Entonces, como se puede notar, $f(1,i) = f(1,1)$ por lo que es trivialmente la cantidad máxima de mosquitos muertos de la primera zona con i litros.

Paso inductivo ($P(n) \Rightarrow P(n+1)$)

Supongo que $f(n,i)$ es la máxima cantidad de mosquitos muertos hasta la zona n con i litros ($0 \leq i \leq l$) y demuestro que $f(n+1,i)$ es la cantidad máxima hasta la zona $n+1$. Como $n+1 > 1$ tenemos que

$$f(n+1,i) = max_{0 \leq k \leq i}(f(n, i-k) + MM[n+1][k])$$

Como $i-k \leq l$ para cualquier $k \leq i$ puedo aplicar la hipótesis inductiva, es decir, $f(n,i-k)$ es la máxima cantidad hasta la zona n con $i-k$ litros. Entonces con $max_{0 \leq k \leq i}(f(n,i-k) + MM[n+1][k])$ obtengo la máxima cantidad de mosquitos muertos hasta la zona $n+1$ con i litros pues $f(n,i-k)$ es máxima y se calcula el máximo para todas las cantidades k de litros posibles para dicha zona. Entonces, como vale $P(1)$ y $P(n) \Rightarrow P(n+1)$ por principio de inducción queda demostrada la proposición.

La función presentada, a pesar de ser correcta, hace repetidas llamadas recursivas a valores ya calculados. Sin embargo, sirve de base para idear un algoritmo que use la técnica de programación dinámica para resolver este problema de forma más eficiente. En las posteriores secciones veremos en detalle cómo se relaciona dicha función con el algoritmo implementado.

Algoritmo utilizando programación dinámica

A continuación se encuentra el pseudocódigo del algoritmo que utiliza la técnica de programación dinámica para resolver el problema.

```
maxMMParcial[zonas][litros]

lleno la primer fila de maxMMParcial con 0

para i desde 1 hasta zonas
  para j desde 0 hasta litros
    max_parcial = maxMMParcial[i-1,j]
    k = 1
    mientras k <= j
      max_parcial = maximo(max_parcial, maxMMParcial[i-1,j-k] + MM[i][k])
    maxMMParcial[i][j] = max_parcial
```

Este algoritmo está basado en la función recursiva presentada en la sección anterior, pero usando una estrategia *Bottom-up* para calcular el resultado. Dicha estrategia se basa en el uso de una matriz para almacenar los resultados ya calculados y, a partir de ellos, construir las distintas etapas de la solución. Esta matriz tiene la siguiente forma:

litros

$$zonas \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & f(1,1) & f(1,2) & \cdots & f(1,l) \\ 0 & f(2,1) & f(2,2) & \cdots & f(2,l) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & f(n,1) & f(n,2) & \cdots & f(n,l) \end{pmatrix}$$

En el algoritmo presentado cada fila de esta matriz se construye a partir de la fila anterior, por lo que la primer fila (que representa el caso base de f) debe ser 0 al comienzo del procedimiento. Además, en este cálculo también es necesario utilizar los valores de la matriz de entrada (MM) que contiene la cantidad de mosquitos muertos según la cantidad de zonas y litros. Esto se corresponde a la definición recursiva de la función f ya que el valor que retorna f para cada zona i se construye a partir de la zona $i-1$.

Como se puede apreciar en el pseudocódigo cada elemento de la matriz se construye de la siguiente manera:

$$maxMMParcial[i][j] = max \begin{cases} maxMMParcial[i-1][j] \\ maxMMParcial[i-1][j-k] + MM[i][k] \quad \text{con } 1 \leq k \leq j \end{cases}$$

Esta definición muestra claramente la relación que existe entre el algoritmo que utiliza programación dinámica y la función que resuelve el problema de manera recursiva. Cada posición de la matriz se calcula de la misma manera que f, por lo que, si el caso base es el mismo, se obtendrán los mismos resultados. Por esta razón, al terminar de calcular toda la matriz (hasta la posición n y l) se obtiene $f(n,l)$, es decir, la solución para la instancia del problema.

Cálculo de los litros por zona

Una vez obtenida la matriz que contiene los valores óptimos para las distintas instancias del problema. Se implementó un algoritmo para utilizar estos valores de manera tal de encontrar la cantidad de litros que debían emplearse en cada zona para finalmente obtener el valor óptimo ya calculado.

Este algoritmo consistió en, utilizando el principio de optimalidad, recorrer la matriz de la última fila hacia la primera, observando cómo cambiaría el valor en la matriz de una zona a otra utilizando una determinada cantidad de litros y cuál fue el cambio observado. Al coincidir el resultado observado con el esperado, se encuentra la cantidad de litros que deben ser utilizados en esa zona y se guarda en un array. Una vez recorridas todas las filas de la matriz, se posee un array con los litros que deben usarse en cada zona para conseguir el resultado óptimo.

```
recorrer matriz maxMMparcial desde la ultima fila a la primera
  Buscar en cada fila (i) en MM los listros que cumplan con que:
    maxMMparcial[i][litrosQueNoUse] - los mosquitos muertos en la zona i con esos litros
    = a maxMMparcial[i-1][litrosQueNoUse - litros que acabo de usar].
  guardar en litrosUsadosEn[i] los litros encontrados para dicha zona.
```

Complejidad

En la presente sección se calculará la complejidad en el peor caso del algoritmo *fumigar* basándose en el modelo de cómputo uniforme. Esto se debe a que este ejercicio no se basa en el tamaño de los elementos de la matriz recibida como parámetro, sino en las dimensiones de dicha matriz, por lo que no se trabajará con valores muy altos. Para realizar los cálculos de manera más clara, se dividirá el analisis en el estudio de las dos partes que conforman la función: *fumigación* y *cálculo de litros por zona*

En la función *fumigación* todas las operaciones y asignaciones utilizadas se logran en tiempo constante. Esto se debe al mencionado modelo de cómputo utilizado, en donde las operaciones presentes en el algoritmo no dependen del tamaño de los parámetros de entrada. De esta manera, la complejidad de esta función estará supeditada al comportamiento de los ciclos que contiene. Llamaremos *a b* y *c* a los ciclos, donde *a* contiene a *b* y este último contiene a *c*. Se puede apreciar claramente que b itera m veces (donde m es la cantidad de litros) y que en cada iteración de b, se realiza una vez el ciclo c. El límite de este ciclo esta dado por la variable de iteración de b (en el pseudocódigo se llamada j), de manera que, en cada iteración de b, la cantidad de vueltas del ciclo c aumenta en uno, comenzando desde cero (valor de inicio de la variable j) hasta m. La complejidad del ciclo b sera entonces:

$$\sum_{j=0}^{m} j = \sum_{j=1}^{m} j = (m+1)m/2 \ \in O(m^2)$$

El ciclo analizado se encuentra dentro de a, este último se recorre n veces (donde n es la cantidad de zonas) debido a que su variable de iteración (en el pseudocódigo llamada i) es inicializada en cero y aumentada en uno hasta llegar al valor n. Finalmente podemos concluir que la complejidad de la función *fumigación* es:

$$T(n,m) = n((m+1)m/2) \ \in O(nm^2)$$

Por otro lado, analizaremos la complejidad en peor caso de la otra parte del algoritmo: *cálculo de litros por zona*. En cuanto a los costos de las operaciones matemáticas y asignaciones, sucede lo mismo que en el caso anterior por lo que nuevamente el estudio de la complejidad se concentrará en los ciclos. En este caso tenemos dos ciclos. Se puede apreciar facilmente que el ciclo externo realiza n iteraciones ya que la variable i utilizada para iterar es inicializada en n y es decrementada hasta llegar a 1. Como muestra el pseudocódigo, el limite del ciclo interno esta dado por una condición por lo que la cantidad de iteraciones que realize varía cada vez que es ejecutado. Lo que si se puede ver es que, en total (es decir, para las n iteraciones del ciclo externo), el ciclo será recorrido m veces. Esto se debe a que en caso de cumplir con la condición mencionada, la variable de iteración del ciclo j es decrementada en 1, y nunca es reiniciada. Como dicha variable es inicializada con la cantidad de litros y en cada vuelta del ciclo interno se corrobora que su valor sea mayor o igual que cero, podemos concluir que dicho ciclo será recorrido m veces en total (ya que si se cumple la condición, j es decrementado, y si no se satisface la guarda, no se entra al ciclo). Por estos motivos la segunda parte del algoritmo de *fumigar* tiene complejidad $T(n) \ \in O(n+m)$.

De esta manera, como la complejidad de la primer parte de la función pertenece a $O(nm^2)$, la segunda pertenece a $O(n+m)$ y $O(n+m) \ \in O(nm^2)$, concluimos que la complejidad en peor caso del algoritmo *fumigar* es

$$T(n,m) \in O(nm^2)$$

Finalmente estudiaremos la complejidad en función del tamaño de la entrada. Sea t el tamaño de la entrada, n, m y M la cantidad de zonas, litros y la matriz pasados como parámetros respectivamente. Tenemos entonces que:

$$t = \log(n)+\log(m)+\sum_{i=1}^{n}\sum_{j=1}^{m}\log(M_{i,j}) > \log(n)+\log(m)+\sum_{i=1}^{n}\sum_{j=1}^{m}1 > \log(n)+\log(m)+nm > nm$$

$$\implies \text{como } T(n,m) \in O(nm^2) \text{ y } nm^2 < (nm)^2 < t^2 \implies T(t) \in O(t^2)$$

Análisis de resultados

Con el fin de analizar la complejidad de los algoritmos propuestos (fumigación y cálculo de litros por zona) se realizaron varias pruebas. Estos experimentos están orientados a estudiar el tiempo de ejecución del mismo en relación a los datos de entrada, comparando el costo real de los algoritmos junto con su complejidad teórica calculada. Para cada algoritmo se verán 3 gráficos distintos. El primero de ellos servirá para analizar la complejidad con respecto a los litros, es decir, fijando la cantidad de zonas. Reciprocamente, el segundo será en base a la cantidad de zonas con una cantidad fija de litros. Por último, el tercer gráfico en 3 dimensiones mostrará como se comporta el algoritmo considerando ambas variables en conjunto.

A continuación se describirán las características de las instancias utilizadas para las pruebas así como las particularidades de cada experimento. Dichas cuestiones son comunes al analisis de ambos algoritmos por lo que no haremos referencia a ninguno en particular. Más adelante analizaremos resultados de estos experimentos aplicado a cada uno de los algoritmos.

Para las instancias de prueba se utilizó la función *random* del lenguaje para generar matrices de *mosquitosMuertos*) de tamaño $zonas * litros$ con valores aleatorios comprendidos entre 1 y 10000 con el fin de garantizar que las pruebas resulten lo suficientemente genéricas como para poder obtener resultados "interesantes". Cabe mencionar que no se harán estudios sobre instancias particulares ya que, como se mencionó antes, el costo de ejecución del algoritmo de *fumigación* no depende de los valores de la matriz de entrada que conforma cada instancia.

En primer lugar, se hicieron cálculos utilizando como variable la cantidad de zonas a fumigar, fijando en 10 los litros disponibles. El rango de zonas analizados es desde 1 hasta 10000 zonas, de a intervalos de tamaño 10. La elección de estos valores fue producto del análisis de 2 cuestiones importantes: que números permitirían tomar muestras utiles para la realización de los gráficos y cual era la máxima cantidad de datos que nuestra aplicación podría manejar, acotada por el tiempo disponible para realizar las pruebas y las limitacion propias de la máquina en cuestión. Empiricamente, se comprobó que dicho rango de zonas permitía correr la aplicación en un tiempo razonable y al mismo tiempo proveer datos interesantes para el análisis en cuestión. Del mismo modo, fijar en 10 los litros aseguró que sea la cantidad de zonas el parámetro ponderante en el cálculo de tiempo de ejecución.

La segunda prueba resulta muy similar a la anterior en cuanto al rango de valores utilizados. La diferencia radica en que ahora el parámetro fijado es la cantidad de zonas en vez de la cantidad de litros, variando los litros en el rango 1 hasta 10000, de a intervalos de tamaño 10 (al igual que en la prueba anterior).

La tercer y última prueba consistió en analizar el algoritmo considerando ambas variables en conjunto, de modo de poder comparar el tiempo de ejecución para distintos valores de zonas y litros junto con la complejidad teórica que depende de estos dos parámetros. Para

esta prueba se tomaron valores de zonas y litros entre 1 y 1000, de a intervalos de tamaño 100 (10 valores posibles). Luego, se calcularon los tiempos de ejecución de todas las permutaciones de los 10 elementos pertenecientes a dicho conjunto de manera tal de obtener resultados para las combinaciones de litros y zonas dentro del intervalo mencionado.

A continuación se presentarán los gráficos realizados para cada algoritmo utilizando estas 3 pruebas descriptas.

Fumigación

Como se vió anteriormente, la primer parte del problema consistía en calcular la máxima cantidad de mosquitos muertos para una instancia determinada. Los siguientes gráficos muestran el comportamiento de este algoritmo en base a los dos parámetros de entrada de manera independiente:

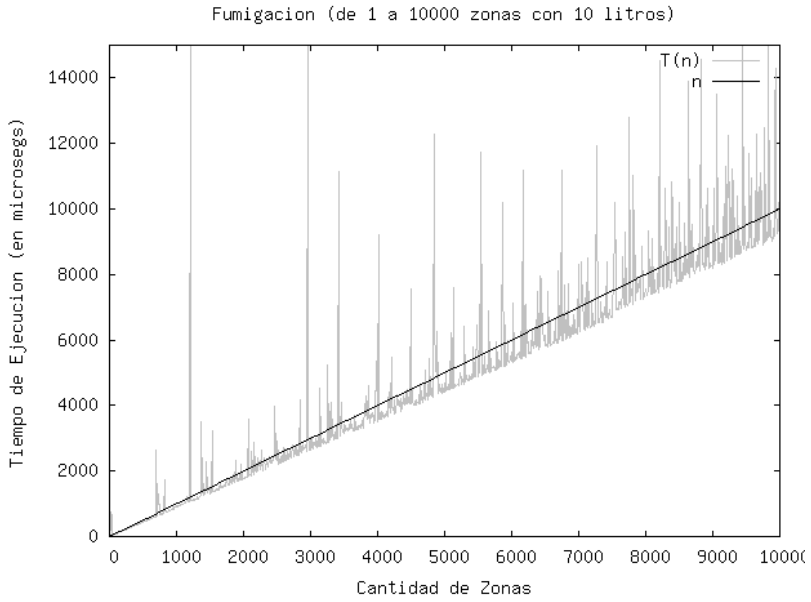


Figura 1.1

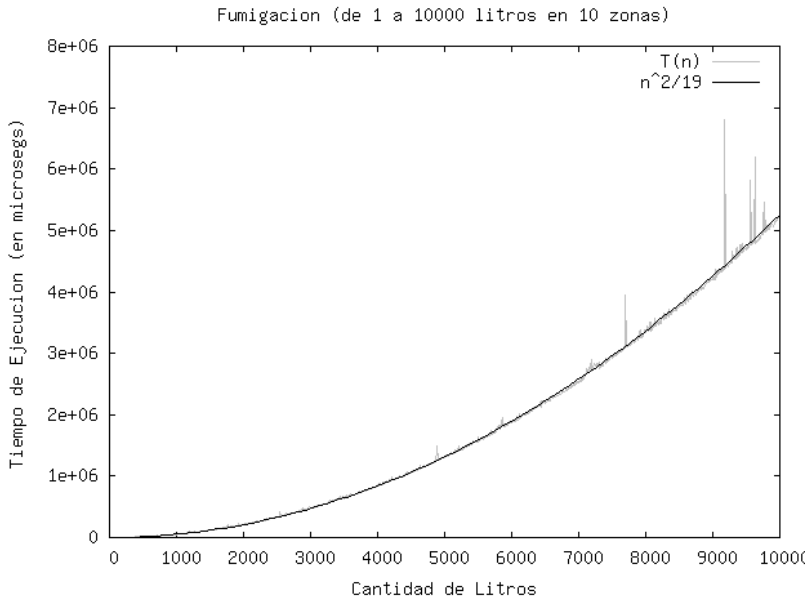


Figura 1.2

Ambos gráficos permiten observar algunas cuestiones interesantes. La Figura 1.1 permite observar como el comportamiento del algoritmo de *fumigación* con respecto a la cantidad de zonas se asimila a una función lineal, hecho que se corresponde con la complejidad calculada anteriormente. Del mismo modo, la Figura 1.2 muestra como la función T(n) que representa el tiempo de ejecución se comporta prácticamente igual que la función $f(n) = n^2/19 \in O(n^2)$. Este hecho demuestra empíricamente que la complejidad con respecto a la cantidad de litros es cuadrática al realizar la fumigación tal como se había analizado anteriormente.

Otras cuestión observable son los "picos" y variaciones abrutas en ambos gráficos. En principio, esto no debería ocurrir ya que la complejidad no depende de la forma de la entrada como se mencionó anteriormente. El motivo de esto radica simplemente en los errores de precisión en la medición de las operaciones propios de las limitaciones de la máquina de cómputo utilizada. Los tiempos de ejecución del algoritmo son mucho mayores en la Figura 1.2 lo que resulta en una mayor estabilidad en los cálculo de tiempo, en contraste con la Figura 1.1 donde hay mayor impresición porque dicho gráfico refleja cambios de valores más abrutos.

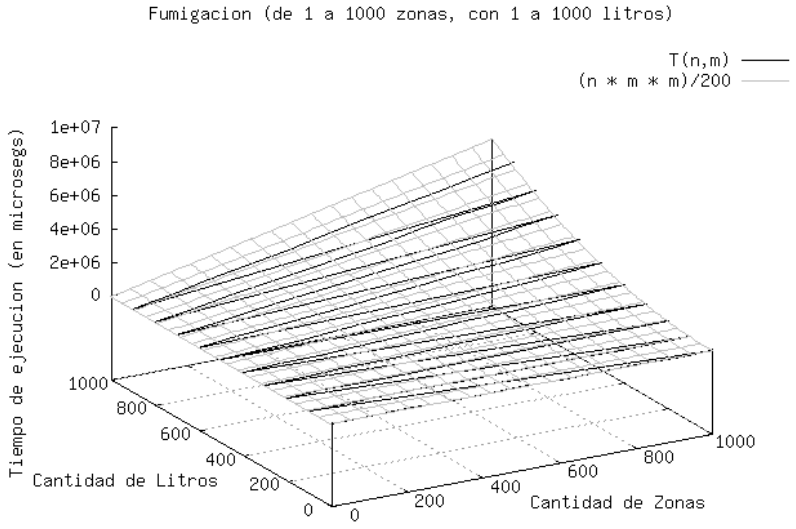


Figura 1.3

Esta última figura permite ver como el algoritmo en su totalidad realmente se comporta de la forma esperada. Para esto basta comparar los tiempos de ejecución al variar ambos parámetros (litros y zonas) junto con la función $f(n, m) = n * m^2/200 \in O(n * m^2)$ viendo que ambas "mantas" coiniciden. Este ultimo hecho constituye la última prueba de que empirícamente el algoritmo funciona tal como se esperaba.

Cálculo de Litros Por Zona

En esta sección analizaremos el comportamiento del otro algortimo propuesto que se encarga de dar parte de la solución del problema: el algoritmo encargado de calcular los litros a utilizar por cada zona. Los siguientes gráficos muestran el comportamiento del mismo en base a las dos variables en cuestión de forma independiente:

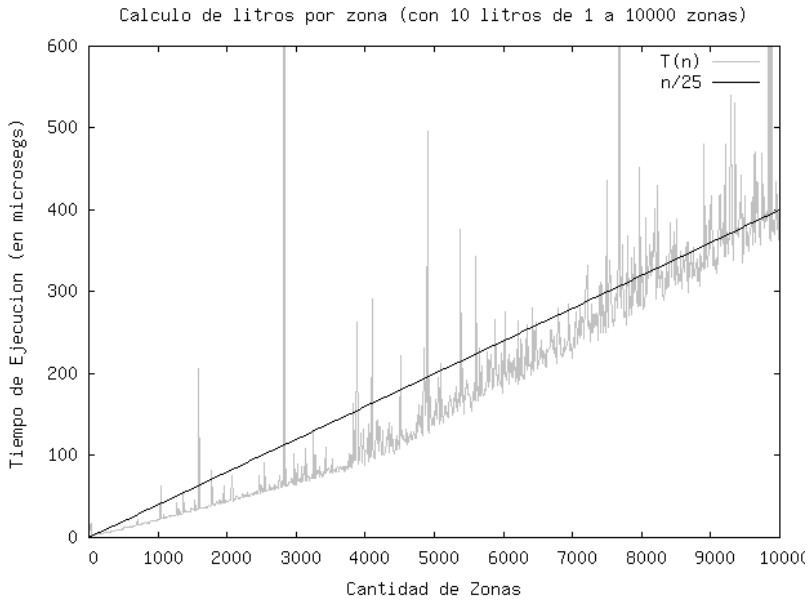


Figura 2.1

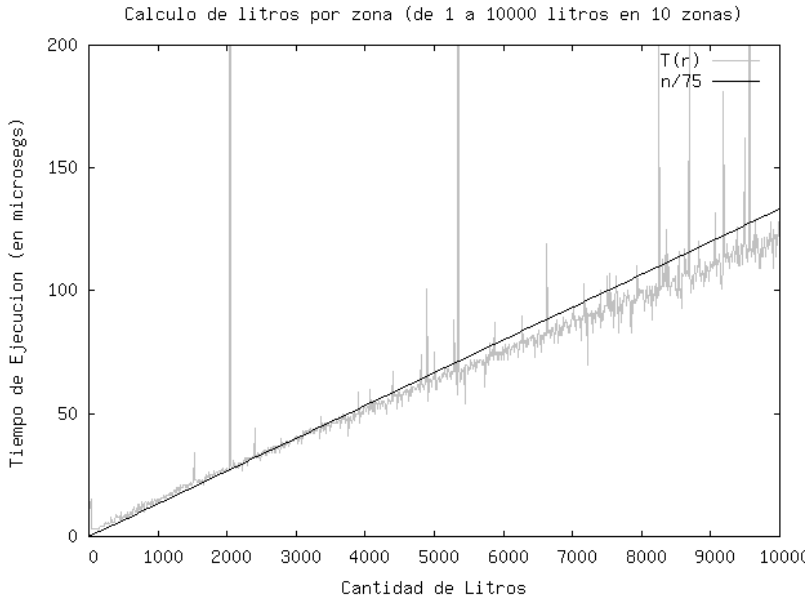


Figura 2.2

Como se puede observar, en ambos casos, el comportamiento del algoritmo es el esperado. La Figura 2.1 muestra como el cálculo de litros por zona se "corresponde" con el gráfico de la función lineal $f(n) = n/25$ al variar la cantidad de zonas y fijar los litros en 10. Asímismo, la Figura 2.2 se comporta de la forma $f(n) = n/75$ al variar los litros y fijar las zonas. Esto comprueba la linealidad en complejidad teórica analizada previamente para ambas variables. Un punto interesante a observar es que las constantes que acompañan a estas funciones difieren, siendo la constante relacionada al primer gráfico (1/25) 3 veces mayor que la otra (1/75). Al tratarse de instancias de igual tamaño (en ambos casos, matrices desde

$1 * 10$ hasta $10000 * 10$), se puede concluir que la *performance* de este algoritmo depende en mayor medida de la cantidad de zonas que de la cantidad de litros. En otras palabras, para instancias de igual tamaño, un caso demora 3 veces más el otro. Si bien se intentó buscar una explicación a este fenómeno, se concluyó que resulta muy complejo observar cual es la causa de esta particularidad, principalmente porque se está trabajando con instancias aleatorias con parámetros arbitrarios.

Con respecto a la irregularidad del gráfico, cabe notar que, a diferencia del algoritmo anterior, este algoritmo no realiza siempre la misma cantidad de operaciones para instancias de igual tamaño tal como se vió en la sección de análisis de complejidad, ya que el recorrido inverso de la matriz resultante depende justamente de los valores que esta contenga que dependen a su vez de los valores de la matriz de entrada *mosquitosMuertos* (ver sección *Algoritmo*). Por este motivo, debido a que se trata de instancias aleatorias, es esperable que no se obtenga un comportamiento "tan" similar a la función lineal propuesta como ocurre para el primer algoritmo.

Por último, al igual que para el algoritmo anterior, se presenta el siguiente gráfico de 3 dimensiones:

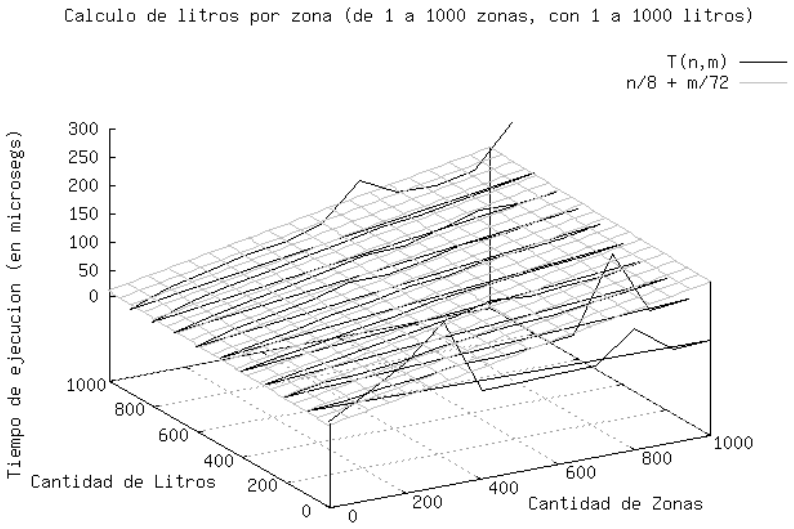


Figura 1.3

Al igual que antes, este gráfico permite ver la complejidad de ambos parámetros en conjunto. Cabe notar al tratarse de un algorimto más veloz que el primer caso, los errores de cálculo de tiempo son aún mayores por lo cual no resulta simple obtener una conclusión demasiado fiable. Sin embargo, se puede ver que la función $f(n, m) = n/8 + m/72 \in O(n+m)$ resulta una estimación aceptable para $T(n, m)$. Al igual que antes, podemos concluir que la complejidad teórica total del algoritmo calculada tiene directa relación con los resultados obtenidos.

Conclusiones

Este ejercicio srivió fundamentalmente para descubrir y profundizar las caracteríticas principales de la programación dinómica. Los aspectos más relevantes estudiados fueron la

comparación en complejidad y dificultad de implementación entre un algoritmo que actúa de manera ingenua y otro que utiliza la técnica de programacién dinámica y lo determinante que es la utilización del principio de optimalidad en este tipo de ejercicios.

En términos de complejidad, el algoritmo tiene una gran eficiencia respecto de su 'competidor'. Claro está que el algoritmo recursivo con el que fue comparado no propone ningún tipo de eficiencia, no obstante es el más natural de imaginarse, lo que también muestra cómo puede optimizarse un algoritmo con complejidad exponencial a otro polinomial utilizando alguna técnica algorítmica aplicable.

Si bien la función recursiva utilizada mostró gran deficiencia, sirvió para tener la idea del algoritmo y a partir de ella, comenzar a pensar en còmo aplicar programación dinámica.

En el análisis de los resultados, se pudo ver como se evitan cálculos utilizando resultados anteriores y se pudo comprobar empíricamente que el algoritmo cumple con la complejidad calculada. Los gráficos también mostraron cómo en este ejercicio en particular, el tiempo de ejecición dependía en mayor medida de la cantidad de zonas que de la cantidad de litros que se tengan aunque la complejidad depende de ambos parámetros $o(n * m^2)$.

Respecto de la dificultad de la implementación, quedò absolutamente claro que la solución recursiva era intuitiva y sencilla mientras que el algoritmo de programación dinámica requiere de un pensamiento más elavorado. Sin embargo, el ejercicio mostró que entendiendo como aplica el principio de optimalidad al problema, las ideas se vuelven mucho más claras y menos desagradables. En un comienzo el principio de optimalidad parecía sencillo de entender. Sirvió para construir una matriz que guardara los valores óptimos hasta el momento que eran usados para calcular los siguientes. Una vez obtenido el resultado óptimo para el problema planteado, empezó la parte rebuscada del uso del principio de optimalidad, pero también la más interesante.

No fue para nada sencillo ver como usar los valores calculados, y el principio para calcular los litros que debían emplearse en cada zona para obtener el mejor resultado. Ésta fue la parte más complicada del algoritmo, comprender cómo volver por la matriz para obtener los datos deseados que nos interesaban.

En este punto fue donde el principio de optimalidad necesitó ser realmente comprendido y mostró su importancia en el algoritmo de programación dinámica.

Desde el punto de vista de la demostración también resulta crucial tener claro el principio de optimalidad y demostrarlo ya que en este se basa la programación dinámica. El ejercicio mostrò una relaciòn entre el principio de optimalidad y la posibilidad de la utilizaciòn de inducción para la demostración del algoritmo. Debido a su característica, y teniendo en cuenta que la técnica la utiliza para llegar a la solución, se vuelve natural pensar en partir de un caso base y luego mostrar que se mantiene su correctitud cuando se calcula un valor posterior.

En general, podemos afirmar que los algoritmos de programación dinámica pueden mejorar muchísimo la complejidad sobre otros algoritmos que utilizan otro tipo de técnicas, sin resultar excesivamente complicados. Esa dificultad pasa principalmente por comprender a fondo como aplica el principio de optimalidad en dicho problema y que además, una vez logrado esto, la demostración de correctitud se puede volver màs sencilla.

Ejercicio 2: Diamante

Introducción

El objetivo del siguiente ejercicio es diseñar un algoritmo que, dado un grafo, encuentre el menor subgrafo inducido isomorfo a un diamante, en caso de que éste exista. Un diamante puede definirse como un grafo K_4 menos una arista. Este algoritmo está basado en una propiedad sobre diamantes (enunciada en el Anexo A) que, además de ser usada para resolver el problema, será demostrada de manera formal.

En un principio, cuando se comenzó a idear el algoritmo, surgió la idea de utilizar solamente la propiedad enunciada para resolver el problema. Es decir, para toda vecindad de cada nodo, estudiar cada componente conexa para determinar si la misma es completa. Sin embargo, esto sólo aportaba información sobre las componentes conexas que formaban un diamante, pero no brindaba detalles sobre los nodos que formaban el diamante, y no aseguraba que este fuera mínimo para dicha componente conexa. Por esta razón, se optó por utilizar un método más complejo que, aunque se basa en la propiedad mencionada, obtiene de cada componente conexa no completa (existe diamante) los nodos mínimos que forman un diamante. Esto se realiza calculando el mínimo nodo que puede formar un diamante y, a partir de este, construir el mínimo probando las posibles combinaciones que construyan un diamante. De esta manera se pueden obtener los diamantes mínimos de cada componente conexa no completa de una determinada vecindad y así determinar el mínimo de la misma. Finalmente, se comparan los diamantes obtenidos de cada vecindad para obtener el mínimo diamante del grafo. Este algoritmo será explicado en detalle, se mostrará su correctitud y se presentará la implementación y pruebas de ejecución del mismo en posteriores secciones.

Demostración de la parte a

A continuación se presentará la demostración de la siguiente propiedad:

Sea G un grafo de n vértices y m aristas, G no contiene un subgrafo diamante como subgrafo inducido si y solamente si para todo vértice v, su vecindad $N(v)$ se divide en componentes conexas completas.

Llamaremos p a la proposición "G no contiene un subgrafo diamante como subgrafo inducido" y q a la proposición "para todo vértice v, su vecindad $N(v)$ se divide en componentes conexas completas". Dividiremos la demostración en dos partes: la implicación y la recíproca.

- $p \implies q$

  Por propiedades de la lógica $p \implies q$ equivale a mostrar $-q \implies -p$. De esta manera, probaremos aquí que si existe un vértice v tal que su vecindad $N(v)$ contiene una componente conexa no completa entonces G contiene un subgrafo diamante como subgrafo inducido.

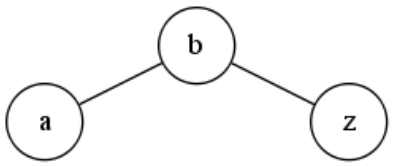
  Primero probaremos la siguiente propiedad (propiedad 1): Sea v un vértice de G y sea $N(v)$ su vecindad asociada. Si $N(v)$ tiene a K_3 menos una arista (lo llamaremos $K_3 - 1$) como subgrafo inducido, entonces G contiene un diamante como subgrafo inducido.

  Esto se debe a que, al pertenecer $K_3 - 1$ a la vecindad de v, el grafo original contiene a $K_3 - 1$ uniendo todos sus vertices con v, esto implica agregarle un nodo (v) y tres aristas

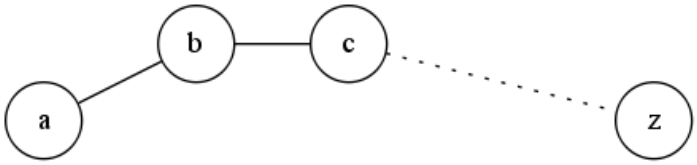
(ya que v es adyacente a los tres nodos que conforman el $K_3 - 1$ porque pertenecen a su vecindad). Como $K_3 - 1$ tiene dos aristas entonces agregando un nodo y sumando tres aristas tengo un $K_4 - 1$ (definición de diamante).

Viendo esto, basta probar que si existe un vértice v tal que su vecindad $N(v)$ tiene una componente conexa no completa entonces dicha componente contiene un $K_3 - 1$. Para esto, mostraremos ahora que todo grafo G conexo y no completo contiene un $K_3 - 1$ (propiedad 2).

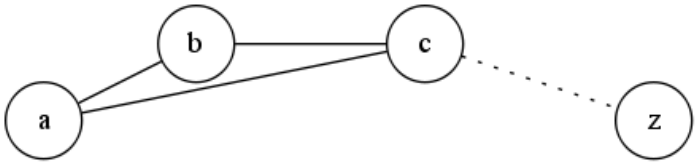
Como G no es completo pero si conexo, existen dos nodos a y z tal que no son adyacentes y existe un camino entre ellos. Sea b un nodo adyacente a a que pertenece al camino entre a y z. b es distinto de z ya que si fueran iguales a y z serían adyacentes y eso es absurdo por hipotesis. Ahora tomamos un nodo adyacente a b que también pertenezca al camino entre a y z al que llamaremos c. Este nodo puede ser igual a z (en caso de que el camino tenga longitud dos) o un nodo intermedio en el camino entre a y z. Si c es igual a z entonces me puedo formar un $K_3 - 1$ ya que a es adyacente a b, b es adyacente a z y z no es adyacente a a por hipotesis:



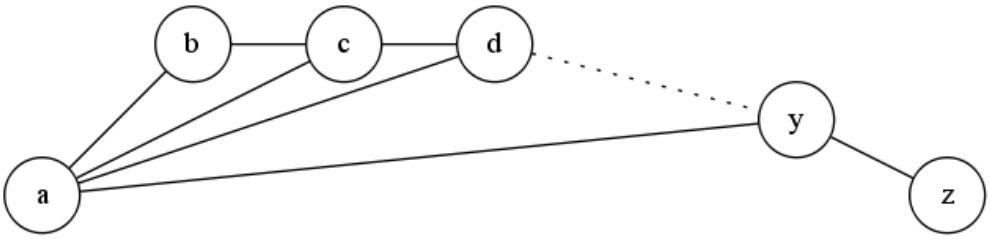
Si c es distinto de z entonces tenemos dos casos: si c no es adyacente a a entonces podemos formar un $K_3 - 1$ con dichos nodos:



Si c si es adyacente a a, entonces no podemos formar un $K_3 - 1$ entre a b y c:



En este último caso tomamos como nuevo nodo b a c (es adyacente a a y pertenece al camino entre a y z) y como nuevo nodo c a un nodo adyacente a c que pertenezca al camino entre a y z y todavia no haya sido recorrido (siguiendo el orden alfabetico seria el nodo d). Nuevamente caemos en el caso anterior, donde la posibilidad de poder armar un $K_3 - 1$ depende de si d es adyacente o no a a. Si no es posible obtener un $K_3 - 1$ con el nodo d (debido a que d es adyacente a a), seguiremos avanzando los dos nodos pertenecientes al camino entre a y z hasta llegar a que el primero de los dos es el nodo anterior a z (lo llamaremos y) del camino y el segundo nodo es igual a z. En este caso, siempre es posible armar un $K_3 - 1$ ya que z no es adyacente a a por hipotesis:

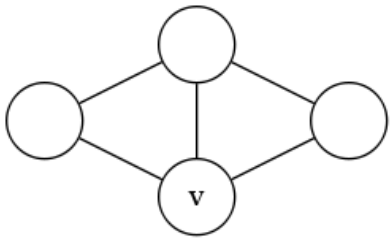


Finalmente hemos probado que si existe un vértice v de G tal que su vecindad $N(v)$ tiene una componente conexa no completa entonces dicha componente contiene un $K_3 - 1$ (por 2), y esto implica que G contiene un subgrafo diamante como subgrafo inducido (por 1).

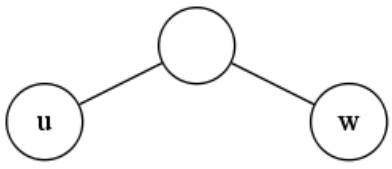
- $q \Longrightarrow p$

  Por propiedades de la lógica $q \Longrightarrow p$ equivale a mostrar $-p \Longrightarrow -q$. De esta manera, probaremos aquí que si G contiene un subgrafo diamante como subgrafo inducido entonces existe un vértice v tal que su vecindad $N(v)$ contiene una componente conexa no completa.

  Sea G un grafo que contiene un subgrafo diamante como subgrafo inducido. Sea G_D el subgrafo diamante de G:



  Si tomo uno de los dos nodos de G_D de grado tres, que llamaremos v, su vecindad $N_{G_D}(v)$ sera:



  Podemos ver entonces que $N_{G_D}(v)$ tiene una única componente conexa y esta no es completa ya que los dos nodos de grado dos del diamante (en la figura llamados u y w) no son adyacentes.

  Sea v uno de los dos nodos de grado tres del subgrafo diamante de G y u y w los dos nodos de grado dos de dicho diamante. Si tomo la vecindad $N(v)$ de v, como u y w son adyacentes a v (pertenecen al diamante) entonces pertenecen a $N(v)$. Ademas como u y w son los nodos de grado dos del subgrafo diamante entonces no son adyacentes en G. Como u y w no son adyacentes en G tampoco lo serán para ningun subgrafo inducido

de G, en particular para $N(v)$. De esta manera la componente conexa de $N(v)$ que contiene al diamante no es completa.

Concluimos entonces que la propiedad es cierta ya que existe un v para el cual se cumple.

Algoritmo

A continuación se presenta el pseudocódigo del algoritmo para resolver el problema propuesto.

```
crear grafo representado por array de lista de adyacencias (llamado 'adyacencias')
excepto para los nodos de grado 1 y 0

armar la matriz de adyacencias de ese grafo

para cada nodo (que llamaremos superNodo) del grafo
  si el tamaño de la lista de adyacencia de superNodo es mayor que 3
    crear la vecindad del superNodo (crearVecindad)
    buscar el diamante minimo de esa vecindad (buscarDiamanteMinimoEnVecindad)
    y agregarlo a diamantesMinimos (si es que hay alguno)

  si hay algun diamante en diamantesMinimos retorno minimo(diamantesMinimos)

crearVecindad(superNodo)
  para cada nodo k de adyacencias[superNodo]
    para cada nodo j de adyacencias[k]
      si nodo j es adyacente a superNodo
        agregar nodo j a listaVecindad
    vecindadDeSuperNodo[k] = listaVecindad
  devolver vecindadDeSuperNodo (grafo representado por array de listas de adyacencia)

buscarDiamanteMinimoEnVecindad(vecindadDeSuperNodo)
  recorrer cada componente conexa de vecindadDeSuperNodo por DFS
  sumando los grados de los nodos recorridos (sumaGradosCompConexa) y
  guardando en nodosCompConexa los nodos recorridos y
  en nodoMinimoCompConexa el nodo minimo de la componente conexa

  si sumaGradosCompConexa != a la cantidad de nodos
  del grafo completo de los nodos de la comp conexa
    si diamanteMinimo no existe
      diamanteMinimo = diamante minimo de comp conexa actual
      (buscarDiamanteMinimoDeCompConexa)
    sino
      si nodoMinimoCompConexa <= el minimo nodo de diamanteMinimo
        diamante = diamante minimo de comp conexa actual
        (buscarDiamanteMinimoDeCompConexa)
        si diamante existe y es mas chico que diamanteMinimo
```

```
          diamanteMinimo es ese diamente

  devolver diamanteMinimo (puede ser nulo en caso de que no haya diamante)

buscarDiamanteMinimoDeCompConexa(superNodo, vecindadDeSuperNodo, nodosCompConexa)
  nodo1 = superNodo
  nodoMinimo = nodo minimo de nodosCompConexa con grado menor
  que cantidad de nodos de la componente conexa - 1

  tuplaMin = <nodo mas grande del grafo, nodo mas grande del grafo>
  para cada nodo (adyacenteANodo2) de vecindadDeSuperNodo[nodo2]
    para cada nodo (nodoCandidato) de vecindadDeSuperNodo[adyacenteANodo2]
      si nodoCandidato no es adyacente a nodo2
      y <nodoCandidato, adyacenteANodo2> < tuplaMin
        tuplaMin = <nodoCandidato, adyacenteANodo2>

  nodo3 = primero(tuplaMin)
  nodo4 = segundo(tuplaMin)

  diamanteMinimo = ordenar(nodo1, nodo2, nodo3, nodo4)

  devuelvo diamanteMinimo
```

Como se mencionó anteriormente, este algoritmo está basado en la propiedad demostrada en el punto a, es decir, para cada nodo del grafo estudia las componentes conexas de la vecindad de dicho nodo para determinar si existe un diamante. El primer paso del algoritmo consiste en cargar las listas de adyacencias provistas en el archivo de entrada a un estructura que llamada *adyacencias*, que no es más que un arreglo de estas listas, donde cada posicion i está asociada a un nodo i. Luego, se eliminan todos los nodos de grado 0 ó 1, ya que estos no pueden formar parte de ningún diamante y eliminarlos no afecta a ningún diamante del grafo. Esto se debe a que cualquier nodo que pertenezca a un diamante tiene por lo menos grado 2. Para poder eliminar los nodos y poder reducir efectivamente el tamaño del grafo se realizó el siguiente prodecimiento: se creo un nuevo arreglo de adyacencias del tamaño del nuevo grafo (previamente se contaron la cantidad de nodos de grado mayor a 2) y, mediante el uso de arreglos, se realizó un mapeo entre los nodos del grafo original y los valores de los nodos del nuevo grafo para mantener la coherencia de la estructura. Como se observa en el pseudocódigo, al retornar el diamante en cuestión, se realiza un mapeo inverso restaurando los valores de los nodos originales, manteniendo así la consistencia de la entrada de los nodos de entrada. Por último, se creará la matriz de adyacencia que resultará necesaria verificar si dos nodos son adyacentes en tiempo constante (como veremos mas adelante, condición necesaria para cumplir con los requisitos de complejidad).

Luego, el algoritmo comienza a chequear por cada nodo (en adelante llamaremos *superNodo*) su vecindad para encontrar componentes conexas no completas y así determinar donde se encuentra el diamante. La función *crearVecindad* se encarga de construir la vecindad (que es un grafo representado con una lista de adyacencia) de cada *superNodo* a partir de la lista *adyacencias*, es decir, por cada elemento k de la lista de adyacencia de *superNodo* se reco-

rre la lista de adyacencia del elemento k agregando en la vecindad los elementos que están relacionados con *superNodo*.

Una vez obtenida la vecindad (*vecindadDeSuperNodo*), se comienza a revisar las componentes conexas de la misma para verificar si son completas a través de la función *buscarDiamanteMinimoEnVecindad*. Con este fin, se utiliza la técnica *Depth First Search (DFS)* para recorrer cada componente conexa almacenando la suma de los grados de cada nodo recorrido así como el nodo mínimo encontrado (*nodoMinimoCompConexa*) y la lista de nodos por los que se avanzó (*nodosCompConexa*). Al terminar de recorrer cada componente conexa, se verifica si la misma es completa o no. En caso de que lo sea, se sabe que ninguno de los nodos de esa componente conexa forma un diamante[1], por lo que se descarta y continúa verificando las siguientes componentes. En caso contrario, se procede a encontrar los nodos que forman el diamante[1] mínimo para esa componente conexa por medio de la función *buscarDiamanteMinimoDeCompConexa*. Esta función busca primero el nodo mínimo de la componente conexa que posea grado menor a $n-1$ siendo n la cantidad de nodos de esa componente conexa (identificado como *nodo2*). Luego, toma la lista de adyacencia de ese *nodo2* y para cada nodo k de dicha lista, recorre todos los elementos j de su lista de adyacencia para determinar si forma un grafo $K_3 - 1$ entre *nodo2*, k y j . En este caso, se sabe que los elementos encontrados (*nodo3* y *nodo4*) forman un diamante con *superNodo* (*nodo1*), ya que *superNodo* es adyacente a los mismos pues pertenecen a su vecindad. Como la función *buscarDiamanteMinimoDeCompConexa* verifica en cada paso si el diamante obtenido es menor al que se tiene hasta ese momento y se prueban todas las posibles diamantes construídos a partir de *nodo2*, se puede afirmar que obtiene el diamante mínimo para dicha componente conexa. Esto se debe a que *nodo2* es el menor nodo que puede formar un diamante[1] y los restantes nodos (*nodo3* y *nodo4*) se obtienen seleccionando el menor de todos los posibles diamantes.

Luego, dentro de la función *buscarDiamanteMinimoEnVecindad* se verifica si el diamante mínimo de la componente conexa actual es menor que el diamante que se tiene hasta ese paso (almacenado en *diamanteMinimo*). Si esto sucede, el diamante de la componente conexa actual pasa a ser el diamante mínimo. Una vez verificadas todas las componentes conexas se obtiene, en *diamanteMinimo*, el mínimo diamante para la vecindad de *superNodo*. Este diamante se almacena en una lista llamada *diamantesMinimos* que, cuando el algoritmo termina de verificar las vecindades de todos los nodos, contendrá los diamantes mínimos para cada una de las mismas, en caso de que este exista. Finalmente, se busca el mínimo de la lista diamantesMinimos, con lo que se obtiene el diamante mínimo de todo el grafo, resolviendo así el problema.

Complejidad

Para calcular la complejidad de este ejercicio el modelo utilizado fue el uniforme, esto se debe a que lo que define el orden del algoritmo es la cantidad de nodos y de aristas (y cómo están relacionados los nodos por dichas aristas). Teniendo en cuenta que los nodos van de 1 a n, no es relevante observar el tamaño que ocupa cada nodo, sino la cantidad de estos, ya que si el número que representa a un nodo es muy grande, igual de grande es la cantidad de nodos en el grafo y pierde sentido utilizar el modelo logarítmico.

Al crear las listas de adyacencias, el algoritmo lee $2m$ aristas del archivo de entrada y

[1]Las demostraciones de estas propiedades fueron presentadas en la sección "Demostración de la parte a"

las agrega en dichas listas. Para esto crea n listas que representan las adyacencias de cada nodo. Como hay $2m$ aristas que se distribuirán en n listas(dónde la inserción es en $O(1)$), la complejidad temporal de realizar esta tarea será $O(2m + n)$.

Luego se filtran los nodos de grado 0 y 1. Aquí se recorren las listas de adyacencias y se eliminan los nodos que no tengan(en el grafo original) al menos dos nodos adyacentes. Esto requiere recorrer todas las listas de adyacencias que cuesta $2m + n$ (esto se debe a que para que un nodo sea adyacente a otro debe existir una arista y una arista no puede conectar a más de un nodo).

```
crear grafo representado por array de lista de adyacencias (llamado 'adyacencias')
excepto para los nodos de grado 1 y 0
```

A partir de ahora llamamos n a la cantidad de nodos que pertenecían al grafo y no fueron filtrados por su grado. Dado esto, sn està acotado por m, ya que cada nodo tiene asociada por lo menos 2 aristas y como cada arista sòlo puede asociar a dos nodos, la mìnima cantidad de aristas es n. Entonces el costo de n^2 està acotado por $n * m$.

El costo de armar la matriz de adyacencias resulta de crear una matriz de n^2 posiciones con ceros y recorrer las listas de adyacencias y agregar en las 2 posiciones adecuadas en la matriz ($O(1)$) un 1. Entonces cuesta $n^2 + m$. Al ser $n < m$, cuesta $n * m$.

```
armar la matriz de adyacencias de ese grafo
```

Luego se crean las vecindades para los nodos con grado mayor o igual a 3 y se busca el diamante mínimo que exista en cada una de dichas vecindades.

```
para cada nodo (que llamaremos superNodo) del grafo
  si el tamaño de la lista de adyacencia de superNodo es mayor que 3
    crear la vecindad del superNodo (crearVecindad)
    buscar el diamante minimo de esa vecindad (buscarDiamanteMinimoEnVecindad)
    y agregarlo a diamantesMinimos (si es que hay alguno)
```

Creación de la vecindad para un nodo i:

Para cada nodo k adyacente a i se recorren sus adyacentes y se los agrega a una componente de la vecindad si el nodo es también adyacente a i. Esto implica recorrer la lista de adyacencias de i y ,para cada uno de sus adyacentes, recorrer su lista de adyacencia. Como en una lista de adyacencias de un grafo no hay nodos repetidos, recorro como máximo una sola vez la lista de adyacencias de un nodo. A su vez, la cantidad de nodos contenidos entre todas las listas de adyacencias de un grafo es $2m$.

Luego, para crear la 'vecindad' de un nodo, se recorren como máximo $2m$ nodos con los cuales se verifica ,por cada uno, si dos son adyacentes ($O(1)$ en la matriz de adyacencia) y se los agrega a una lista ($O(1)$ por agregarse al principio). Por lo que crear la 'vecindad' de un nodo es de orden m.

```
crearVecindad(superNodo)
  para cada nodo k de adyacencias[superNodo]
    para cada nodo j de adyacencias[k]
      si nodo j es adyacente a superNodo
```

```
      agregar nodo j a listaVecindad
    vecindadDeSuperNodo[k] = listaVecindad
  devolver vecindadDeSuperNodo (grafo representado por array de listas de adyacencia)
```

Búsqueda del diamante mínimo en una determinada vecindad.

Para buscar el diamante mínimo en una vecindad lo que hace el algoritmo es recorrer por DFS cada componente conexa de la vecindad de un nodo, sumando los grados de cada nodo, guardando los nodos y eligiendo el mínimo de ellos. Para esto recorremos la vecindad agregando los nodos que van apareciendo en el camino a una pila si no fueron marcados (en un array, $O(1)$). Es decir que, una vez más, para cada nodo se revisa su lista de adyacencias. Esto es de orden $n + m$.

Si existe el diamante, se lo busca de la siguiente manera:

Para esto se calcula el nodo mínimo nodo, que no se relacione con todos los demás de la componente, es decir, se recorren a lo sumo n nodos y se va guardando el menor, lo que cuesta a lo sumo n. Luego para cada uno de sus adyacentes que pertencen a la vecindad de i, se recorre su lista de adyacencia (ya se mencionó antes que el costo de realizar esto es m) y se verifican adyacencias hasta encontrar los nodos que cumplan la condición de ser adyacentes a i, y no ser adyacentes entre sí. Una vez encontrados estos nodos, se los ordena. Al ser cuatro nodos el tiempo es constante. Es decir que la complejidad de encontrar el diamante minimo en una determinada componente conexa es $O(n + m) = O(m)$

Como hay que crear la vecindad para cada uno de los nodos de grado mayor o igual a 3, el número de vecindades va a depender de la cantidad de nodos. Luego, si existe el diamante se lo busca en orden m. Es decir que el orden de crear una vecindad (n) se sumará al de buscar su diamante si es que existe (n) por cada nodo estudiado. Luego, esto se realiza a lo sumo n veces, una vez por cada nodo.

Es decir que el costo de crear todas las vecindades y buscar su diamante es $n * m$.

```
buscarDiamanteMinimoEnVecindad(vecindadDeSuperNodo)
  recorrer cada componente conexa de vecindadDeSuperNodo por DFS
  sumando los grados de los nodos recorridos (sumaGradosCompConexa) y
  guardando en nodosCompConexa los nodos recorridos y
  en nodoMinimoCompConexa el nodo minimo de la componente conexa

  si sumaGradosCompConexa != a la cantidad de nodos
  del grafo completo de los nodos de la comp conexa
    si diamanteMinimo no existe
      diamanteMinimo = diamante minimo de comp conexa actual
      (buscarDiamanteMinimoDeCompConexa)
    sino
      si nodoMinimoCompConexa <= el minimo nodo de diamanteMinimo
        diamante = diamante minimo de comp conexa actual
        (buscarDiamanteMinimoDeCompConexa)
        si diamante existe y es mas chico que diamanteMinimo
          diamanteMinimo es ese diamente

  devolver diamanteMinimo (puede ser nulo en caso de que no haya diamante)
```

```
buscarDiamanteMinimoDeCompConexa(superNodo, vecindadDeSuperNodo, nodosCompConexa)
  nodo1 = superNodo
  nodoMinimo = nodo minimo de nodosCompConexa con grado menor
  que cantidad de nodos de la componente conexa - 1

  tuplaMin = <nodo mas grande del grafo, nodo mas grande del grafo>
  para cada nodo (adyacenteANodo2) de vecindadDeSuperNodo[nodo2]
    para cada nodo (nodoCandidato) de vecindadDeSuperNodo[adyacenteANodo2]
      si nodoCandidato no es adyacente a nodo2
      y <nodoCandidato, adyacenteANodo2> < tuplaMin
        tuplaMin = <nodoCandidato, adyacenteANodo2>

  nodo3 = primero(tuplaMin)
  nodo4 = segundo(tuplaMin)

  diamanteMinimo = ordenar(nodo1, nodo2, nodo3, nodo4)

  devuelvo diamanteMinimo
```

Cuando ya se tienen los diamantes mínimos de cada vecindad, resta encontrar el mínimo de todos. Es importante notar que no puede haber más componentes conexas que nodos, ya que la componente conexca más chica tiene un nodo(más allá de que debe tener por lo menos cuatro nodos para que haya un diamante.

Como los diamantes tienen 4 nodos, se los compara componente a componente en $O(1)$ y se descarta el mayor de los dos hasta que sólo quede un diamante. Entonces se comparan a los sumo n diamantes en $O(1)$ por lo que en tiempo lineal se consigue el mínimo de los diamantes a partir de los diamantes mínimos de cada vecindad.

```
si hay algun diamante en diamantesMinimos retorno minimo(diamantesMinimos)
```

Luego de haber analizado la complejidad del algoritmo en cada una de las partes (siendo éstas independientes entre si), se puede concluir que el orden del algoritmo en total será equivalente a la de mayor complejidad, es decir $O(n * m)$.

Finalmente estudiaremos la complejidad en función del tamaño de la entrada. Sea t el tamaño de la entrada, n la cantidad de nodos, y m la cantidad de aristas. Tenemos entonces que:

$$t = 2m + 1 \log(n) > m \log(n)$$
$$\implies \text{como } T(n,m) \in O(nm) \text{ y } nm < n^m = 2^{m \log(n)} < 2^t \implies T(t) \in O(2^t)$$

Análisis de resultados

En la presente sección se mostrarán los análisis realizados para grafos con distinta cantidad de nodos. El enfoque principal de las pruebas radica en estudiar los tiempos de ejecución del algoritmo para casos aleatorios y estudiar el comportamiento para los peores casos. Con

este fin se implementó un algoritmo que genera grafos aleatorios de n cantidad de nodos, siendo n el parámetro de dicha función. De esta manera, utilizando dicho algoritmo para distintos valores de n, se contrastaron los tiempos de ejecución con la cota estudiada. Como se mencionó en la sección anterior, la complejidad del algoritmo es $O(n * m)$ pero a fin de estudiar los resultados acotaremos esta complejidad a $O(n^3)$ ya que en cualquier grafo la cantidad de aristas m está acotada por $n * (n-1)/2$ es decir, m es del orden de n^2. Esta acotación aporta simplicidad al estudio de resultados, sin embargo, como el algoritmo tiene una cota menor se espera que los tiempos de ejecución tengan un crecimiento menor que cúbico (en relación a la cantidad de nodos).

La primera prueba realizada consistió en generar grafos aleatorios variando su cantidad de nodos entre 1 y 500. Para cada grafo se estudió su tiempo de ejecución y a continuación se presentan los resultados obtenidos.

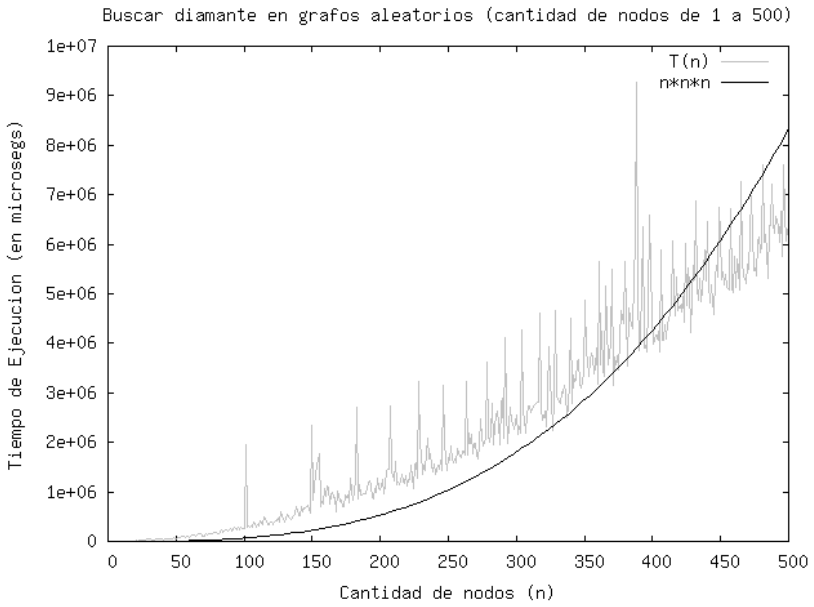


Figura 2.1

Como se puede apreciar en el gráfico, el crecimiento de los tiempos de ejecución del algoritmo es ligeramente inferior a la cota teórica propuesta. Como se mencionó anteriormente, esto se corresponde con lo esperado, ya que para este análisis de utilizó una cota mayor a la del algoritmo ($O(n^3)$). Además se pueden apreciar irregularidades en la curva obtenida, lo que indica que el tiempo que se tarda en encontrar un diamante en un determinado grafo, además de depender de la cantidad de nodos, está determinado por la disposición de las aristas del mismo. Es decir, por la forma en la que está hecho el algoritmo, el tiempo de ejecución va a depender de la cantidad de componentes conexas en las que tenga que buscar el diamante mínimo, para cada vecindad de cada nodo. Los peores casos se darán en grafos en los que se busque un diamante en la mayoría de las componentes conexas de cada vecindad. Estos casos hacen que las optimizaciones del algoritmo, que descartan la búsqueda de un diamante mínimo en ciertas componentes conexas, sean ineficientes.

A continuación presentaremos alguno de estos casos "patológicos" y se verá por qué no se aplican las optimizaciones.

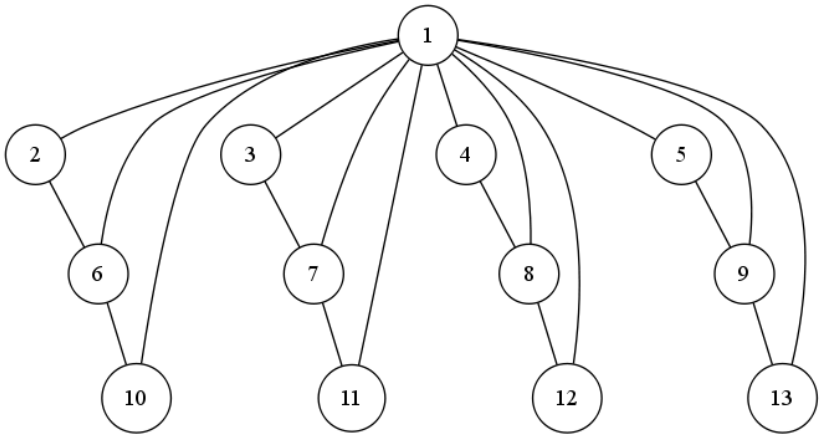


Figura 2.2

Como se puede observar en el grafo presentado, la vecindad del nodo 1 tiene varias componentes conexas que forman un diamante, ya que ninguna es completa. Por esta razón, se deben estudiar todas las componentes conexas para encontrar el diamante mínimo. Sin embargo, podría darse el caso en que el algoritmo implementado no tenga que buscar el diamante mínimo para todas debido a que el mínimo nodo de una componente conexa (llamado nodoMinimoCompConexa en la sección algoritmo) es menor que el mínimo nodo de las otras componentes. Esto sucedería, por ejemplo, si el algoritmo comenzara a estudiar la componente conexa que comienza con el nodo 2. Por el contrario, en el caso en que comenzara por la componente conexa que tiene el nodo 5 y recorriera de derecha a izquierda las componentes conexas del gráfico, se tendría que calcular el mínimo diamante para todas las componentes, por lo que la cantidad de operaciones sería mayor. La forma en que se recorren las componentes conexas depende exclusivamente del orden de las listas de adyacencias pasadas como parámetro, por lo que la aparición de estos peores casos estará ligada a esta distribución.

Cabe aclarar que el esquema presentado en la figura 2.2 representa sólo un caso simplificado sobre los peores casos que se pueden encontrar para el algoritmo mencionado. Es decir, en este gráfico sólo la vecindad del nodo 1 podría representar un peor caso. Fácilmente, este gráfico podría extenderse para formar casos aún peores, ya que cualquiera de las vecindades de los otros nodos podría tener las mismas dificultades que las del nodo 1. Como los peores casos dependen del orden en que se reciben las listas de adyacencias, se hace dificultosa la búsqueda de los mismos. Por esta razón, se optó por explicar en detalle que representaría un peor caso, ya que encontrar y hacer preubas de tiempo de ejecución sobre los mismos resultaba una tarea muy compleja.

Conclusiones

En esta última sección se realizará una breve descripción de los temas y resultados a tener en cuenta sobre este ejercicio. La primer parte del ejercicio requirió un estudio profundo de teoria sobre subgrafos diamantes lo que permitió una mayor facilidad para encarar la segunda parte y una buena cantidad de ideas a tener en cuenta. Sin embargo, el algoritmo presentado debió encargarse de numerosas complicaciones que, a la hora de realizar la demostración de la propiedad referida al tema, no habían sido tenido en cuenta.

En mayor parte esto se debió a la complejidad exigida por la cátedra lo que motivó a la utilización de distintas estructuras y métodos (matrices y listas de adyacencia, estructuras de mapeo, algoritmos de busqueda inteligente, etc) que pudieran cumplir con lo pedido. Otro aspecto dificultoso fue el hecho de tener que retornar el diamante de menor tamaño, lo que implicó tener que guardar un registro de cada uno por componente conexa asociada a una vecindad, y no el primero en ser encontrado. Quedó a las claras aquí como una idea que no es muy complicada ,como es la de encontrar un subgrafo inducido, puede volverse sumamente compleja al pedir ciertos requisitos sobre los resultados (en este caso el diamante mínimo).

En cuanto a las pruebas realizadas, estas arrojaron los resultados esperados en los casos en que se utilizaron grafos generados aleatoriamente. También hemos visto que para cierto tipo de grafos de entrada, puede que el algoritmo varié notoriamente su tiempo de ejecución, aún asi cuando la cantidad de nodos es la misma. Esto se debe principalmente a dos razones. Una de ellas es la manera en que se construyen las listas de adyacencia utilizadas como estructura. Al no recibir los parámetros de entrada en orden, puede que el algoritmo obtenga primero los diamantes con nodos más grandes, lo que no permitiría evitar una gran cantidad de cálculos como si sucedería en caso de obtener inicialmente los mínimos. El otro factor importante a tener en cuenta es el de la distribución de las aristas en el grafo, donde puede ocurrir que casi todas las componentes conexas de cada vecindad tengan la mayor cantidad de aristas pero sin ser completas. Esto implicaría una gran cantidad de diamantes a buscar a comparación de casos promedio. Estos y otros casos patológicos son bastante complejos de generar sobre todo en grafos de gran tamaño por lo que su análisis no pudo ser tan extenso como se deseó.

Ejercicio 3: Red Astor

Introducción

Básicamente, el enunciado del ejercicio propone resolver el problema de minimizar el costo de producción de una red ferroviaria que comunique distintos locales. Se sabe que cada vía tiene un costo de producción determinado y que existen ciertas vías fijas que deben ser incluidas en la solución. Sin demasiado esfuerzo, se observó que este problema se podría modelar como un problema de grafos de la siguiente manera: Cada nodo sería un local y las vias junto a su costo estarían representadas por cada arista. Como se trata de una red ferroviaria de menor costo producción, el problema a resolver es el de encontrar un árbol generador mínimo del grafo completo representado por cada local (nodo) considerando que ciertas aristas ya están prefijadas. Esto se debe a que en principio se podrían comunicar los locales de cualquier forma posible (grafo completo), se debe minimizar el costo (hallar aristas de menor peso) y es necesario que todo local esté comunicado y que exista un único camino entre ellos (es decir, un grafo conexo donde existe un único camino simple entre cada par de nodos, en otras palabras, un árbol).

Luego de modelado el problema, se procedió a idear una solución. De inmediato surgió la idea de utilizar un algoritmo similar a los vistos en clase que encuentran un árbol generador mínimo a partir de un grafo conexo ponderado cualquiera: Algortimo de Kruskal o Algoritmo de Prim. Debido a que el problema original plantea que ciertas aristas sean parte de la solución obligatoriamente, se evaluó si ambos algoritmos permitían encontrar dicho árbol aún partiendo de un conjunto de ejes ya establecidos. Rápidamente se descartó el algortimo de

Prim ya que el mismo tiene como invariante que haya solamente una única componente conexa en cada paso por lo cual en caso de las aristas prefijadas no formen una única componente conexa dicho algoritmo no funcionará. Sin embargo, por la forma en la Kruskal se comporta funcionará aún cuando las aristas pasadas como parámetro formen más de una componente conexa como veremos más adelante.

En la sección siguiente veremos en detalle el algorimto propuesto junto con sus detalles implementativos.

Algoritmo

Antes comenzar con el análisis, se presenta aqui el pseudocódigo que esboza la idea del algoritmo:

```
armarRed()
  aristasPorAgregar = lista de aristas del grafo original
  aristasAstor = lista de aristas prefijadas por Astor
  ordenar aristasPorAgregar por peso

  insertar aristasAstor en el grafo resultado
  costoProduccion = sumo los pesos de todas las aristas de astor

  mientras la cantidad agregada sea menor a n-1 nodos
    tomo una arista de aristasPorAgregar
    si se puede insertar la arista (sePuedeMeter)
      sumar a costoProduccion el peso de la arista
      insertar la arista en el grafo resultado (meterArista)

  ordenar las aristas del grafo resultado segun los valores de los nodos

sePuedeMeter(arista)
  si los dos nodos pertenecen a distintas componentes conexas
  o si alguno de los dos no pertenece a ninguna
    retornar true
  sino
    retornar false

meterArista(arista)
  si ninguno de los dos nodos pertenece a alguna componente conexa
    asignar una nueva componente conexa a ambos nodos

  sino si un nodo pertenece a alguna componente y el otro no
    asignar al otro nodo esa componente

  sino si ambos nodos pertenecen a componentes conexas distintas
    uno las dos componentes en una y le asigno a ambos nodos esa componente

  agrego la arista al grafo resultante
```

En primer lugar, como se puede observar, *armarRed()* es la función principal del algoritmo encargada de generar el arbol generador mínimo del grafo (con aristas prefijadas), es decir, armar la red ferroviaria. Como se mencionó, este algorimto es básicamente el algoritmo de Kruskal para armar el árbol generador mínimo con algunas particularidades. El primer procedimiento consiste en almacenar las aristas prefijadas por Astor y las aristas para agregar en listas enlazadas. Luego se ordenan las aristas a agregar de menor a mayor mediante un algoritmo de sorting eficiente (basado en *Mergesort*) provisto por las librerías del lenguaje, con el fin de poder luego tomar cada arista mínima en cada paso en tiempo constante. Finalmente, el ciclo principal se encarga se armar el árbol resultado de la siguiente manera: Toma una arista de la lista (la mínima hasta ese paso) y pregunta si se puede insertar (*sePuedeMeter()*), es decir si no formará un ciclo que rompa con el invariante de árbol. En caso afirmativo, inserta la arista al grafo (*meterArista()*), acomodando las estructuras como veremos a continuación. Caso contrario, descarta la arista y prosigue con la siguiente, hasta haber insertado $n - 1$ aristas, es decir, haber formado el árbol.

Probablemente el aspecto más interesante a analizar de este algoritmo sean las formas de verficación de ciclo y las estructuras asociadas a estos procedimientos. Razonando, se llegó a la conclusión que verificar la existencia de un ciclo en el árbol al insertar una arista equivale a observar a que componente conexa pertenece cada nodo de la arista en cuestión. En caso de pertenecer ambos nodos a la misma componente conexa es porque la arista formará un ciclo y debe ser descartada. Por este motivo, se concluyó que lo ideal sería encontrar una estructura que permita asociar cada nodo a una componente conexa, permitiendo verificar esto de manera eficiente. Abstrayéndose de esta cuestión, se notó que se debería contar con una estrucutura que represente a varios conjuntos disjuntos, uno por cada componente conexa, que logre buscar a que conjunto pertenece un determinado elemento (nodo) y que consiga unir dos conjuntos en forma óptima. Razonando más profundamente, se llegó a la conclusión de que no se podía obtener una estructura que permita realizar ambas operaciones en tiempo constante: si se podía ver la pertenencia en forma constante, la union tomaría tiempo de orden lineal y viceversa. Como la unión se realiza en ciertos casos (al hallar una arista cuyos nodos pertenezcan ya a distintas componentes conexas) mientras que la busqueda es obligada en cada iteración (para verificar si hay ciclo) se optó por priorizar esta última.

Las estructuras elegidas para respresentar a las componentes conexas (conjuntos disjuntos) fueron dos arreglos de igual tamaño (cantidad de nodos/locales): *índices* y *componentesConexas*. Para el primero de ellos (*índices*), cada posición representa un nodo de ese valor mientras que su contenido es alguna posición del segundo arreglo (*componentesConexas*). Este último contiene un número que representa una componente conexa. Ambos arreglos comienzan con todas sus posiciones en 0 (los nodos no pertenecen a ninguna componente conexa, ya que ninguna arista fue insertada). De esta forma las funciones *sePuedeMeter()* y *meterArista()* se encagarán de observar y modificar estas estructuras, respectivamente.

Con estas estructuras, verificar si hay ciclo (*sePuedeMeter()*) radica simplemente en ir a la posición del arreglo *índices* para ambos nodos y luego acceder al arreglo *componentesConexas* para comprobar si esas posiciones hacen referencia las mismas componentes conexas. De este modo, la operacion de búsqueda se realiza en tiempo constante.

La función *meterArista()* se encargará de acomodar la estructura para mantener la coherencia. De acuerdo a los nodos de la arista en cuestión, tal como se puede ver en el pseudocódigo, existen 3 casos:

- **Los nodos no pertenecen a ninguna componente conexa:** En este caso, para

cada posición de ambos nodos se asigna un nuevo valor de componente conexa (contador *compConexaActual*) al arreglo de *índices*. Luego, en la posición *compConexaActual* del arreglo *componentesConexas* se guarda ese mismo valor. Finalmente, se incrementa el contador *compConexaActual* de componentes conexas.

- **Un nodo pertenece a alguna componente conexa y el otro no:** Simplemente se actualiza la posición del nodo que no pertenecía a ninguna componente conexa en el arreglo *índices* con el valor de la componente conexa del otro nodo (que si se encontrada en alguna componente conexa).

- **Los nodos pertenecen a componentes conexas distintas:** Este último caso es el más costoso, ya que es en el cuál se debe realizar la unión de las componentes conexas a las que pertencen cada uno de los dos nodos. Para realizar esto, se recorre el arreglo *componentesConexas* y por cada elemento que pertenecía a la componente conexa del segundo nodo se le asigna el número de componente conexa del primer nodo.

De este modo, la función *meterArista()* logra mantener la consistencia en todos los casos posibles de forma de que *sePuedeMeter()* retorne siempre los valores esperados.

Concluido el cicilo, se habrán insertado las $n - 1$ aristas necesarias para generar el árbol de forma que no haya ciclos logrando así el objetivo propuesto. Finalmente, a fines de cumplir con lo pedido, se ordenan las aristas según los valores de cada nodo.

Con respecto a la utilización de otras estructuras para mejorar la complejidad, se investigó acerca del asunto, concluyendo efectivamente que no se podrían encontrar formas de resolver el problema de los conjuntos disjuntos de una manera *asintóticamente* mejor. Sin embargo, como veremos en la sección "Conclusiones", existen formas de optimizar las estructuras y los algorimos propuestos.

Correctitud

A continuación se presentará la demostración de correctitud del algoritmo implementado. Dividiremos la demostración en dos partes. Primero mostraremos que el grafo que retorna el algoritmo es un árbol generador del grafo recibido como parámetro. Luego veremos que ademas dicho ábrol generador es mínimo.

Vale aclarar que las aristas agregadas en el primer ciclo del algoritmo (las aristas elegidas por Astor) nunca pueden formar un ciclo por precondición.

Sea G el grafo recibido como parámetro y T el grafo resultado de aplicar *armarRed*:

- T es árbol generador de G

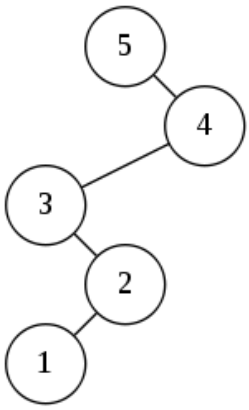
  Para comenzar, mostraremos que T es conexo. Supongamos entonces que no lo es para llegar a un absurdo. Sea e una arista de G que une dos componentes conexas distintas de T. Dicha arista debió ser considerada en algun paso del algoritmo como una posible arista para agregar a T. Como todas las aristas que no formen un ciclo son agregadas en cada paso del algoritmo, es absurdo que e no pertenezca a T, puesto que si en el grafo final e une dos componentes conexas de T, es decir que no forma ciclo en T, tampoco hubiese formado un ciclo en el momento en el que fue considerada para ser insertada ya que nunca se remueven aristas.

Trivialmente se puede apreciar que T es acíclico, puesto que no se agrega ninguna arista que forme un ciclo. Además es también trivial que T contiene todos los nodos de G (se agregan $n-1$ aristas que no forman ciclos).

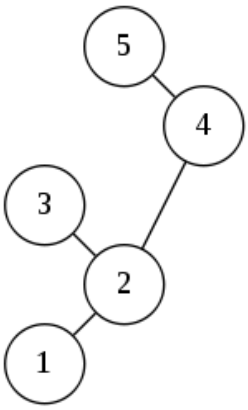
De esta manera podemos concluir que T es un árbol generador mínimo de G, puesto que contiene todos los nodos de G, es conexo y no posee ciclos. Cabe mencionar que si la cantidad de aristas que se colocan sin chequeos por la elección de Astor es igual a $n-1$ entonces no se recorerá el segundo ciclo para agregar aristas porque por precondición ya tendría formado mi árbol generador.

- T es árbol generador que contiene las aristas prefijadas mínimo de G

  Nuevamente encararemos esta demostración por el absurdo. Supongamos entonces que T es árbol generador (conteniendo las aristas de Astor) no mínimo de G y que existe un grafo S distinto de T que si lo es. Sea e la primer arista elegida por el algoritmo que pertenece a T pero no a S. Como e no pertenece a S, entonces si a S le agrego la arista e (notado $S+e$) $S+e$ contendrá un ciclo. Además existe e' perteneciente al ciclo formado en S tal que e' no pertenece a T. Esto sucede porque si e' perteneciera a T entonces tendria un ciclo lo que es absurdo porque es árbol:



Grafo T



Grafo S

En el ejemplo mostrado e sería la arista (3,4) y e' sería (2,4). Sea $S' = S + e - e'$, de esta manera S' es árbol generador de G. Si $l(e') < l(e)$ ($l(e)$ = peso de la arista e) llego a un absurdo. Esto se puede ver porque S contiene todas las aristas anteriores a e más e' y no se forma un ciclo en dicho grafo, por consiguente el algoritmo que va armando el grafo T hubiese elegido a e' en lugar de e y esto es absurdo porque dijimos que e' no pertenece a T. Entonces tenemos que $l(e') \geq l(e)$. Podemos apreciar entonces que el peso de S' es a lo sumo igual al de S, pero no puede ser menor porque S es AGM. Esto implica que S' tiene el mismo peso que S y entonces tambien es árbol generador mínimo de G (conteniendo las aristas prefijadas), más aún, S' tiene una arista más en común con T que S. Repitiendo este paso para todas las aristas distintas entre T y S (como máximo serán $l - f - 1$ pasos, donde l es la cantidad de nodos y f la cantidad de pares fijados por Astor), se llegará a un grafo árbol generador mínimo de G con las mismas aristas de T, lo que es absurdo porque partimos de que T no es mínimo.

Complejidad

Para calcular la complejidad de este ejercicio el modelo utilizado fue el uniforme, esto se debe a que lo que define el orden del algoritmo es la cantidad de nodos y de aristas (y cómo están relacionados los nodos por dichas aristas). Es decir, no es relevante observar el tamaño que ocupa cada nodo, sino la cantidad de nodos que contenga el grafo. En consecuencia, la utilización del modelo logaritmíco carece de sentido para este tipo de problema.

Antes de comenzar, es importante aclarar lo siguiente: Al contar con un grafo completo como parámetro de entrada, el mismo posee $n * (n-1)$ aristas, siendo n la cantidad de nodos. Es decir que $O(n^2) = O(m)$, por lo que, de ahora en adelante, consideraremos a n como único parámetro al calcular la complejidad.

A continuación analizaremos parte por parte la complejidad de cada función del algoritmo:

```
aristasPorAgregar = lista de aristas del grafo original
aristasAstor = lista de aristas prefijadas por Astor
```

El primer paso de la función principal consiste en crear dos listas donde se almacenarán aristas. Para la primera se cuenta con $m = n * (n-1)$ aristas mientras que para la segunda, la cantidad de aristas será menor ya que las aristas fijadas no pueden superar a $n-1$ (porque sino habría ciclos y el problema no tendría solución). Por lo tanto hasta aquí la complejidad resulta de orden $O(n^2)$.

```
ordenar aristasPorAgregar por peso
```

Como se mencionó antes, el algoritmo utilizado para realizar el ordenamiento es una variante del *Mergesort* optimizada, por lo que su complejidad es $O(m * log(m)) = O(n^2 * log(n^2)) = O(n^2 * log(n))$.

```
insertar aristasAstor en el grafo resultado
costoProduccion = sumo los pesos de todas las aristas de astor
```

La primera operación de inserción resulta $O(n)$ por la cota vista, por la que las aristas de Astor no pueden ser más de $n-1$. Con respecto a la segunda operación, es del mismo orden por la misma causa.

```
mientras la cantidad agregada sea menor a n-1 nodos
  tomo una arista de aristasPorAgregar
  si se puede insertar la arista (sePuedeMeter)
    sumar a costoProduccion el peso de la arista
    insertar la arista en el grafo resultado (meterArista)
```

En esta parte, hay un ciclo que se realiza como máximo, m cantidad de veces, debido a que en el peor caso se recorrerán las m aristas del grafo completo. Esto se debe a que en cada iteración una determinada arista es insertada en el grafo resultado, o es descartada como posible arista a insertar.

Con respecto a las operaciones realizadas dentro del ciclo, la primera resulta constante porque consiste en tomar el primer elemento de una lista enlazada. Luego se verifica si la arista obtenida se puede insertar llamando a la funcion *sePuedeMeter()*. Esta función tiene

un costo de peor caso $O(1)$ como veremos más adelante. En caso de poder meter la arista, se realizan dos operaciones: Una suma de costo constante y la insercion de la arista en el grafo. Del mismo modo, veremos luego que la complejidad de esta última función es $O(n)$.

A partir de lo mencionado anteriormente, podemos concluir que la complejidad total de este fragmento del algoritmo resulta $O(n * m) = O(n^3)$ ya que la operación de mayor costo dentro del ciclo es lineal con respecto a n y la cantidad de iteraciones es a lo sumo $O(n^2)$.

```
  ordenar las aristas del grafo resultado segun los valores de los nodos
```

Al igual que antes, *Mergesort* realizará esta operación en $O(n * log(n))$ ya que el grafo resultado posee $n - 1$ aristas por ser árbol generador del grafo original.

Por todo esto, podemos afirmar que la complejidad total del algoritmo es $O(n^3)$ ya que ninguna parte del algorimo supera la complejidad del ciclo analizado.

sePuedeMeter()

```
si los dos nodos pertenecen a distintas componentes conexas
o si alguno de los dos no pertenece a ninguna
  retornar true
sino
  retornar false
```

Como vimos anteriormente, verficar esta condición implica el acceso a dos posiciones de cada arreglo (*índices* y *componentesConexas*) por lo cual el orden resulta $O(1)$.

meterArista()

```
si ninguno de los dos nodos pertenece a alguna componente conexa
  asignar una nueva componente conexa a ambos nodos

sino si un nodo pertenece a alguna componente y el otro no
  asignar al otro nodo esa componente

sino si ambos nodos pertenecen a componentes conexas distintas
  uno las dos componentes en una y le asigno a ambos nodos esa componente

agrego la arista al grafo resultante
```

La primer etapa de la función consiste en evaluar los 3 casos mencionados anteriormente en la descripción del algoritmo. Para los primeros dos, se realizan únicamente operaciones en una cantidad de posiciones acotadas de ambos arreglos por lo cual la complejidad es constante. Sin embargo, el último caso, encargado de unir las dos componentes conexas es $O(n)$ en el peor caso, ya que se deberá recorrer el arreglo de tamaño n para cambiar las posiciones de los nodos relativos a una de las dos componentes conexas. Con respecto a la última operación, es trivial observar que dicho costo es de orden constante ya que se trata de agregar un elemento a una lista enlazada.

Por todo esto, podemos concluir que la función meterArista() es $O(n)$ para el peor caso.

Finalmente estudiaremos la complejidad en función del tamaño de la entrada. Sea t el tamaño de la entrada, n la cantidad de locales, f la cantidad de aristas de Astor, L la matriz de pesos de las aristas y F el arreglo de aristas de Astor. Tenemos entonces que:

$$t = \log(n)+\log(f)+\sum_{i=1}^{n}\sum_{j=1}^{n}\log(L_{i,j})+\sum_{i=1}^{f}\log(F_i) > \log(n)+\log(n)+\sum_{i=1}^{n}\sum_{j=1}^{n}1+\sum_{i=1}^{f}1 > 2\log(n)+n^2+n > n^2$$

$$\implies \text{como } T(n) \in O(n^3) \text{ y } n^3 = (n^2)^{3/2} < t^{3/2} \implies T(t) \in O(t^{3/2})$$

Análisis de resultados

En la siguiente sección analizaremos el comportamiento en la práctica del algoritmo propuesto. El análisis está basado en estudiar el tiempo de ejecución para instancias aleatorias en función de la cantidad de locales manteniendo fija la cantidad de pares de locales fijados por Astor es decir, según el modelo propuesto, la cantidad de aristas a ingresar en el árbol antes de iniciar el algoritmo. A su vez, se verá la relación que existe entre la complejidad teórica calculada y los resultados obtenidos. Además, para completar el análisis se estudiarán los peores casos que presenta el algoritmo.

Para realizar los tiempos de ejecución, se implementó un algoritmo que genera instancias aleatorias, es decir, genera la matriz de los pesos de cada arista para todo el grafo completo y una cierta cantidad de aristas, que representan los locales fijados por Astor, asegurándose que no formen un ciclo.

A continuación se presentan los resultados de las pruebas realizadas para instancias aleatorias, variando la cantidad de locales y manteniendo fija la cantidad de pares fijado por Astor en 4, 20 y 50.

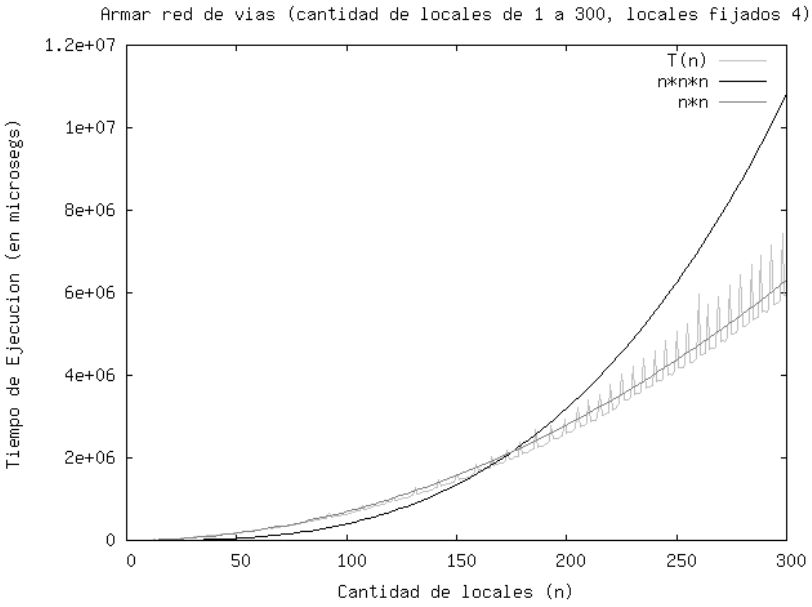


Figura 3.1

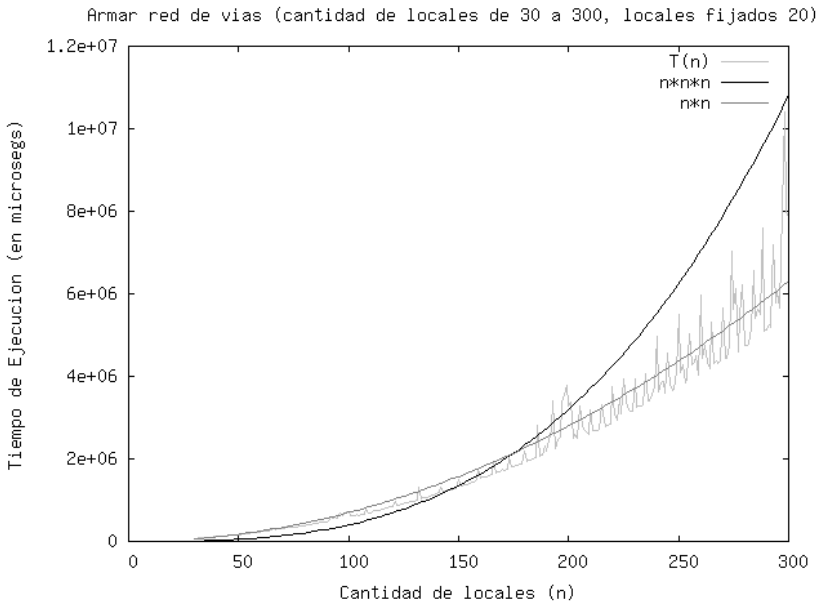


Figura 3.2

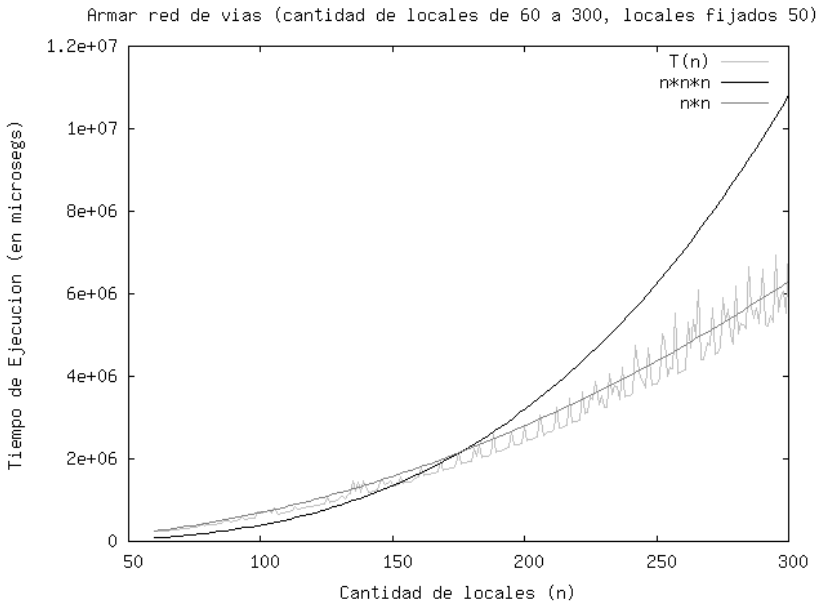


Figura 3.3

Como se puede apreciar, en las figuras 3,1, 3,2 y 3,3 el crecimiento de la función que mide el tiempo de ejercución es similar, a pesar de que, a mayor cantidad de locales fijados por Astor, los tiempos se vuelven más irregulares. La similitud en el crecimiento de los gráficos se debe a que el tiempo que tarda algoritmo depende sobre todo de la disposición de las aristas fijadas por Astor y de la forma en que se van agregando las aristas durante el algoritmo de Kruskal, y no tanto de la cantidad de aristas recibidas. Es decir, si se mantiene fija la cantidad de aristas de Astor y se aumenta significativamente la cantidad de locales totales, la incidencia de este parámetro en el tiempo de ejecución será mínima.

Es importante observar que en las figuras presentadas el crecimiento de la función del tiempo de ejecución es menor que la cota teórica propuesta, $O(n^3)$. El gráfico muestra además que dicho crecimiento se asemeja mucho más al orden cuadrático. Esto se debe a la forma en que el algoritmo arma el árbol resultante: la mayoría de las aristas que agrega en los sucesivos pasos del algoritmo de Kruskal, se agregan en tiempo constante. Agregar una arista es de orden lineal sólo en el caso en que dicha arista une dos componentes conexas del árbol, es decir, cuando se deben recorrer las estructuras utilizadas para cambiar una de las componentes conexas fusionadas. Sin embargo, empíricamente esta última operación se realiza pocas veces en relación a la cantidad total de aristas agregadas. Por esta razón, en el caso promedio la complejidad del algoritmo es menor que la cota teórica propuesta de $O(n^3)$. Por medio de las pruebas, podríamos afirmar entonces que en la práctica el algoritmo presentado se asemeja más a un algoritmo de orden cuadrático que a uno cúbico.

Según lo mencionado anteriormente, se puede deducir que los peores casos serán aquellos en los que agregar una arista al árbol resultado implique unir componentes conexas y, por lo tanto, tener que cambiar la estructura que guarda el número de componente conexa de cada nodo (llamada ComponentesConexas en la sección "Algoritmo"). A continuación presentaremos un ejemplo de peor caso y lo explicaremos en detalle.

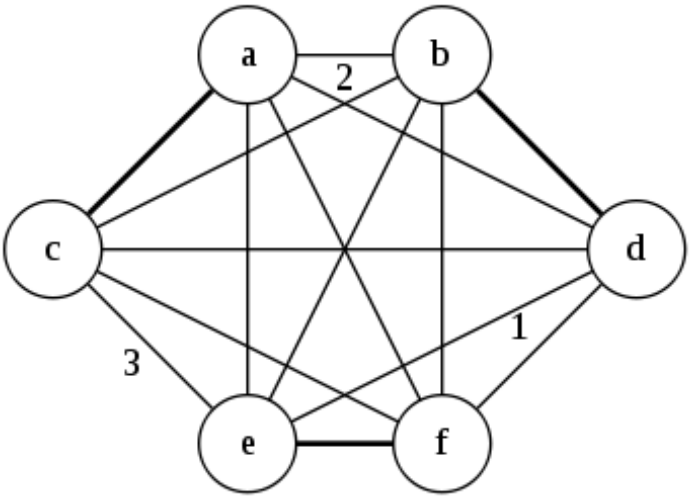


Figura 3.4

En primer lugar, se deben hacer ciertas aclaraciones:

- Las aristas fijadas por Astor son las que figuran resaltadas.
- El peso de las aristas del "interior del grafo" es mayor que las del "borde".

Como se puede notar en el gráfico, la arista de menor peso es la $D - F$ por lo que será la primera en ser agregada en el algoritmo. Agregar esta arista implica unir las componentes conexas de las aristas $E - F$ y $D - B$, por lo que se deberá recorrer todo el arreglo ComponentesConexas para eliminar las apariciones de la componente conexa $D - B$ y así fusionar las tres aristas mencionadas en una única componente conexa que contenga a los nodos E, F, D, B. Esta misma operación se realizará para unir esta última componente conexa con la componente que contiene a los nodos C y A por medio de la arista $A - B$. En este caso, todas

las aristas agregadas unieron componentes conexas, por lo que se debió recorrer el arreglo ComponentesConexas por cada arista haciendo que este sea un peor caso para el algoritmo presentado. Cabe aclarar que estos casos dependen tanto de la disposición tanto de las aristas de Astor, como del peso de las aristas a agregar. Esto hace que generar peores casos para estudiar el tiempo de ejecución sea díficil. Sin embargo, según el ejemplo presentado y la cota teórica propuesta se puede apreciar que el tiempo de los peores casos será del orden de n^3.

Conclusiones

A partir de las secciones detalladas anteriormente se pueden realizar las siguientes conclusiones y aclaraciones. En primer lugar, es importante aclarar que a la hora de resolver el problema se podrían haber utilizado otro tipo de estructuras para mejorar el rendimiento del algoritmo. Como el algoritmo propuesto se basa principalmente en hacer operaciones de buśqueda y unión sobre distintos conjuntos de nodos, se podría haber utilizado una estructura *Union-Find*[1]. En la práctica este tipo de estructuras hacen que el algoritmo mejore significativamente su complejidad debido a que las operaciones de unión y búsqueda son prácticamente lineales (utilizando algoritmos inteligentes sobre listas y árboles). Cabe aclarar que a pesar de esto la cota teórica es la misma que se obtiene con las estructuras utilizadas en nuestro algoritmo.

Como se pudo observar empíricamente, llegamos a la conclusión de que el algoritmo es mejor en promedio de lo que se esperaba en teoría. Es decir, se mostró que tiene un comportamiento del orden cuadrático para la mayoría de los casos, aunque existen casos en lo que esto no es cierto. Cabe mencionar también que se podrían haber hecho otras pruebas para estudiar el comportamiento del algoritmo. En concreto, se podría haber hecho un análisis de los tiempos de ejecución en función de la cantidad de aristas de Astor, manteniendo fija la cantidad de locales. Se supone que a medida que aumente la cantidad de aristas de Astor para una cantidad de locales fija, el tiempo de ejecución debería disminuir ya que debe agregar menor cantidad de aristas teniendo que chequear si se forma un ciclo.

Finalmente, podemos concluir que el problema podría haber sido resuelto de manera un poco más eficiente de haber tenido tiempo adicional. Sin embargo, en la práctica, el tiempo de ejecución del algoritmo propuesto se asemeja más al de los algoritmos que utilizan *Union-Find* que a un algoritmo de orden cúbico.

Referencias

- Artículo de Wikipedia sobre Programación Dinámica
- Artículo de Wikipedia sobre Grafos
- Artículo de Wikipedia sobre Algoritmo de Kruskal
- Artículo de Wikipedia sobre Estructuras Union-Find
- Artículo de Wikipedia sobre Depth/Breadth First Search

[1] Ver referencias sobre este tipo de estructuras en la sección "Referencias"