Aclaraciones Generales

Ejercicio 1: Matching Máximo

Introducción

En este ejercicio se pedía encontrar el matching máximo dentro de un grafo. Se define como matching a un subconjunto de aristas que no comparten vértices. Si bien este ejercicio podría ser de gran complejidad, el mismo se encuentra en una presentación mas accesible dado que no hay que aplicar el algoritmo sobre cualquier tipo de grafos, sino que solamente tiene que ser aplicable a grafos de 3 o más nodos que sean ciclos simples, es decir, grafos que en su isomorfismo planar sean simplemente un dibujo de un polígono.

Algoritmo

El algoritmo propuesto como resolucion del problema consiste en transformar el grafo en un vector, dado que al poder ser representado como un polígono, se puede ver que, quitando una sola arista, el grafo se convierte en una sucesión de aristas con peso, lo cual se puede representar con un vector de números. La idea del algoritmo es poder encontrar el máximo matching posible, una primera aproximación a la solución final, podría ser la siguiente:

1. Se toma una arista cualquiera para comenzar. Luego, el matching buscado tiene dos opciones: contener esa arista, o no contenerla. Entonces el resultado será el máximo entre el matching máximo del grafo sin esa arista(no se utiliza la primer arista tomada) y el matching máximo del grafo sin esa arista, ni ninguno de sus dos aristas vecinas, más el valor de la arista elegida en primer término(si se utiliza la primer arista tomada). De esta manera, el problema sobre un grafo, se convierte en 2 problemas similares pero sobre vectores.

2. En este momento, se necesita sacar el matching máximo, pero sobre vectores, para hacer esto se piensa de manera parecida al punto anterior comenzando con el último elemento: o bien el matching máximo lo contiene, o bien no lo contiene. Luego, el matching buscado para el vector, sera el máximo entre el matching máximo del vector sin el último elemento(caso en el que no se usa el último elemento) y el matching máximo del vector sin los últimos dos elementos más el valor del último elemento(caso en el que si se utiliza el último).

   $matchingMaximo(v_1, v_2, .., v_n) = maximo(matchingMaximo(v_1, v_2, .., v_{n-1}), matchingMa$
   $v_n)$

3. De esta manera, se puede ver que el problema se torna recursivo, siendo solucionado mediante la técnica de dividir y conquistar, teniendo como caso base los vectores de dos o un elemento resolubles trivialmente.

Si bien el algoritmo propuesto retorna el valor esperado, la complejidad del mismo no es óptima. Al hacer los llamados recursivos sucede que hay varios matching máximos que se realizan sobre los mismos vectores, generando así más cálculos de los necesarios. Fue por esto, que el siguiente paso fue modificar el algoritmo para que no calcule las cosas de modo *top down*, sino que fuese un algoritmo *bottom up*, para así evitar los cálculos repetidos, utilizando así la mayor ventaja de la programación dinámica, técnica que define al algoritmo final.

De esta manera, se pensó como teniendo instancias más peque nas del problema, se puede conocer la solución de una instancia mayor. Se utilizó que, dadas las soluciones óptimas para vectores de n-2 y n-1 elementos, la solución para el vector de n elementos es el máximo entre la solucion de n-1 elementos, y la solución de n-2 elementos más el elemento n-ésimo. Es asque el cálculo se torna *bottom up*, calculando los máximos de los subvectores, una sola vez.

A continuación se muestra el pseudocódigo del algoritmo propuesto como resolución del problema.

```
matchingMaximo(grafo G)
   tomar v una arista cualquiera
   tomar u y w vecinos de v
   res := max(matchingSobreVector(G-u-v-w) + v, matchingSobreVector(G-v))

matchingSobreVector(Vector peso_arista)
si tamanio(peso_arista) = 0
   devolver 0
si tamanio(peso_arista) = 1
   devolver peso_arista[1]
si tamanio(peso_arista) = 2
   devolver max(peso_arista[1], peso_arista[2])
si tamanio(peso_arista) >= 3
   peso_maximo_hasta_i-1 := max(peso_arista[1], peso_arista[2])
   peso_maximo_hasta_i-2 := peso_arista[1]
   Para i = 3 hasta n {
      temp := peso_maximo_hasta_i-1
      peso_maximo_hasta_i-1 :=
         max( peso_maximo_hasta_i-2 + peso_arista[i], peso_maximo_hasta_i-1)
```

```
   peso_maximo_hasta_i-2 := temp
}
devolver peso_maximo_hasta_i-1
```

A continuación se presenta un seguimiento del algoritmo sobre un ciclo simple de siete aristas.

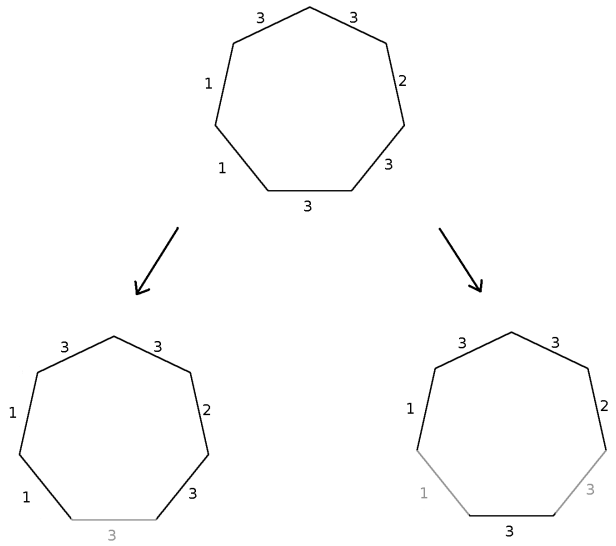
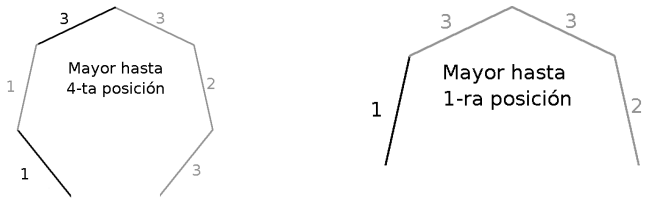


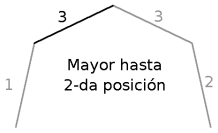
Figure 1: Se toma la arista inferior y se bifurca entre utilizar la misma o no



(a) Derecha 1 (b) Izquierda 1

Figure 2: Dos figuras en la misma linea

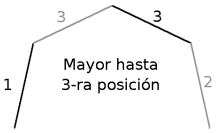
(a) Derecha 1



(b) Izquierda 1

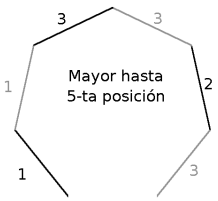
Figure 3: Dos figuras en la misma linea

(a) Derecha 1

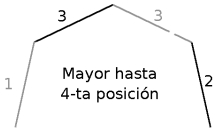


(b) Izquierda 1

Figure 4: camino a



(a) Derecha 1



(b) Izquierda 1

Figure 5: fin de camino a

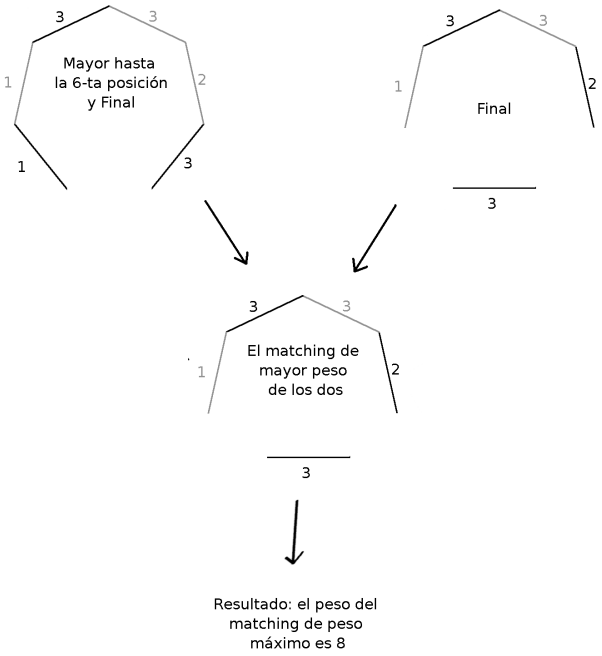


Figure 6: union de los caminos

Demostración de correctitud

Aca va la demostracion, Doc yo no entendi muy bien tu demo porque parece como que definis matching maximo posta de la misma manera que lo define nuestro algoritmo y nose, no digo que este mal, sino que me choca un toque, despues lo hablamos bien.

Complejidad

Se analizará la complejidad de este algoritmo en función del modelo uniforme.

En primer lugar, el algoritmo diseñado toma una arista cualquiera, lo cual se ejecuta en un tiempo constante dado que los vértices se encuentran en un vector de acceso indexado. Luego, se toman los dos vecinos que, dado que el grafo es un circuito simple, también se los puede obtener en un tiempo constante. Luego, el algoritmo realiza una llamada a otro algoritmo que resuelve el matching máximo sobre vectores. Por último, se realiza una comparación

entre los dos resultados arrojados por la rutina auxiliar para poder devolver el máximo, sienda esta una operación también de costo constante. Es por esto que la complejidad de este algoritmo esta regida entonces por la complejidad de la rutina auxiliar a la que hace referencia, ya que todas las demas operaciones toman un tiempo constante de ejecución.

Resulta necesario ver entonces la complejidad del algoritmo de matching máximo sobre vectores. Este algoritmo comienza realizando comparaciones del tamaño del vector para ver si el mismo se resuelve trivialmente. En todos estos casos los costos de las operaciones son constantes. Sin embargo, el peor caso, es que el vector pasado como parámetro contenga tres o más elementos, por lo que la complejidad del algoritmo estará dada por los costos de las operaciones en el caso de que el tamaño sea tres o más.

En este caso, lo primero que se realiza son dos asignaciones y una comparación, lo cual toma tiempo constante. Luego, se realiza un ciclo tomando valores desde tres hasta n. Una vez dentro del ciclo, se realizan varias operaciones de tiempo constante, más especificamente se realizan tres asignaciones, una suma y una comparación. Luego, al tener un ciclo que se ejecuta n-2 veces y sabiendo que dentro del ciclo las operaciones toman un tiempo constante, se puede aseverar que la ejecución de este ciclo será O(n).

Por último, dado que en este último caso las operaciones son dos constantes y el ciclo mencionado, se puede asegurar que la complejidad total del algoritmo para encontrar el matching sobre un vector es O(n).

Finalmente, al ya haber mencionado que la complejidad del algoritmo principal se regía por la complejidad de la subrutina utilizada. Se puede ver que la complejidad de todo el algoritmo será lineal, es decir O(n). Es importante notar que para encontrar un matching máximo, como mínimo hay que observar todas las aristas, por lo que la complejidad lineal resulta ser óptima.

Tamanio de la entrada?!?!?! aca va algo de eso!?!? o ya esta bine asi??

Análisis de resultados

Para analizar este algoritmo, se basó el enfoque en dos aspectos diferentes. Por un lado, se encuentran los análisis sobre la correctitud de la solución propuesta y, por otro lado, se encuentra el análisis sobre el tiempo de ejecución para diferentes archivos de entrada, para poder realizar así, una correlación entre el tiempo de ejecución y la complejidad teórica calculada anteriormente.

Casos de correctitud

Con el fin de realizar una comprobación empírica de la solución propuesta se genero un archivo de entrada con diez casos de prueba.

- Los primeros nueve casos de prueba, estan conformados por los entregados por la cátedra, teniendo de esta manera las soluciones reales para contrastar con las arrojadas por el algoritmo.
- El último caso de prueba esta conformado por un grafo de 30 aristas, con pesos de 1 a 30, ordenados consecutivamente. En este caso, se puede ver que el matching máximo esta dado por tomar las aristas con valor par.

El análisis en este tipo de casos se baso solamente en la correctitud de los mismos y no en el tiempo de ejecución debido a que son grafos de tamaños muy pequeños y el tiempo de ejecucion no sobrepasa los 2 microsegundos en ninguno de los casos.

Para todos los casos propuestos los resultados fueron satisfactorios al ser contrastados con la soluciones previamente obtenidas.

Casos para probar el tiempo de ejecución

Para poder analizar como se comportaba el algoritmo en función del tamaño del grafo a procesar, se implemento un generador de ciclos simples al azar. El mismo se realizó para que arrojase una salida con 500 grafos diferentes, el primero con 10000 aristas, el segundo con 10500 y así sucesivamente. Una vez obtenidos los tiempos de ejecución de cada uno de estos casos, se procedió a graficar los resultados para poder contrastar con la complejidad obtenida teoricamente.

A continuación se presenta un gráfico donde se encuentran simultaneamente los datos obtenidos y un ajuste lineal de los mismos.

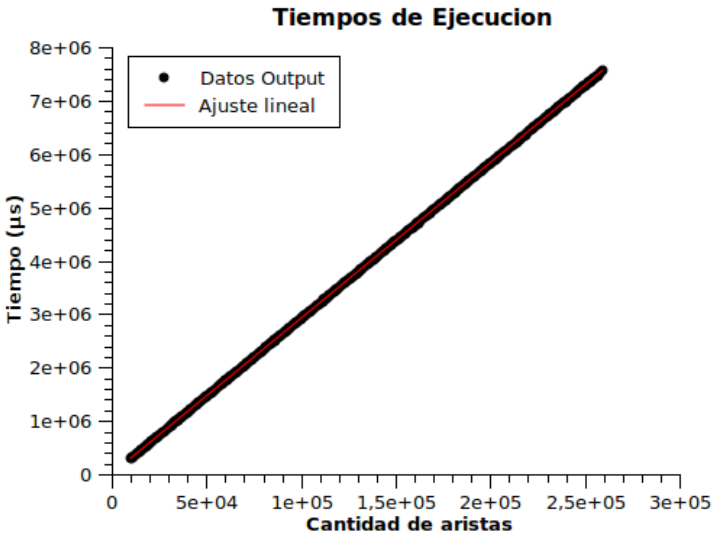


Figure 7: Grafico 1

Como se puede observar en el gráfico, los resultados fueron satisfactorios. El ajuste lineal arrojó un coeficiente de correlación de aproximadamente 0.99999, mostrando empíricamente que la complejidad teórica calculada se condice con los tiempos reales de ejecución.

Conclusiones

Luego de obtener los resultados para el presente ejercicio, se concluyen varios puntos importantes:

- Si bien el problema de matching es de gran complejidad algorítmica, es necesario explotar el hecho de que solo se debe realizar sobre ciclos simples. El hecho de haber explotado esta característica de los grafos presentes en la entrada del algoritmo, fue de gran importancia al momento de lograr un algoritmo eficiente que solo recorriese cada aristas una única vez.

- Resulta de gran importancia pensar diferentes variantes para el algoritmo con el fin de encontrar la versión más eficiente. Durante el diseño del algoritmo se pensó tanto una estrategia de divide and conquer, como una estrategia de programación dinámica. Si bien ambas resolvían el problema satisfactoriamente dada que la idea era basicamente la misma solo que se invertía el orden del recorrido del grafo, cabe destacar como

la simple mejora de recorrer desde el primer elemento hasta el último trajo la gran ventaja de preguntar por cada configuración de aristas una sola vez, generando así un algoritmo de orden lineal, a diferencia del algoritmo exponencial obtenido por divide and conquer.

- La complejidad teórica calculada se pudo ver reflejada en los casos de prueba propuestos. Se cree que la calidad del ajuste lineal que se consiguio se debió a que el algoritmo no presenta mejores o peores casos; es decir, para todo grafo posible el algoritmo debe recorrer exactamente la misma cantidad de veces cada arista. De esta manera, los casos de prueba creados aleatoriamente no pueden ser beneficiados por el azar en ningún caso, generando una casi perfecta correlación lineal en función de la cantidad de aristas.

Ejercicio 2: Se inunda la isla

Introducción

En este ejercicio se pedia encontrar el area que no se inunde en una isla plana luego de ciertas condiciones.

En primer lugar, se tiene una isla que posee la misma altura en todos sus puntos, es por esto que si la marea sube la isla se inunda por completo. La solución para que las partes importantes de la isla no se inunde cuando sube la marea, es poner una serie de vallas rectangulares que no dejen pasar el agua hasta cierta altura. La idea sería entonces, colocando adecuadamente estas vallas, encerrar partes de la isla para que el agua no pueda entrar.

Aca se podria poner un dibujo de ejemplo de la isla con ciertas vallas o algo asi.

El problema consiste en, dado un conjunto de vallas y el nivel de la marea. Calcular cual es el area de la isla que no va a ser inundada.

Algoritmo

A continuación se presenta una explicación al algoritmo propuesto como solución, seguido por su respectivo pseudocódigo.

En primer lugar se observa que, por restricción del problema, las vallas no pueden estar en cualquier lado, sino que su coordenada (x,y) del punto inferior izquierdo esta formada por x e y enteros. Asimismo, como la longitud

de una valla tambien es entera, la coordenada del vértice restante tambien será entera. De esta manera, podemos ver que la isla se puede pensar como una grilla de cuadrados de 1x1. Luego, esta grilla fue pensada como un grafo, donde cada cuadrado de la grilla es un nodo y las aristas estan dadas por la relación entre un cuadrado y sus 4 posibles vecinos. Para armar esta grilla, se calcularon los mínimos y máximos valores de las vallas en x y en y, ya que por fuera de estos los cuadrados que pertenezcan a la isla se inundarán de todos modos (ya que no hay vallas que los cubran).

Al comenzar el algoritmo, todos los cuadrados estan relacionados con sus vecinos con aristas de peso 0. Esto quiere decir que si un cuadrado se inunda, los cuadrados que esten relacionados con ese por una arista de peso 0 tambien se va a inundar.

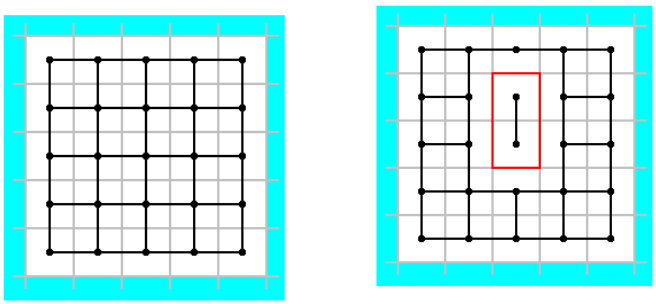
Luego, se recorren todas las vallas dadas por el problema y se setean las nuevas relaciones entre los cuadrados, es decir, si existe una valla de altura 4 entre el nodo v_1 y el nodo v_2, lo que se hace es ponerle un peso de 4 a la arista que los relaciona. Indicando así que solamente si la marea es mayor a 4, el agua pasara de ese v_1 a v_2 directamente.

Por último, lo que se hace es crear una circunvalación de nodos que se inundan alrededor del grafo real y realizar BFS desde uno de estos (que seguro se inunda, por estar por fuera de las vallas) y contar cuantos nodos tiene la componente conexa que el BFS recorre por completo. Cabe destacar, este BFS toma que un nodo es vecino de otro si la arista que los une tiene peso menor a la marea, sino se puede decir que la valla es efectiva entre esos dos nodos y no hay inundación de uno hacia el otro. Una vez obtenida la cantidad total de nodos dados por el recorrido en anchura, resta el último paso que es realizar la sustracción entre los nodos totales de la grilla, y los nodos inundados; obteniendo así, la cantidad total de nodos que no fueron inundados gracias a la protección de las vallas. Por último, como cada nodo representa un cuadrado de área 1, la cantidad de nodos no alcanzados por el BFS es igual al área no inundada.

A continuación se presenta el pseudocódigo del algoritmo recientemente explicado.

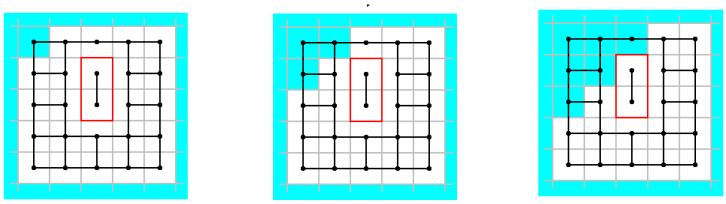
```
areaNoInundada(){
vector vallas;
leer(vallas)
maxmin(vallas)
matriz nodo[alto][ancho];
seteamosMatriz(matriz)
int inundadas  <- bfsContador(matriz)
return (ancho*alto - inundadas)
```

}



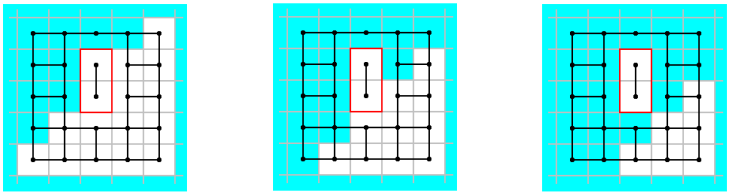
(a) Derecha 1 (b) Izquierda 1

Figure 8: fin de camino a



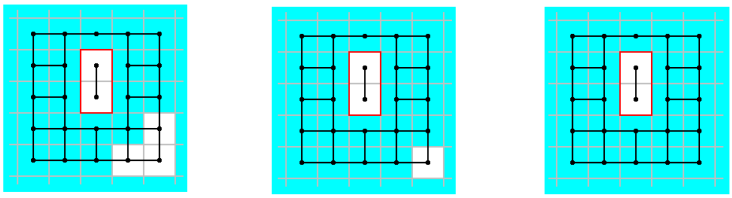
(a) Derecha 1 (b) Izquierda 1 (c) mas izq

Figure 9: fin de camino a



(a) Derecha 1 (b) Izquierda 1 (c) mas izq

Figure 10: fin de camino a



(a) Derecha 1 (b) Izquierda 1 (c) mas izq

Figure 11: fin de camino a

- leer(vallas): Carga en el vector vallas todas las provenientes del input.
- maxmin(vallas): Setea ciertas variables con el tamaño máximo que tendr la isla delimitada por las vallas mas externas.
- seteamosMatriz(matriz): Usando las variables seteadas por maxmin crea un grafo (representado con una matriz) donde establece 4 relaciones por nodo (con sus vecinos) donde el peso del a arista es la altura de la valla que debe atravesar.
- bfsContador(matriz): Recorre el grafo usando bfs, por cada nodo que visita suma uno a una variable que al final devuelve. Recorre todos los nodos que se inundan dado q son los que se relacionan (si la marea supera el peso de las aristas). De este modo al finalizar obtenemos cuantos nodos visitamos coincidiendo con cuantos nodos se inundan.

Complejidad

Para analizar la complejidad se utilizó el modelo uniforme. En este modelo el análisis no esta centrado en el tamaño de los operandos, por lo que el tiempo de ejecución de cada operación se considera constante.

Si tomamos como tamaño de entrada la cantidad de vallas (CV en adelante), y como sabemos que estas no se solapan en más de un punto, podemos acotar por arriba a la cantidad de vallas, por un múltiplo de la cantidad de nodos, si bien acotar las vallas por cuatro veces la cantidad de nodos resulta una cota grosera ya que se estan contando muchas vallas varias veces, la cota resulta util para poder enfocar el análisis del algoritmo simplemente a la cantidad total de nodos; es decir, a la cantidad de casillas en la cuadrícula.

En primer lugar, la función maxmin, busca máximos y mínimos en un vector de vallas linealmente, por lo cual es de orden O(CV) ya que se recorre todo el vector de vallas una vez.

En segundo lugar, se realiza el seteo de toda la matriz mediante la información entregada por las vallas, en esta rutina se itera sobre todas las vallas y se setean las relaciones entre los nodos a partir de las alturas de las vallas pasadas como parámetro. En el peor de los casos, se tienen que setear las cuatro relaciones para cada uno de los nodos de la grilla. Como acceder al nodo se realiza en tiempo constante, y luego hacer los cuatros posibles cambios tambien se realiza en tiempo constante, esta rutina posee una complejidad de O(n) siendo n la cantidad de nodos de la grilla, ya que se realiza un ciclo sobre todos ellos, mientras que en cada iteración se realizan operaciones de complejidad O(1).

En último lugar, lo que se hace es realizar un BFS para lograr identificar la cantidad de nodos de la componente conexa que se inunda. La rutina de BFS tiene una complejidad de O(n+m) siendo n la cantidad de nodos del grafo, y m la cantidad de aristas del mismo. Sin embargo, en este caso podemos acotar m por 4*n, ya que en el peor de los casos, cada nodo posee 4 vecinos. Luego, la complejidad de BFS en este caso es O(n).

En resumen, se realizan tres rutinas, la primera tiene una complejidad O(CV) que ya se mencionó que se puede acotar por O(n); luego, se realizan otras dos rutinas que ambas tiene complejidad O(n). Entonces, la complejidad total del algoritmo, es O(n), siendo n la cantidad de nodos del grafo que representa la isla.

Análisis de resultados

Conclusiones

Ejercicio 3: Bernardo Armando Pandillas

Introducción

En este problema lo que se quiere es, teniendo un conjunto de personas y sabiendo si se conocen entre sí, armar dos grupos de tres personas cada uno, un grupo en el cual las 3 personas se conozcan mutuamente, y otro grupo en el cual ninguno esté relacionado con otro.

Si se pudieran armar varios grupos que cumplan con la restricción planteada, el que debemos encontrar es el menor lexicográficamente. Es decir, si repre-

sentamos cada grupo con una terna ordenada ascendentemente, la terna será menor que cualquier otra terna que represente a un grupo válido.

Consideramos que una terna es menor que otra si la primera componente de la primera terna es menor, o si es igual y la segunda de la misma es menor, o si tanto la primera como la segunda son iguales y la tercera es menor, en cada caso con respecto a la misma componente de la otra terna.

$$(a, b, c) < (d, e, f) \Leftrightarrow (a < d) \vee (a = d \wedge b < e) \vee (a = d \wedge b = e \wedge c < f)$$

Algoritmo

Para resolver el problema se opta por representarlo mediante un grafo.

Cada vértice del grafo representa a una persona diferente y dos vértices están unidos si y sólo si las personas se conocen.

Se puede ver que existe un clique de tres nodos si y sólo si las personas representadas por los nodos que estén en dicho clique se conocen todas entre sí. Ver Apéndice 1.

Para encontrar el grupo de tres personas que no se conozcan entre sí resolvemos el mismo problema pero en el grafo complemento. Ver Apéndice 1.

Sabiendo que el modelo que elegimos es correcto resolvemos el problema de encontrar un clique de tres nodos en el grafo.

Se confecciona la matriz de adyacencias, llamémosla A.

El i-ésimo nodo representa a la i-ésima persona.

Sea $C = A^3$, C tiene en la i-ésima posición de su diagonal la cantidad de caminos de longitud 3 desde el nodo i hasta el nodo i. [1]

Buscamos en C el mínimo i tal que $C_{i,i} \neq 0$, llamémoslo j. Si existe, entonces, el menor clique de 3 nodos contendrá a j. Si no fuera así, entonces existe $k < j$ tal que $C_{k,k} \neq 0$, lo cual es un absurdo, ya que j es el mínimo i tal que $C_{i,i} \neq 0$. Si no existe, termina el algoritmo indicándolo de la manera correspondiente.

De todas las combinaciones posibles para los otros dos nodos, elegimos la menor tal que los tres nodos forman un clique. En nuestra implementación optimizamos esta ultima selección, evitando generar combinaciones que sabemos anticipadamente que no corresponden a un clique. De esta manera obtuvimos el grupo de tres personas que se conocen entre sí.

Por último obtenemos el grafo complemento de G, y realizamos las mismas operaciones para obtener el grupo de tres personas que no se conocen entre sí.

[1] Teórica Algoritmos III

Complejidad

Obtenemos $C = A^3$ utilizando el algoritmo de Strassen para realizar el producto de matrices. La cantidad de operaciones que realiza este algoritmo está en $O\left(m^{log_2(7)}\right)$, para una matriz de m filas y m columnas.[2] Si el grafo tiene n nodos, es necesario generar una matriz de $m \times m$, con $m \leq 2n$, ya que es necesario que m sea una potencia de dos. De esta forma, la cantidad de operaciones para realizar el producto de matrices está en $O\left(7 \times \left(n^{log_2(7)}\right)\right) = O\left(n^{log_2(7)}\right)$.

Se puede encontrar en C el mínimo i tal que $C_{i,i} \neq 0$ en $O(n)$ operaciones, ya que para cada posición de la diagonal se realiza una cantidad constante de operaciones para verificar si en dicha posición hay un cero. Como máximo se recorre toda la diagonal, que tiene n posiciones.

Generando todas las combinaciones de dos nodos la cantidad de operaciones para obtener la menor está en $O(n^2)$, ya que se requiere una cantidad constante de operaciones para generar cada combinación, verificar si es un clique y compararla con la menor hasta el momento.

Por último se realizan las mismas operaciones sobre el grafo complemento. Para lo cual se invierte la matriz de adyacencias en $O(n^2)$.

Como se puede ver, la complejidad computacional del algoritmo en el modelo uniforme está dada por las operaciones correspondientes a la multiplicación de matrices, en $O\left(n^{log_2(7)}\right)$, ya que el resto del algoritmo tiene un orden menor. Como $log_2(7) < 2.9$, el algoritmo realiza una cantidad de operaciones de un orden estrictamente menor a $O(n^3)$.

Análisis de resultados

Conclusiones

Apéndices

1 Demostración de correctitud del modelo

Para resolver el ejercicio 3 se opta por representarlo mediante un grafo.

Cada vértice del grafo representa a una persona diferente y dos vértices están unidos si y sólo si las personas se conocen.

A continuación se demuestra que existe un clique de tres nodos en el grafo que modela el problema si y sólo si las personas representadas por los nodos

[2] Ver Brassard, pág. 273.

que estén en dicho clique se conocen todas entre sí. Luego se demuestra que existe un clique de tres nodos en el complemento del grafo planteado si y sólo si las personas representadas por los nodos que estén en dicho clique no se conocen entre sí.

Sea $G = (V, E)$ el grafo con el que modelamos el problema. Sea $C = (\{v_a, v_b, v_c\}, E_c)$ un clique de tres nodos donde C es subgrafo de G y donde v_a representa a la persona a, v_b a la persona b y v_c a la persona c. Como $(v_a, v_b) \in E$, a conoce a b. También $(v_b, v_c) \in E$, por lo que b conoce a c. Por último como $(v_c, v_a) \in E$, c conoce a a. Por lo tanto las personas representadas por los nodos que están en un clique de tres nodos se conocen entre sí de a pares.

A su vez, si tres personas, llamémoslas a, b y c, se conocen entre sí de a pares, existe un clique de tres nodos que contiene a los nodos que las representan.

Sea $G = (V, E)$ el grafo con el que modelamos el problema. Sean v_a, v_b, v_c los nodos que las representan, como a conoce a b, $(v_a, v_b) \in E$. Como b conoce a c, $(v_b, v_c) \in E$. Como también c conoce a a, $(v_c, v_a) \in E$, por lo tanto $(\{v_a, v_b, v_c\}, \{(v_a, v_b), (v_b, v_c), (v_c, v_a)\})$ es un clique de 3 nodos subgrafo de G.

Resta demostrar que existe un clique de tres nodos en el complemento del grafo planteado si y sólo si las personas representadas por los nodos que estén en dicho clique no se conocen entre sí.

Llamemos $G' = (V', E')$ al complemento de G. Sea $C = (\{v_a, v_b, v_c\}, E')$ un clique de tres nodos donde C es subgrafo de G y donde v_a representa a la persona a, v_b a la persona b y v_c a la persona c. Como $(v_a, v_b) \in E'$, a y b no se conocen. También $(v_b, v_c) \in E'$, por lo que b y c no se conocen. Por último $(v_c, v_a) \in E'$, y por lo tanto c y a no se conocen. Luego a, b, y c forman un grupo de tres personas que no se conocen.

Recíprocamente, si sabemos que a, b y c forman un grupo de tres personas que no se conocen entre sí, y sean v_a, v_b, v_c los nodos que las representan, entonces como a y b no se conocen, $(v_a, v_b) \in E'$. Tampoco b y c se conocen, por lo que $(v_b, v_c) \in E'$. Por último c y a no se conocen, por lo cual $(v_c, v_a) \in E'$. Luego $(\{v_a, v_b, v_c\}, \{(v_a, v_b), (v_b, v_c), (v_c, v_a)\})$ es un clique de tres, subgrafo de G'.