DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Grafos

October 7, 2016 — Problemas, Algoritmos y Programación

Pseudónimo

Emmy Noether

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) 4576-3300
http://www.exactas.uba.ar

Contents

1 Ejercicio 1

1.1 El Problema

Se tiene un grafo $G = (V, E)$ de N vértices y M ejes bidireccionales. Se tienen además dos subconjuntos disjuntos $A \subset V$ y $Esc \subset V$ que representan alumnos y escuelas respectivamente. El problema consiste en encontrar la cantidad mínima de vértices D (que representan divulgadores) tales que para cualquier camino C que una un vértice de A con un vértice de Esc, algún $d_i \in C$. Se pide implementar un algoritmo que resuelva este problema con complejidad temporal $\mathcal{O}(NM^2)$

1.2 Desarrollo

Se pensó el problema utilizando la idea de corte mínimo. Esto es, se quería buscar el corte mínimo tal que los ejes de A queden de un lado y los de Esc del otro. Para esto se armó una red de flujo, agregando dos vértices s y t (sumidero y fuente respectivamente). Se conectó s a los vértices de A y t a los de Esc. Se decidió ponerle capacidad 1 a todos los ejes. De esta forma se pensó en, obteniendo el corte mínimo, ver la mínima cantidad de ejes que separan los vértices de A de los de Esc. Esto sería equivalente a obtener el flujo máximo, por el teorema de $max - flow$ $min - cut$. Tras discutir esta idea se llegó a un ejemplo que mostró que esa solución no era la indicada.

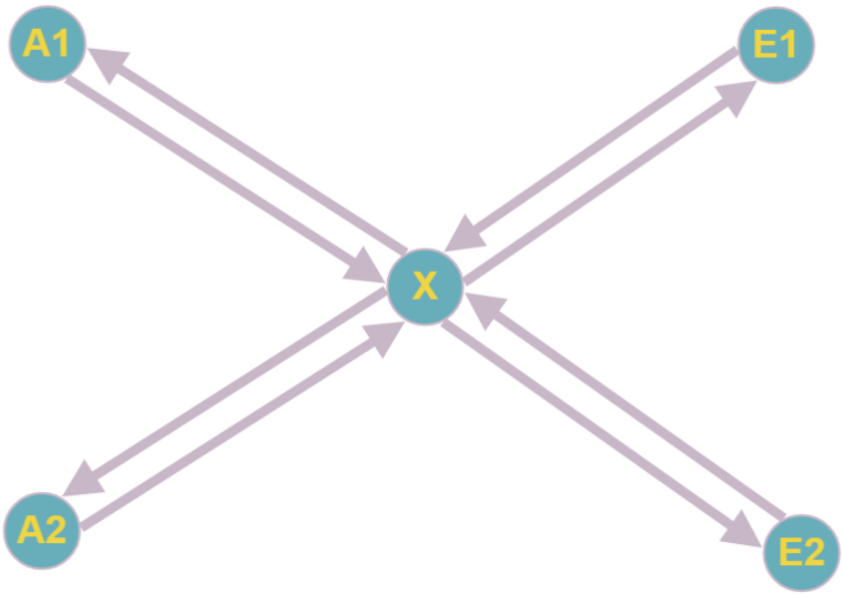


Figure 1: Grafo 1

En el Grafo 1, se puede ver que hay dos alumnos, dos escuelas, y una esquina intermedia. La correcta solución al problema sería poner un divulgador en la esquina central, vértice X en la

imagen. Al armar la red de flujo quedaría el Grafo 2. Asumir que todas las capacidades de los ejes son 1. No están en la imagen para no ensuciarla.

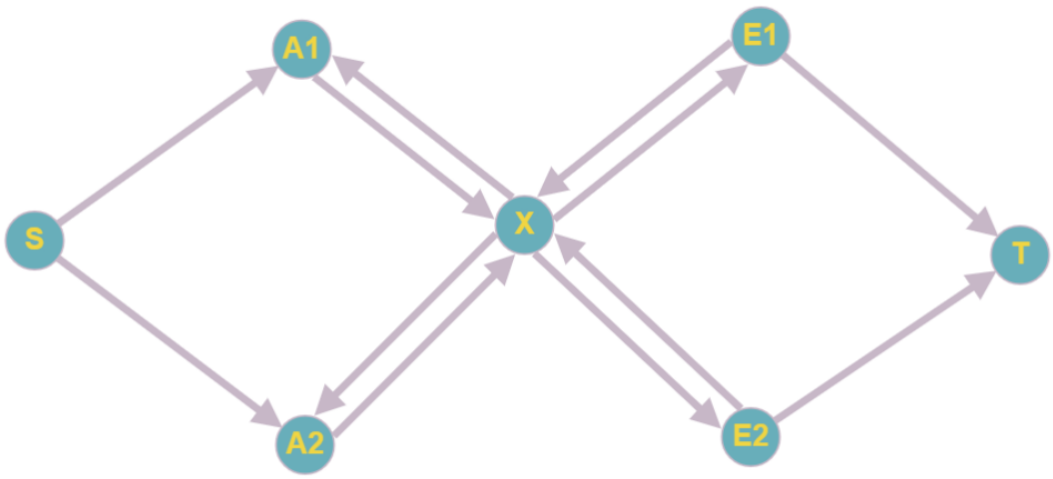


Figure 2: Grafo 2

En el Grafo 2, se puede observar que el flujo máximo es 2, pero con poner un único divulgador en el vértice del medio el problema se solucionaría. Esto se debe a que los divulgadores se ubican en los vértices y no en los ejes. Es decir, los vértices deberían tener capacidades también. Entonces se decidió poner capacidad 1 a los vértices. Así se armó un grafo nuevo, partiendo cada nodo v en v_{in} y v_{out} (conectados entre si por un eje dirigido de *in* a *out*). Se conectó la fuente a los *in* de los alumnos, se conectó el sumidero a los *out* de las escuelas. La idea es que al *in* lleguen todos los ejes entrantes y que del *out* salgan todos los salientes. Quedaría entonces el Grafo 3.

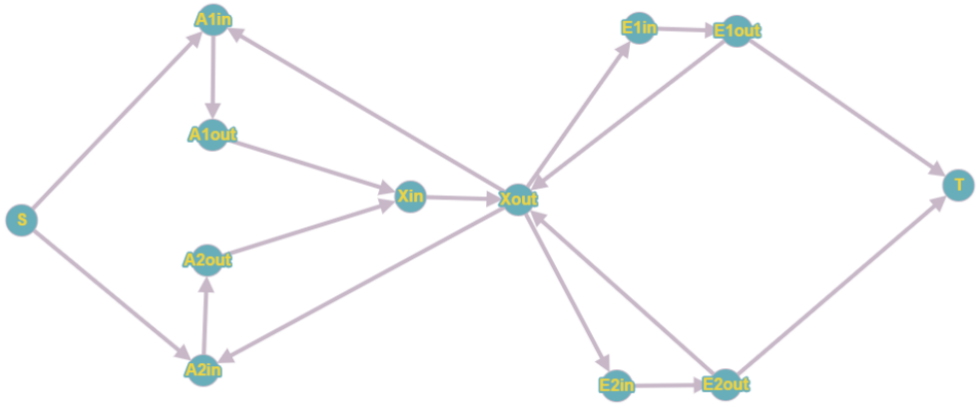


Figure 3: Grafo 3

En el Grafo 3, se puede ver que el flujo máximo va a dar 1. Esto se debe a que el eje de X_{in} a X_{out} es el cuello de botella. El corte mínimo sería el que se puede ver en la siguiente imagen.

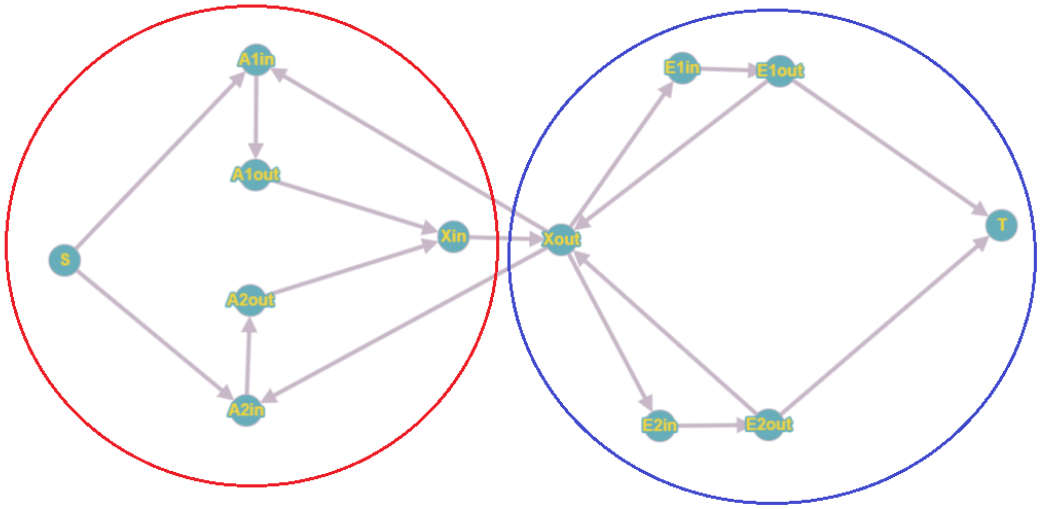


Figure 4: Corte Mínimo del Grafo 3

Al armar el grafo de esta forma, ahora se tienen en cuenta los vértices, por lo que el corte mínimo va a solucionar efectivamente el problema. Este sería, el corte que separa la fuente y el sumidero y minimiza el peso total de los ejes que salen del lado de la fuente hacia el lado del sumidero.

Más formalmente, dado el grafo original $G = (V, E)$, se arma uno nuevo, $G' = (V_{in} \cup V_{out} \cup \{s, t\}, E')$. Para modelar las capacidades de los nodos, se dividió V en dos conjuntos disjuntos V_{in} y V_{out}. Cada vértice de $v_i \in V$ se corresponde con un un $v_{iin} \in V_{in}$ y un $v_{iout} \in V_{out}$.

- Por cada vértice $v \in V$ se agrega a E' el eje (v_{in}, v_{out}).
- Por cada uno de los ejes $(u, v) \in E$, se agregan los ejes (u_{out}, v_{in}) y (v_{out}, u_{in}).
- Por cada vértice $a \in A$ se agrega a E' un eje (s, a_{in}).
- Por cada vértice $e \in Esc$ se agrega a E' un eje (e_{out}, t).

1.2.1 Implementación

Se adjunta un pseudocódigo del algoritmo implementado.

```
ejercicio2(grafo G){
    G' <- armarGrafo(G)
    res <- edmondsKarp(G')
    return res
}
```

Donde la función `armarGrafo` arma el grafo del modelado. Esto es, conectando los alumnos a la fuente y las escuelas al sumidero. Luego por cada eje, cambia el eje (u, v) por (u_{out}, v_{in}) y (v_{out}, u_{in}).

Se implementó luego el algoritmo de Edmonds-Karp para resolver el problema de flujo máximo (que resuelve el problema de corte mínimo).

1.2.2 Complejidad

En esta sección se comentará la complejidad del algoritmo implementado. Primero se analizará la complejidad del armado del grafo. El grafo se va armando a medida se va leyendo la entrada, implementado con listas de adyacencia. Esto es:

- Primero se lee N y se arma un vector de tamaño $2 * N + 2$ (un *in* y un *out* por cada vértice original más fuente y sumidero). Esto es del orden de $\mathcal{O}(N)$.
- Por cada vértice que se lee de la entrada, se conecta su *in* con su *out*. Si se trata de una escuela, se conecta su *out* al sumidero y si se trata de un alumno, se conecta la fuente a su *in*. Esto es del orden de $\mathcal{O}(1)$ y como se hace una vez por vértice, este paso es del orden de $\mathcal{O}(N)$.
- Por cada eje (u, v) que se lee de la entrada, se conecta (u_{out} con v_{in}) y (v_{out} con u_{in}). Esto es $\mathcal{O}(1)$ y como se hace una vez por cada eje, este paso es del orden de $\mathcal{O}(M)$.

Esto deja una complejidad total de $\mathcal{O}(N + M)$ para el armado del grafo. Luego le sigue el algoritmo de Edmonds-Karp. Este tiene una complejidad de $\mathcal{O}(M^2N)$. Cabe mencionar que el grafo que se arma tiene $2 * N + 2$ vértices y $2 * M + A + E + N$ ejes (con A cantidad de alumnos y E cantidad de escuelas).

Como el grafo original es conexo, tiene al menos $N - 1$ ejes, esto es, $O(N) \subseteq O(M)$. Además, $\mathcal{O}(A) \subseteq \mathcal{O}(N)$ y $\mathcal{O}(E) \subseteq \mathcal{O}(N)$ porque $A + E \leq N$. Entonces $\mathcal{O}(2 * M + A + E + N) \subseteq \mathcal{O}(M)$. Además, el grafo nuevo tiene $2 * N + 2$ vértices, es decir, $\mathcal{O}(N)$ vértices.

Por lo tanto, el nuevo grafo tiene $\mathcal{O}(N)$ vértices y $\mathcal{O}(M)$ ejes, por lo que la complejidad de Edmonds-Karp es de $\mathcal{O}(M^2N)$.

Entonces la complejidad total del algoritmo es $\mathcal{O}(M^2N + N + M)$. Como $N > 0$ y $M > 0$, $\mathcal{O}(N + M) \subseteq \mathcal{O}(M^2N)$. Por lo tanto, la complejidad total del algoritmo es de $\mathcal{O}(M^2N)$.

2 Ejercicio 2

2.1 El Problema

Se tienen A acciones y sus valores a lo largo de D días consecutivos. Se quiere saber cual es la mínima cantidad de gráficos en los que se pueden poner dichas acciones de forma tal que no se crucen las líneas ni se toquen en ningún punto y que cada acción esté en un único gráfico. Se pide un algoritmo que resuelva el problema en complejidad del orden de $O(A^2 * (A + D))$.

2.2 Desarrollo

Antes de comenzar, algunas definiciones:

- $P(A_i, D_k)$ define el precio de la acción A_i al día D_k.
- $A_i > A_j \Leftrightarrow (\forall d \in D) P(A_i, d) > P(A_j, d)$, que define un orden parcial para las acciones.

A partir del mencionado orden parcial, pudimos obtener un DAG transitivo.

Por ejemplo, considerar el siguiente caso donde se tienen cuatro acciones y sus respectivos valores a lo largo de 3 días.

Acciones	Días		
	1°	2°	3°
A	5	5	5
B	4	4	4
C	3	4	3
D	1	1	1

Esto daría lugar al siguiente DAG $H = (V, E)$, donde $V = A$ y $(A_i, A_j) \in E \Leftrightarrow A_j > A_i$.

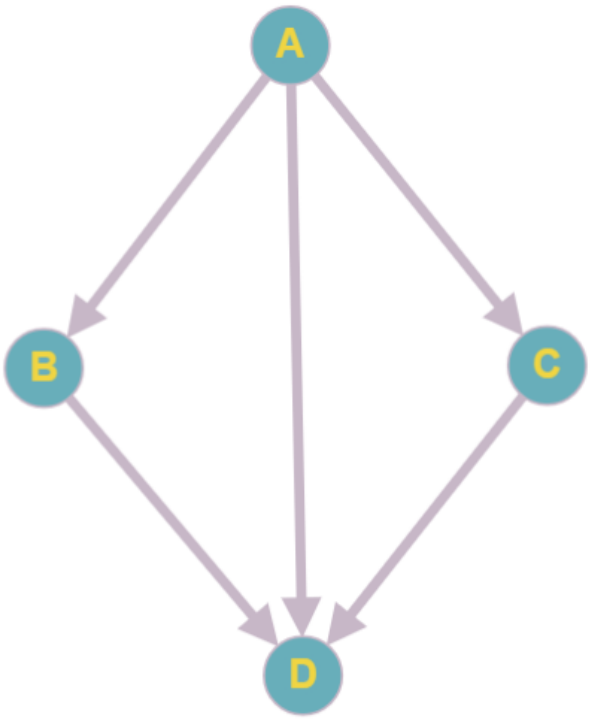


Figure 5: DAG transitivo H

Se puede observar que un camino en este nuevo grafo consiste de acciones que pueden ir por arriba una de otra. Por ejemplo, A puede ir por arriba de B y B puede ir por arriba de D. Esto significa que pueden ir juntas en un mismo gráfico. Entonces, los caminos en el nuevo DAG ilustran grupos de acciones que pueden ir juntas.

También se puede observar que una anticadena en este grafo consiste de acciones que se tocan en algún punto. Es decir, cada vértice de una anticadena debe ir en un gráfico distinto. Se puede ver entonces que va a ser necesario usar tantos gráficos como la anticadena de tamaño máximo.

El teorema de Dilworth dice que el tamaño de la máxima anticadena es igual a la cantidad de caminos en un cubrimiento mínimo por caminos disjuntos en vértices. De esta forma se resuelve el problema. Como se dijo previamente, un camino en el DAG son acciones que pueden ir juntas. Entonces, si se obtiene un cubrimiento por caminos disjuntos en vértices que además sea mínimo, se estaría obteniendo la mínima cantidad de gráficos en los que se pueden dibujar todas esas acciones.

Para resolver este problema se utiliza matching máximo en un grafo bipartito. Para esto se construye un nuevo grafo $G = (IZQ, DER, E)$, donde $IZQ = DER = A$, y $(\forall A_i, A_j \in A)(A_i, A_j) \in E_H \Rightarrow (A_i, A_j) \in E_G$, con $A_i \in IZQ$ y $A_j \in DER$.
Así surge el siguiente grafo bipartito.

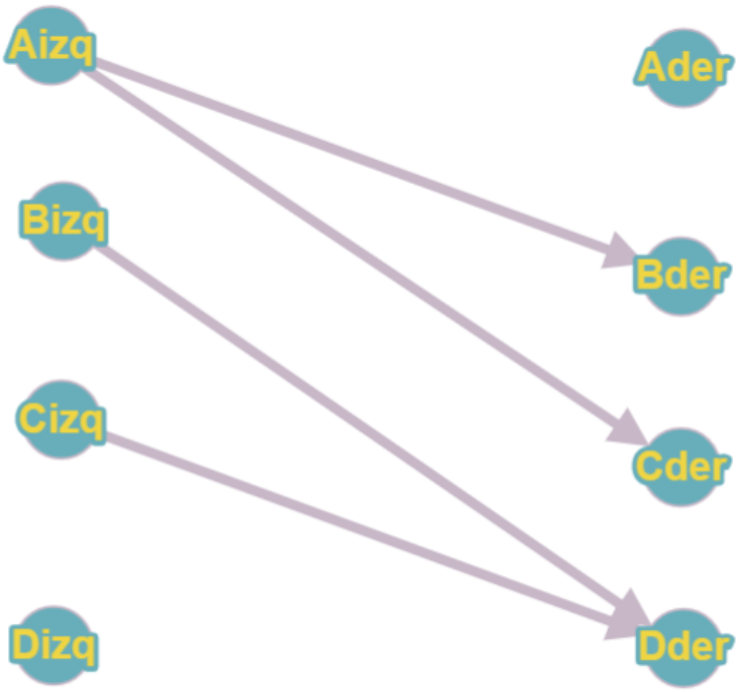


Figure 6: Grafo Bipartito G

Si se busca un matching máximo M en el siguiente grafo, se verá que es de cardinalidad 2. Uno de estos matchings posibles es $M = \{(A_{izq}, B_{der}), (C_{izq}, D_{der})\}$. Se adjunta una imagen del matching en cuestión.

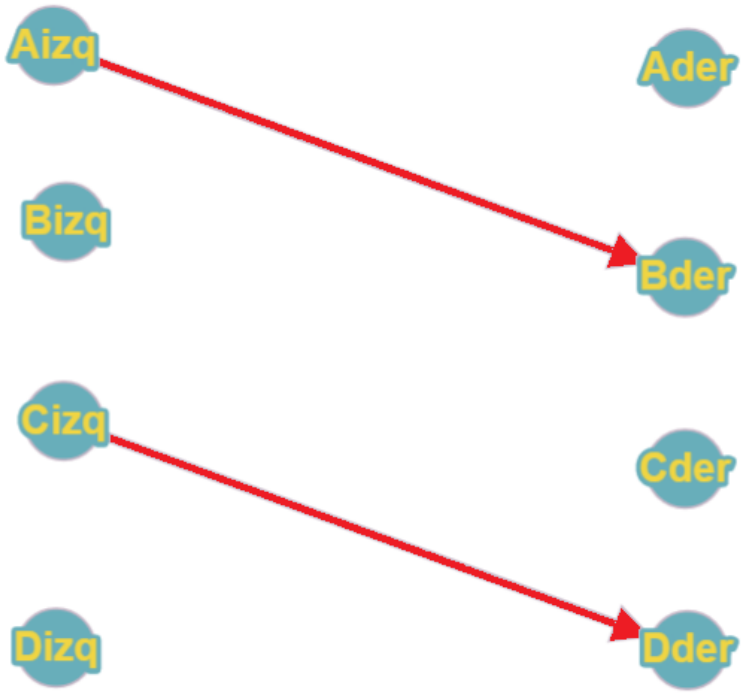


Figure 7: Matching M de G

Se puede observar que cada vértice de IZQ que no tiene ningun eje en el matching, como B_{izq} y D_{izq}, se corresponden con el último vértice de un camino en el DAG H. Para el caso de B, es el camino $A - B$ y para el caso de D, el camino es $C - D$. Si se hubiera elegido otro matching, como $A_{izq} - B_{der}$ y $B_{izq} - D_{der}$, los caminos representados serían $A - B - D$ y C.

Entonces, la cantidad de vértices de IZQ del grafo G que no tienen ningun eje en el matching (esto es, la cantidad de vértices de la partición menos el tamaño del matching máximo encontrado) se corresponde con la cantidad de caminos de un cubrimiento por caminos disjuntos en vértices del DAG H. Entonces, al maximizar el matching, se minimiza la cantidad de caminos. Por lo tanto, resolver matching máximo en el grafo G resuelve el problema de encontrar mínimo cubrimiento por caminos disjuntos en vértices en el DAG H.

El matching bipartito, a su vez, se resuelve utilizando flujo como fue visto en clase. Se conecta una fuente a todos los vértices de la partición IZQ y se conecta todos los vértices de la partición DER a un sumidero, poniendole a todos los ejes capacidad 1.

2.2.1 Complejidad

Nuestro algoritmo consta de las siguientes partes partes:

- Lectura de la entrada: Se leen A acciones y por cada acción, el valor correspondiente al *i-ésimo* día. Complejidad: $\mathcal{O}(A * D)$.
- Armado del grafo: Al momento de armar el grafo, se procedió a armar directamente el grafo bipartito G previamente mencionado. Para eso, se generaron $A * 2 + 2$ nodos. $A * 2$

correspondientes a los conjuntos IZQ y DER de vértices, más la fuente y el sumidero. Por cada acción, se procede a ver *a cuales otras acciones es mayor*, por lo que por cada accion se deben revisar todas las demás acciones y sus días. Complejidad: $\mathcal{O}(A^2 * D)$.

- Calculo de flujo: El cálculo del flujo utilizando el algoritmo de Edmonds Karp (ver 5.1), cuya complejidad es $\mathcal{O}(V * E^2)$. Sin embargo, al tratarse de un grafo bipartito, la complejidad es $\mathcal{O}(V * E)$ [1]. Dado que en nuestro grafo puede haber, en peor caso, $\frac{(A-1)*(A-2)}{2} + 2A$ ejes (es decir, $\mathcal{O}(A^2)$), la complejidad es $\mathcal{O}(A^3)$.

Por propiedad de la *cota asintótica superior*, la complejidad total es $\mathcal{O}(A^2 * (A + D))$

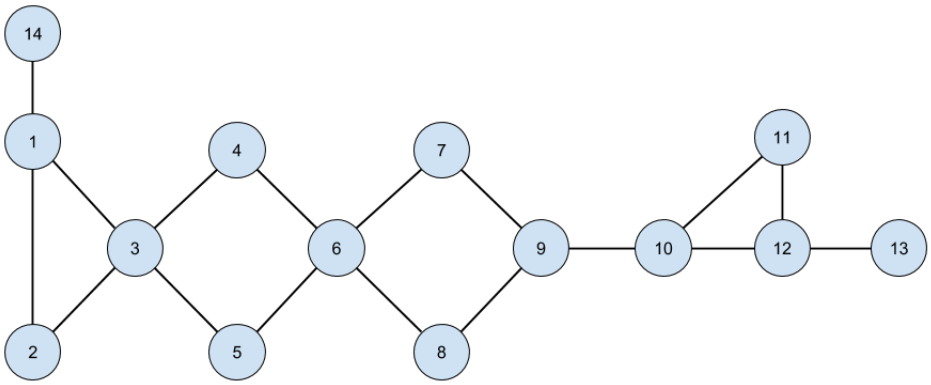
3 Ejercicio 3

3.1 El Problema

Tenemos una ciudad con **N** esquinas, unidas por **M** calles bidireccionales, donde para todo par de esquinas, existe un camino entre ellas. Sobre esta ciudad, vamos a querer responder **Q** consultas, de las cuales existen 3 tipos distintos:

Tipo A: dadas dos esquinas e1 y e2, imprimir la cantidad de calles que, en caso de ser cortadas (solo cortando esa calle), impiden llegar de e1 a e2.
Tipo B: dada una calle, imprimir un 1 si al cortar la calle existen al menos dos esquinas entre las que deja de haber un camino, y 0 en caso contrario.
Tipo C: dada una esquina e1, imprimir la cantidad de esquinas e2 tales que de cortar una sola calle, sea cual sea, seguirá habiendo camino de e1 a e2.

Dadas las características de la ciudad, podemos representarla mediante un grafo conexo.



Y bajo esta nueva representación, podemos traducir las consultas de la siguiente forma:

Tipo A: dados dos nodos n1 y n2, imprimir la cantidad de puentes en el camino para llegar de n1 a n2.
Tipo B: dado un eje, imprimir un 1 si es un puente, y 0 en caso contrario.
Tipo C: dado un nodo n, imprimir la cantidad de nodos en la componente 2-eje-conexa, menos 1.

Aclaraciones:
Tipo A: si bien puede existir más de un camino para llegar de un nodo a otro, todos esos camino pasarán por los mismo ejes puentes, si es que existen. Si existieran dos caminos distintos para llegar de un nodo a otro, y cada camino pasase por ejes puentes distintos, entonces no serían puentes.
Componente 2-eje-conexa: es una componente que no tiene un puente.

3.2 Solución

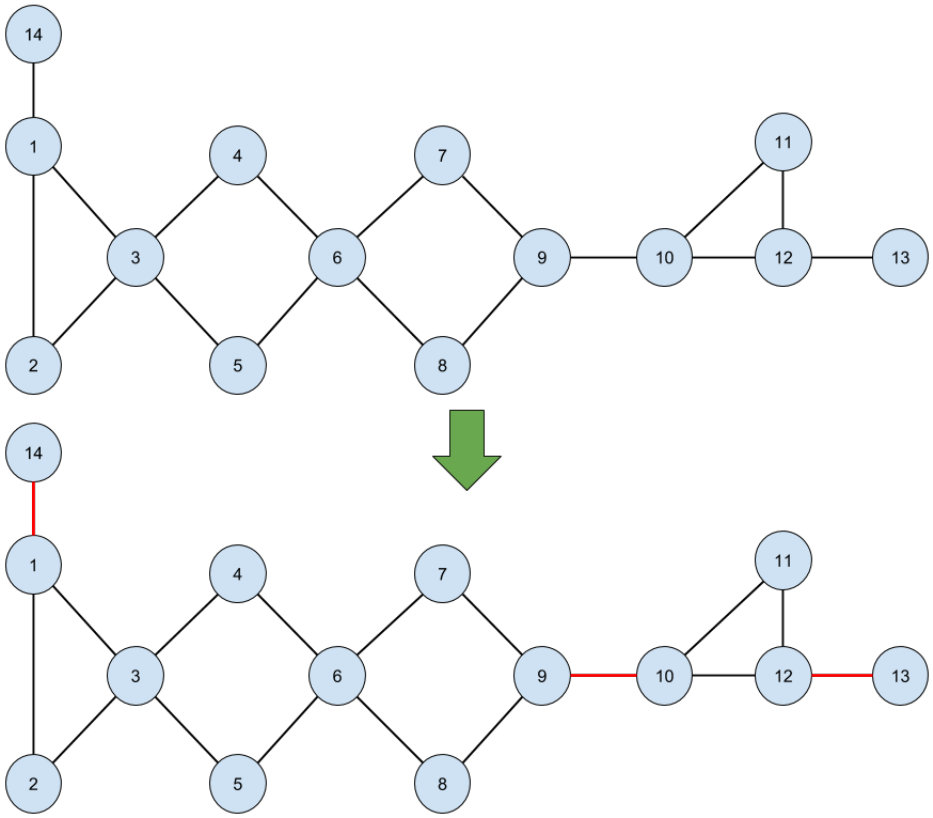
Dividimos la solución en 3 grandes partes:

1. Cálculo de puentes

2. Cálculo de componentes 2-eje-conexas
3. Cálculo de puentes en camino

3.2.1 Cálculo de puentes

Todas las consultas tienen relación algún tipo de relación con ejes puente. Dado que las consultas de tipo B y C deben responderse en tiempo constante, esta información tiene que estar precalculada. Para calcular los ejes puente, utilizamos el algoritmo de *dfs* visto en clase. La única modificación que le hicimos, fue que no guardamos los nodos de articulación. Luego de correr este algoritmo, ya tendremos los ejes puentes distinguidos:



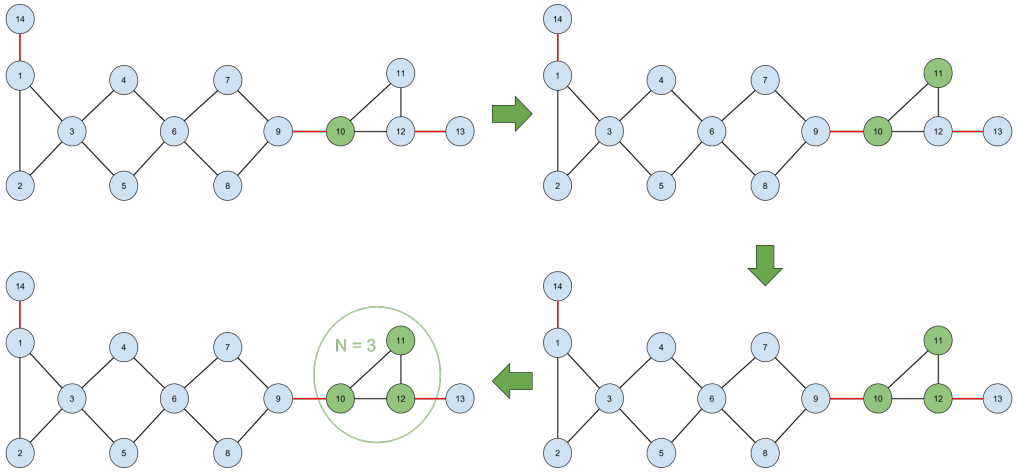
3.2.2 Cálculo de componentes 2-eje-conexas

Una vez calculados los puentes, vamos a calcular las componentes 2-eje-conexas. Lo que hacemos es, por cada nodo:

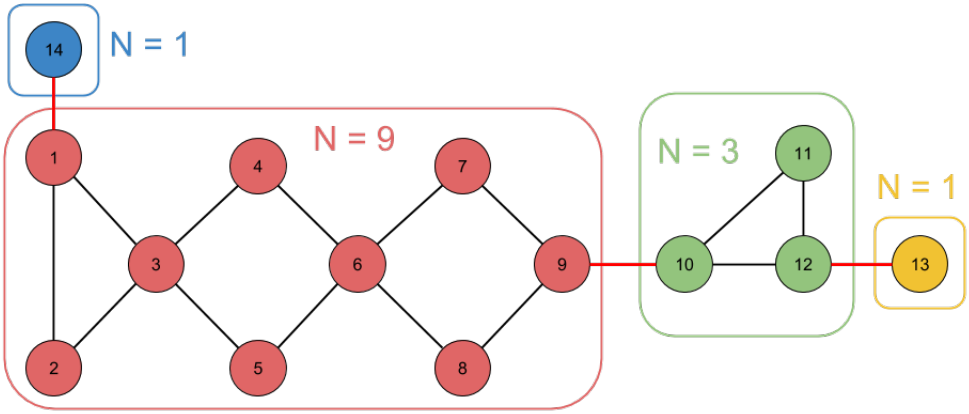
1. Si está visitado, saltearse los siguientes pasos.
2. Guardar el nodo en una lista.
3. Marcarlo como visitado.
4. Por cada vecino que no esté conectado por un puente, ir al paso 1.

Cada vez que ya no pueda visitar más vecinos, en la lista tengo entonces todos los nodos de la componente deseada.

Así sería el cálculo de una componente:



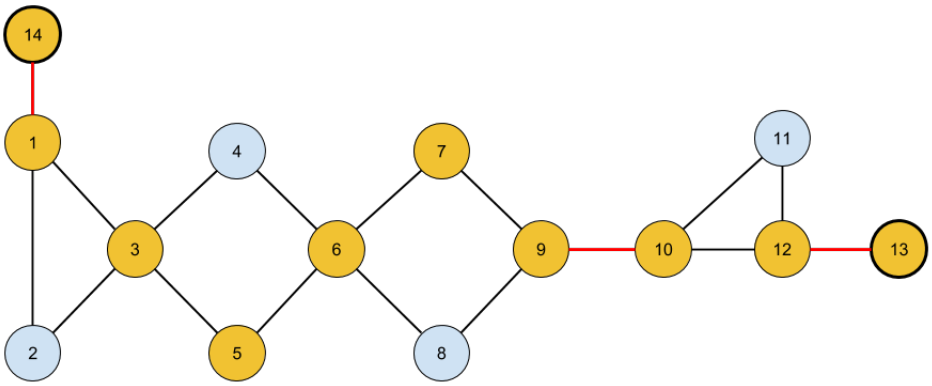
Y así quedaría el grafo luego de calcular todas las componentes:



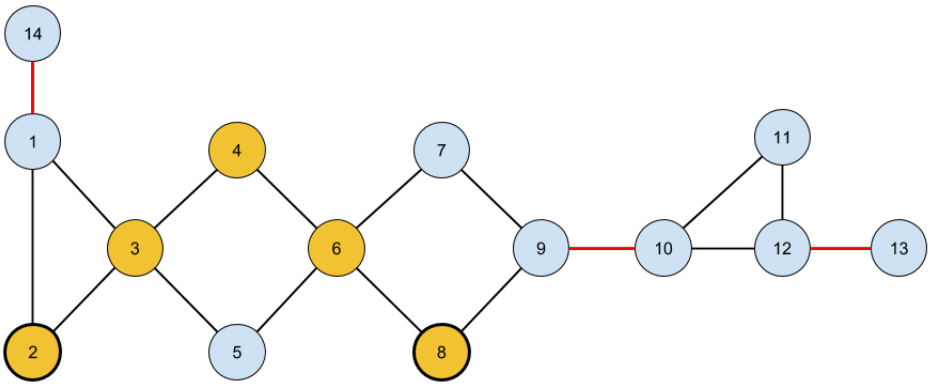
Una vez calculadas todas las distintas componentes, guardamos para cada nodo la cantidad de nodos en la misma componente (menos 1). Esta información nos permitirá responder las consultas de tipo C.

3.2.3 Cálculo de puentes en camino

Para las consultas de tipo A, nos interesa saber la cantidad de puentes en el camino entre dos nodos. Obviamente hablamos de un camino sin nodos repetidos.
En el ejemplo actual, si tenemos la consulta *A 13 14*, un posible camino es el siguiente:



Podemos ver que en camino existen 3 ejes puente, por lo que esa sería la respuesta a la consulta. Ahora, si recibimos esta otra consulta *A 2 8*, un camino posible es el siguiente:



En este caso no cruzamos ningún puente, por lo que la respuesta sería 0. Lo que estamos haciendo entonces es un simple *dfs*, desde uno de los nodos, en busca de otro. Vamos a ir contando los puentes que pasamos en el camino, y sin importar el camino que se tome, la cantidad de puentes va a ser siemple la misma.

Una vez resueltos estos tres subproblemas, podremos resolver consultas de tipos A, B y C.

3.2.4 Complejidad

Nuestro algoritmo tiene las siguientes partes:

1. Inicialización
2. Cálculo de puentes
3. Cálculo de componentes 2-eje-conexas
4. Resolución de consultas
   (a) Tipo A
   (b) Tipo B
   (c) Tipo C

3.2.5 Inicialización

Utilizamos lista de adyacencia para representar el grafo, por lo que la inicialización del mismo tiene complejidad $\mathcal{O}(N + M)$. Ahora, como dice la Pista 1, entre todo par de nodos existe un camino, por lo que $N \in \mathcal{O}(M)$. Entonces, la inicialización termina teniendo costo $\mathcal{O}(M)$. También inicializamos otras estructuras como vectores de tamaño N y M, pero su costo sigue entrando dentro de $\mathcal{O}(M)$.

3.2.6 Cálculo de puentes

El algoritmo de cálculo de puentes es el visto en clase. La única modificación hecha fue que no guardamos los nodos de articulación. La implementación es un *dfs* normal, donde cada nodo es visitado una única vez. Su costo es entonces $\mathcal{O}(N + M)$, pero al igual que en la sección anterior, sabemos que equivale a $\mathcal{O}(M)$.

3.2.7 Cálculo de componentes 2-eje-conexas

Cómo vimos previamente, el algoritmo consiste en correr un *dfs* desde cada nodo, para encontrar todos los nodos dentro de la misma componente. Sin embargo, nos aseguramos de compartir el vector de nodos visitados entre todos los *dfs*. Entonces, en total, cada nodo es visitado una única vez, y cada eje como mucho será recorrido 2 veces. Nos encontramos entonces nuevamente con una complejidad $\mathcal{O}(N + M)$, que otra vez reemplazamos por $\mathcal{O}(M)$.

3.2.8 Resolución de consultas

Tipo A
Ya leer las consultas de tipo A nos toma $\mathcal{O}(Q_A)$. Ahora, por cada consulta, queremos poder calcular la cantidad de puentes en el camino. Como dijimos antes, vamos a hacer un *dfs* desde uno de los nodos, en busca del otro. Esto nuevamente (y por última vez) tiene una complejidad una complejidad $\mathcal{O}(N + M)$. Y otra vez, reemplazamos por $\mathcal{O}(M)$. Entonces, en total, el costo termina siendo $\mathcal{O}(MQ_A)$

Tipo B
Las consultas de tipo B se responden en tiempo constante, ya que previamente calculamos los ejes puente. Entonces, terminamos teniendo complejidad $\mathcal{O}(Q_B)$.

Tipo C
Las consultas de tipo C también se pueden responder en tiempo constante, ya que previamente calculamos para cada nodo la cantidad de nodos en la misma componente 2-eje-conexa. Entonces, responder este tipo de consulta tiene complejidad $\mathcal{O}(Q_C)$.

3.2.9 Conclusión

Resumiendo, estas fueron las complejidades calculadas:
Inicialización: $\mathcal{O}(M)$
Cálculo de puentes: $\mathcal{O}(M)$
Cálculo de componentes 2-eje-conexas: $\mathcal{O}(M)$
Resolución de consultas:
Tipo A: $\mathcal{O}(MQ_A)$
Tipo B: $\mathcal{O}(Q_B)$

Tipo C: $\mathcal{O}(Q_C)$

Sumando cada una de estas complejidades, terminamos con: $\mathcal{O}(M + MQ_A + Q_B + Q_C)$

4 Ejercicio 4

4.1 El Problema

Se tienen A aulas y P pasillos, que se pueden recorrer unidireccionalmente. Se desea saber, a través de una serie de Q consultas, si es posible llegar desde un aula A_i a un aula A_j y volver al aula A_i.

4.2 Desarrollo

Para resolver el problema en cuestion, se modeló un grafo dirigido $G = (V, E)$ con $V = A$ y $E = P$. En dicho grafo, se calcularon las componentes fuertemente conexas. Por definición de componente fuertemente conexa, si dos aulas A_i y A_j pertenecen a la misma componente, eso significa que es posible llegar desde A_i a A_j y volver a A_i [2]. Por lo tanto, las consultas se responden simplemente verificando si las aulas por las que se pregunta, pertenecen a la misma componente.

4.2.1 Complejidad

Nuestro algoritmo consta de las siguientes partes

- Lectura de la entrada y armado del grafo: Se crean los vértices A y se van agregando los P pasillos. Complejidad: $\mathcal{O}(A + P)$.
- Calculo de componentes fuertemente conexas: Para el calculo de las componentes, se utilizó el algoritmo de Kosaraju, que calcula las componentes fuertemente conexas, y cuya complejidad es $\mathcal{O}(V + E)$ [1]. La implementación de dicho algoritmo se corresponde con la indicada en el libro *Introductions to Algorithms* [1], con la única diferencia de que cuando se armó el grafo G, se armó tambien el grafo G^T (es decir, el calculo de G^T no se lleva a cabo en la ejecución de Kosaraju). Complejidad: $\mathcal{O}(A + P)$.
- Consultas: Las consultas se realizan de forma inmediata, simplemente preguntando si dos aulas pertenecen a la misma componente conexa. Complejidad: $\mathcal{O}(Q)$.

Por propiedad de la *cota asintótica superior*, la complejidad total es $\mathcal{O}(A + P + Q)$.

5 Apéndice

5.1 Edmonds Karp

Inicializar flujo en 0
while HayCaminoDeAumento **do**
    Incrementar flujo
    ActualizarRedResidual
end while
Retornar flujo

function HayCaminoDeAumento
    Hacer DFS pasando solo por nodos no visitados y que aun no hayan llenado el flujo
end function

function ActualizarRedResidual
    for eje en camino de aumento **do**
        Saturar eje
        Agregar eje en direccion opuesta
    end for
end function

NOTA 1: Se consideró que todos los ejes tienen capacidad 1. Por lo tanto, de haber camino de aumento, su flujo será siempre 1.

NOTA 2: "Agregar eje en dirección opuesta" a lo sumo duplica la cantidad de ejes en la red residual [1]*. Esto implica que encontrar un camino de aumento no modifica la cota de complejidad y sigue siendo* $\mathcal{O}(E)$*. Como el flujo se puede aumentar a lo sumo* $\mathcal{O}(VE)$ *[1] veces, la complejidad del algoritmo es* $\mathcal{O}(VE^2)$

References

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*, pages 553,556, 669. McGraw-Hill Higher Education, 2nd edition, 2001.

[2] Strongly Connected Component. `https://en.wikipedia.org/wiki/Strongly_connected_component`.

[1] Por la nota 1, los ejes de un camino de aumento no pueden ser elegidos en otro.