# Trabajo Práctico 2

July 3, 2019

## 1 Alumnos de secundario - Peso 10

Haiku milenario:

*La respuesta está*
*en el máximo flujo.*
*Yo te lo digo.*

Para la resolución de este ejercicio se nos ocurrió que podíamos representarlo como el problema de Min-Cut.

### 1.1 Construcción del grafo

Con la información dada vamos a construir un grafo de la siguiente forma:

1. Cada esquina $v$ va a pasar a ser dos nodos del grafo: un $v_{in}$ y un $v_{out}$ conectados por una arista de capacidad 1
2. Cada calle bidireccional que conectaba dos esquinas (u,v) será una arista de capacidad 1 que conecte $u_{out}$ con $v_{in}$ y otra arista de capacidad 1 que conecte $v_{out}$ con $u_{in}$
3. Agregamos un nodo source s conectado a los *in* de las esquinas donde haya un alumno
4. Agregamos un nodo sink t conectado a los *out* de las esquinas que son escuelas.

La forma que usamos para representar el grafo es una matriz $C$ de tamaño $V \times V$ con $V = 2N+2$. Esto es así porque por cada nodo vamos a tener su $in/out$ y además necesitamos dos nodos extra para s y t.

Cada posición $i, j$ de $C$ tendrá un valor indicando la capacidad de la arista $i \to j$, que en nuestro caso es siempre 1.

Para construir la matriz tenemos que:

- Crear los nodos $in/out$ para cada esquina: $O(2N)$
- Recorrer cada nodo original para agregar la capacidad 1 entre su $in$ y su $out$: $O(N)$
- Recorrer cada arista $(u, v) \in M$, para agregar la capacidad 1 de la arista $u_{out} \to v_{in}$ y $v_{out} \to u_{in}$: $O(M)$

Por lo que la construcción total de la matriz tiene una complejidad $O(N + M)$.

### 1.2 Reducción a Max-Flow, Min-Cut

La idea es que si ponemos un estudiante de computación en cada punto de corte del Min-Cut, vamos a poder interceptar a todos los posibles estudiantes yendo a sus escuelas. Si existiera un corte en el cual no ponemos a un representante del departamento, tenemos un potencial alumno no interceptado, ya que se podría escapar y llegar a su escuela.

Por lo tanto, este problema es fácilmente reducible al problema de Max-Flow/Min-Cut utilizando el grafo construido.

Por las características de este grafo, todo flujo que parte de s y termina en t, debe tener capacidad 1, ya que todas las aristas tienen capacidad 1.

Los puntos de corte estarán en las aristas que conectan al $v_{in}$ con el $v_{out}$ de una esquina v, ya que al quitar estas aristas de capacidad 1, la capacidad del corte que obtengamos será siempre igual o menor que la capacidad de corte que obtengamos rompiendo conexiones cualquiera entre esquinas (u,v). Esto es así porque de un nodo pueden salir múltiples aristas y no una única como el caso de in/out.

Es por esto que encontrar el Min-Cut en este grafo nos va a dar la mínima cantidad de estudiantes computación que debemos ubicar en determinadas esquinas para poder interrumpir a todo alumno que quiera llegar a la escuela.

## 1.3 Implementación Flujo Máximo

Para resolver el problema en la complejidad pedida, utilizamos el algoritmo de Edmons-Karps.

La idea es hacer Ford-Fulkerson, pero a la hora de buscar los caminos de aumento en el grafo residual, hacerlo en un orden predefinido. Esto se puede lograr utilizando BFS. Es por eso, que por lo visto en clase, este algoritmo tiene una complejidad de $O(VE^2)$.

En el grafo construido tenemos que:

$V = 2N + 2$

$E = 2M + N$ (porque estamos agregando dos aristas por cada calle bidireccional y otra a cada nodo para conectar su $in/out$).

Como toda esquina está conectada con al menos otra esquina por una calle bidireccional, tenemos que:

$N \leq M + 1$, por lo que $2M + N \leq 2M + M + 1$.

Por lo que nos quedaría una complejidad $O((2N + 2)(3M + 1)^2) = O(NM^2)$

## 2 Buenos gráficos - Peso 10

*Así empieza esta respuesta,*
*Al problema del trabajo,*
*El cual mucho pensar trajo,*
*Y mi cabeza protesta.*

*Al leer primero el problema,*
*Pensé cómo modelarlo,*
*Se me ocurrió hacer un grafo,*
*Ya que era uno de los temas.*

*Y cuáles serán los nodos,*
*Qué atribuyo a las aristas,*
*Ya creo que con los artistas,*
*Yo compito codo a codo.*

*¿Si los nodos son acciones?,*
*Fue lo que a mí mismo dije,*
*Aunque todavía no fije,*
*Cuáles son las conecciones.*

*Qué pasa si las aristas,*
*No son direccionadas,*
*De un orden no dice nada,*
*Pero me tira una pista.*

*Que las aristas me indiquen,*
*Que las acciones de ambos,*
*Nodos que aquí conectamos,*
*Puedo poner sin que crucen*

*A esta altura pensé,*
*Que un gráfico permitido,*
*Cuidado si te intimido,*
*Tenía que ser un cliqué.*

*Un máximo cubrimiento,*
*Era yo lo que quería,*
*Y al googlear cómo se hacía,*
*Vi que era NP-Completo.*

*Se derrumbó mi entusiasmo,*
*Quería cambiar de ejercicio,*
*Hacer uno más sencillo,*
*Ponele el del secundario.*

*En mi mente circulaba,*
*El teorema de Dilworth,*
*Excepto que tengas Keywords.*
*Eso no rima con nada,*

*Alta clase la de Mel,*
*Recuerdo que anoté todo,*
*Que no haya celos Leopoldo,*
*Del TAP me gustó tu E.*

*Bueno, basta ya de rimas,*
*Los nodos son las acciones,*
*Flechas representaciones,*
*Que una está por encima.*

*Y qué significa esto,*
*Que A de B esté encima,*
*Es que para cada día,*
*De A era más alto el precio.*

*Para cada par de acciones,*
*Recorro todos los días,*
*Con complejidad pedida,*
*Obtengo las conecciones.*

*Ya lo paso a resolver,*
*Espero haya entretenido,*
*Si quedaste sorprendido,*
*Esperá al otro TP :)*

### 2.1 Modelación

La idea con la que resolví el problema fue planteando un bipartite matching. Ubicaré A nodos de cada lado en el grafo bipartito, uno por cada acción. Llamo "nodos del lado izquierdo" a los que están a la izquierda en el grafo bipartito, y "nodos del lado derecho" a los otros.
Como dice el poema, trazaré una arista de un nodo $U$ del lado izquierdo hacia un nodo $V$ del lado derecho, siempre que la acción del nodo $U$ "sea mayor" que la acción del nodo $V$, definiendo que una acción $U$ es mayor que otra acción $V$ (y escribimos $U > V$) cuando para cada día, el precio de la acción $U$ es mayor al precio de la acción $V$ (en este caso, en un gráfico, la acción $U$ quedaría arriba de la acción $V$). Diré que esa arista es la arista $(U, V)$.
Con esta definición, se puede ver fácilmente que para que dos acciones $U, V$ puedan ir en un mismo gráfico permitido, o bien $U > V$ ó bien $V > U$.

Para resolver el problema entonces, debemos encontrar un máximo matching, es decir, debemos seleccionar aristas de manera que cada nodo tenga a lo sumo una arista seleccionada (aunque la respuesta no será el matching). Veamos bien esto.

## 2.2 Demostración

Supongamos que tenemos un matching, y queremos reconstruir los gráficos permitidos a hacer a partir de las aristas seleccionadas en el matching. Digo que si la arista que conecta a $U$ con $V$ está seleccionada, entonces $U$ y $V$ están juntos en el gráfico.

Ya vimos que si existe la arista, las acciones pueden ir juntas, entonces pasar de un matching a armar los gráficos no tiene complicación.

Veamos ahora que efectivamente, cualquier armado de gráficos permitidos lo podemos traducir a un matching de nuestro grafo.

Sea $G = \{A_1, A_2, ..., A_k\}$ un conjunto de acciones que forma un gráfico permitido. Entonces todas estas acciones están relacionadas dos a dos, es decir, para todo $1 <= i, j <= k$, con $i! = j$, se tiene $A_i > A_j$ ó $A_j > A_i$, ya que si no, no podrían estar en el gráfico G, pero por hipótesis están. Entonces, tenemos un conjunto totalmente ordenado. Luego podemos ordenarlo, y decir que $A_1 > A_2 > ... > A_k$. Bueno, traducir esto a un matching en nuestro grafo es simplemente seleccionar las aristas $(A_i, A_{i+1})$ para todo $1 <= i < k$. Como vemos, acá no hay ningún nodo conectado a dos aristas seleccionadas, dado que las aristas conectan un nodo de la izquierda con uno de la derecha, y ninguna acción aparece en una arista "del mismo lado del paréntesis". Luego, cuando queramos seleccionar las aristas de los otros gráficos, como las acciones que están en $G$ no podrán estar en ningún otro gráfico, tendremos garantizado un matching.

Supongamos que tenemos un matching tales que los gráficos generados son $G_1 = \{A_{1_1}, ..., A_{1_{k_1}}\}, G_2 = \{A_{2_1}, ..., A_{2_{k_2}}\}, ..., G_p = \{A_{p_1}, ..., A_{p_{k_p}}\}$ . Es decir, tenemos $p$ gráficos, y el gráfico $i$, para todo $1 <= i <= p$, contiene $k_i$ acciones, y éstas son $A_{i_1} > ... > A_{i_{k_i}}$. ¿Qué sabemos los valores de las cantidad de acciones de cada gráfico, $k_i$? Sabemos que la suma es el total de las acciones, ya que cada acción aparece exactamente una vez. Es decir, $\sum_{n=1}^{p} k_i = A$.

¿Y cuántas aristas habrá en el matching? Para cada gráfico $G_i$, como dijimos antes, seleccionamos las aristas $(A_{i_j}, A_{i_{j+1}})$ para cada $1 <= j < k_i$. Entonces cada gráfico $G_i$ nos hace seleccionar exactamente $k_i - 1$ aristas. Entonces tenemos $\sum_{n=1}^{p}(k_i - 1) = \sum_{n=1}^{p} k_i - p = A - p$ aristas.

Entonces finalmente, demostramos que dado un armado de gráficos permitidos cualquiera, la cantidad de aristas seleccionadas es $A -$ (cantidad de gráficos). Como A está fijo, minimizar la cantidad de gráficos armados equivale a maximizar la cantidad de aristas seleccionadas en el matching.

## 2.3 Complejidad y Algoritmo

Para encontrar el máximo matching, lo resolveremos como un problema de flujo, como vimos en clase. Uniremos los nodos del lado izquierdo a un nodo ficticio "source", y los del lado derecho a un nodo ficticio "sink", daremos a todas las aristas del grafo capacidad 1 y buscaremos el máximo flujo desde el nodo source al sink. Por como está dispuesto el grafo, y como las aristas tienen capacidad 1, el flujo será la cantidad de aristas seleccionadas en nuestro bipartite matching. El nodo "source" será el nodo 0 (es decir, los vecinos serán los del vector bipartite[0]). El nodo de la acción $i$ del lado izquierdo es el nodo $i+1$ y del lado derecho es el nodo $i+1+A$, para $0 <= i < A$. Y el nodo "sink" es el nodo $2 * A + 1$.

### 2.3.1 Creación del grafo

Como tengo A nodos del lado izquierda, A nodos del lado derecho, el source y el sink, tengo $2*A+2$ nodos. En mi código creo un vector de $2 * A + 2$ vectores, donde el vector de la posición $i$ contiene los nodos que están conectados al nodo $i$ mediante una arista. Crear los nodos tiene complejidad $O(2*A+2)$. Ahora, para crear las aristas entre los nodos del bipartite matching, debemos calcular para cada par de acciones (hay $O(A^2)$ pares), si una es mayor que la otra. Dadas dos acciones, calcular si una es mayor que la otra toma $O(D)$, ya que simplemente tenemos que recorrer los días y ver si el precio de una acción, en cada día, es mayor que el precio de la otra acción, lo que hacemos en $O(1)$ . En el **vector**< **vector**<**int**> > **accion** guardo $A$ vectores, cada uno siendo un vector con los precios de cada día de la acción que que representa. Entonces, crear las aristas del bipartite matching toma $O(A^2.D)$. Y como el source y el sink tienen $A$ aristas cada uno, la creación del grafo tiene complejidad $O(A^2.D + 2 * A) = O(A^2.D)$.

Se crea en el siguiente código:

```
forn(i, A){
        bipartite[SOURCE].pb(i+1); bipartite[i+1].pb(SOURCE);
    // aristas del source a cada accion (lado izquierdo)
        bipartite[i+A+1].pb(SINK); bipartite[SINK].pb(i+A+1);
    // aristas de cada accion (lado derecho) al sink
        forn(j, A){
            if(esMayor(accion[i], accion[j])){ // esMayor tiene complejida O(D)
                bipartite[i+1].pb(j+A+1);
                bipartite[j+A+1].pb(i+1);
            //Por cada arista (u, v), digo que u es vecino de v, y v vecino de u
            }
        }
}
```

### 2.3.2 Búsqueda del flujo máximo

Para resolver esta parte, creé una función **findAugPath**, que busca un camino de aumento en nuestra red de flujos. Esta función inicializa el vector parent de tamaño $A$, que guarda el predecesor de cada nodo en el camino encontrado, inicializa el vector $M$ de tamaño $A$ en un valor "infinito", mayor al flujo de cualquier camino (ya que es lo que va a guardar, el flujo que va agregar al camino que pasa por cada nodo), y luego hace simplemente un BFS en nuestra red, que tiene una complejidad $O(A + A^2) = O(A^2)$, ya que la cantidad de nodos es $O(A)$ y la cantidad de aristas es $O(A^2)$, como vimos anteriormente.

Una vez entonces que encontró un camino de aumento en $O(A^2)$, recorre las aristas el camino encontrado desde el sink hasta el source y actualiza el flujo actual de cada una de ellas (y la que va para el otro lado en la red residual). La complejidad de este recorrido es lineal por la cantidad de aristas del camino, por lo que toma $O(A^2)$. El recorrido lo hace empezando en el nodo "sink" (**int v=SINK;**), y mientras el nodo en el que estoy sea distinto al source, tomo el padre $u$ (lo que significa que la arista $(u, v)$ está en el camino), le sumo el valor **path** a la arista, y le resta **path** a la arista $(v, u)$. El valor de *path* es el flujo máximo en el que se puede aumentar el camino (que siempre será 1 ó 0, ya que las capacidades de todas las aristas son 1. En particular, si es 0 es que no hay camino de aumento y ya encontré el flujo máximo.). Entonces por cada camino de aumento, el algoritmo tiene complejidad $O(A^2)$.

¿Cuántos caminos de aumento puedo buscar? Como cada vez que encuentro un camino de aumento, el flujo aumento en por lo menos 1, y el flujo máximo es $A - 1$ (que corresponde a tener un sólo gráfico permitido con todas las acciones), entonces la búsqueda de caminos de aumento la hago $O(A)$ veces.

Entonces, finalmente, la búsqueda del flujo máximo tiene una complejidad temporal de $O(A^3)$.

Finalmente, el algoritmo que encuentra el máximo matching y por lo tanto resuelve el problema, tiene una complejidad de $O(A^2.(D + A))$, como se pide.

# 3 Cortes programados - Peso 8

## 3.1 Introducción y Modelado

El problema trata sobre una ciudad donde hay esquinas y calles y cortes programados para realizar en la ciudad. Las calles son bidireccionales. Se quiere evaluar la posibilidad de realizar cortes en determinadas calles sin desconectar esquinas de la ciudad. Es decir, que al cortar una calle, se pueda seguir llegando de una esquina cualquiera a otra.

Modelamos las esquinas y las calles con un grafo. Como las aristas son bidireccionales, nuestro grafo es no dirigido.

A continuación detallamos algunas definiciones relacionadas con los grafos no dirigidos:

**Punto de articulación:** es un nodo tal que al removerlo del grafo, la cantidad de componentes conexas aumenta.

**Puente:** es un eje tal que al removerlo del grafo, la cantidad de componentes conexas aumenta.

**Grafo biconexo:** un grafo no dirigido es biconexo si es conexo y no tiene puntos de articulación.

**Componente biconexa:** en un grafo no dirigido, una componente biconexa es un subgrafo biconexo maximal.

El grafo es un vector de vectores de tuplas de enteros, donde el índice del primer vector corresponde a un nodo. En el vector de tuplas tenemos en la primera componente al nodo que conecta con el índice del primer vector y en la segunda componente al número de arista.

## 3.2 Desarrollo

Para resolver el problema a parte de modelar el grafo como lo mencionamos anteriormente, modelamos los puentes y las articulaciones de grafo.
Los puentes los modelamos como un vector de enteros donde los índices se corresponden con el número de arista según aparecen en el STDIN. Un 1 en la i-ésima posición del vector implica que en la arista i-ésima es un puente, caso contrario hay un 0.
Análogamente las articulaciones las modelamos como vectores de enteros donde los índices se corresponden con el número de nodo del STDIN. Un 1 en la í-esima posición del vector de articulaciones implica que ese vértice es una articulación. Si el i-ésimo nodo no es una articulación hay un 0.

Al principio levantamos las información del STDIN armando el grafo como se mencionó. Luego usando la variante de DFS propuesta en clase.
Calculamos para cada nodo la menor distancia de la raíz del árbol de DFS actual a la que es posible saltar desde alguna parte del sub-árbol de DFS con raíz v. Guardamos esta distancia en un vector depth[v].

La menor de estas distancias las guardamos en un vector de índices donde cada índice es un nodo. Está información se puede computar como el mínimo entre el low de los hijos del nodo actual y la profundidad mínima a la que llega un back-edge que sale del nodo actual.

Como el grafo es no dirigido, el DFS construye un DFS-forest con un spanning tree de cada componente conexa.
Como las back-edges siempre están involucradas en un ciclo, un back-edge nunca puede ser puente. La raíz de un árbol de DFS es un punto de articulación sí y solo sí tiene más de un hijo. Los nodos v que no son raíz de un árbol DFS son puntos de articulación sí y solo sí, para alguno de sus hijos w, se cumple que low[w] $\geq$ depth[v].

Agregamos una pila al algoritmo de DFS. Cada vez que recorremos una arista, la agregamos a la pila. Cada vez que volvemos de una llamada recursiva por una arista donde v es padre de w, chequeamos si low[w] $\geq$ depth[v] para saber si ahí termina un componente biconexa.

## 3.3 Complejidad

El costo de armar el grafo es O(m), donde m es la cantidad de aristas. La complejidad temporal del DFS es O(m), donde m es la cantidad de aristas. En nuestro DFS modificado, a parte de agregar la pila para calcular las componentes biconexas, también realizamos un reporte de las componentes por cada puente que vamos encontrando y otro al final del algoritmo. Cada llamado a reportar componentes tiene costo igual a la cantidad de elementos de la pila. Como solo agregamos a la pila la primera vez que vemos una arista, podemos asegurar que la complejidad de todos llamados de reportarComponente será de O(m). Luego la complejidad total del DFS es O(m + m) = O(2m) = O(m).

Las consultas del tipo B y C las podemos responder en tiempo constantes pues tenemos precomputado los puentes, que nos permiten responder accediendo en 0(1) a la posición del vector correspondiente a la arista por la que se está realizando la consulta. También podemos responder las consultas de tipo C en tiempo constante gracias a que calculamos previamente las componentes biconexas.

Las consultas de tipo B son referentes al hecho de que si al cortar la calle $c$, dos o más esquinas dejan de estar conectadas. Para que esto ocurra, la calle, que modelamos como arista, es un puente. Por lo tanto, accediendo a la posición $c$ del vector de puentes podemos saber si es un puente en O(1)

Para las consultas de tipo C se quiere saber si dada una esquina $e$, al cortar una calle cualquiera, $e$ con cuentas esquinas seguirá habiendo camino que es lo que guardamos en el vector componentes. Para el nodo v en la posición v del vector componentes, tenemos la cantidad de nodos conectados a la componente biconexa donde está v, sin contar a v. Lo que nos permite responder cada consulta en O(1).

Para las consultas de tipo A tenemos que sumar los puentes que hay en el camino más corto que hay en el árbol para llegar de A a B yendo todo el tiempo por tree edges. Recorrer el camino más corto tiene complejidad temporal en el peor caso de O(m), donde m es la cantidad de aristas.

# 4 Desocupando el pabellón - Peso 9

*Esas son cosas del Nacional,*
*Eso era lo que pensaba*
*Pero ahora en exactas pasaba*
*Hay una amenaza irracional.*

*Al pabellón como un grafo,*
*representar es sencillo,*
*si la aristas son los pasillos*
*con aulas como nodos, safo.*

*queremos un solución ágil,*
*Busquemos a las biconexas*
*No es un operación compleja*
*con kosaraju sale facil.*

*Debe ser por los de Biolgía*
*Con sus teorías evolutivas,*
*Para algunos, intuitivas,*
*Para otros, una herejía.*

*Las reglas están establecidas:*
*Por los pasillos a un solo lado,*
*Volver, ni por el costado.*
*Así que las aristas, dirigidas.*

*O(A+P) es lo que cuesta*
*Saber de cada aula su componente,*
*Guardemos de manera inteligente*
*Para en O(1) tener la respuesta*

*A protocolos de seguridad,*
*Ahora debemos ayudar,*
*Para rápido poder consultar*
*Dadas aulas, su conectividad.*

*Ahora saber responder debemos,*
*Si a un aula puedes avisar*
*Y por tus cosas regresar*
*Para evitar lo que tememos.*

*Q preguntas respondemos*
*Eso nos da O(Q), yo lo se*
*Total O(Q) + O(A+P)*
*Y la pedida es la que tenemos.*

Como se menciona anteriormente, modelamos el pabellón como un grafo, donde los nodos son las aulas y las aristas (que son dirigidas) son los pasillos. Esto lo guardamos en `pasillosQueSalenDe`. Las consultas que nos piden, es decidir si dos aulas están mutuamente conectadas por pasillos, es decir, se puede ir de la primera a la segunda y desde la segunda a la primera. Esto genera una relación de equivalencia entre grupos nodos: que estén en la misma componente biconexa. Realizar el calculo de estas, es $O(A + P)$. Cuando tenemos las componentes conexas las guardamos en un vector (`componente`), donde guardamos el id de la componente correspondiente al numero de aula. Ver si dos componentes están en la misma componente es ahora $O(1)$, porque es comparar los ids. Como tenemos Q consultas, la complejidad total resulta ser $O(A + P) + Q * O(1) = O(A + P + Q)$

# 5 Código

## 5.1 Alumnos de secundario

```
#include <iostream>
#include <string.h>
#include <queue>
using namespace std;

bool bfs(vector<vector<int> > graph, int s, int t, vector<int> & path){
    int size = graph.size();

    vector<bool> visited;
    visited.resize(size,false);

        queue <int> q;
        q.push(s);
    // marcamos el nodo inicial source como visitado
        visited[s] = true;
        path[s] = -1;
        while (!q.empty()){
                int u = q.front();
                q.pop();

                for (int v=0; v<size; v++){
                        if (visited[v]==false && graph[u][v] > 0){
                                q.push(v);
                                path[v] = u;
                                visited[v] = true;
                        }
                }
        }

        // si llegamos al sink quiere decir que hay un camino
        return (visited[t] == true);
}

int edmons(vector<vector<int> > graph, int s, int t)
{
        int u, v;
    int size = graph.size();

  // vector para guardar el camino de aumento de la red residual
  vector<int> path;
  path.resize(size);
        int maxFlow = 0;

        while (bfs(graph, s, t, path))
        {
    // el flujo no puede ser mayor a la cantidad de vertices
                int pathFlow = size;
                for (v=t; v!=s; v=path[v])
                {
                        u = path[v];
      // Actualizo el flow del path
                        pathFlow = min(pathFlow, graph[u][v]);
                }
```

```
                // hacemos el update del flow del grafo recorriendo cada arista de au
                for (v=t; v != s; v=path[v])
                {
                        u = path[v];
                        graph[u][v] -= pathFlow;
                        graph[v][u] += pathFlow;
                }
                // Sumamos al flow total
                maxFlow += pathFlow;
        }
        return maxFlow;
}

int main()
{
  int N;
  cin >> N;
  int M;
  cin >> M;
  // Construyo una matriz para representar al grafo
  //  que tenga el doble del tamano que N mas 2 por s y t.
  // esto es porque cada nodo va a tener un in y un out
  vector<vector<int> > graph(2*N+2);
  for ( int i = 0 ; i < 2*N+2 ; i++ ) graph[i].resize(2*N+2);
  int s = 0;
  int t = N*2+1;
  char tipo;
  int esq1, esq2;
  for(int i = 1; i<N+1; i++){
    cin >> tipo;
    // conecto In con Out
    graph[2*i-1][2*i] = 1;
    // si es alumno lo conecto al s
    if(tipo == 'A'){
      // conecto el s a cada In de los nodos A
      graph[s][2*i-1] = 1;
    }
    // si es escuela lo conecto al t
    if(tipo == 'E'){
      // conecto el out de los nodos E a t
      graph[2*i][t] = 1;
    }
  }
  // Conecto las esquinas mediante las M calles bidireccionales
  for(int i = 0; i<M ; i++){
    cin >> esq1;
    cin >> esq2;
    // el out de la esq1 lo conecto con el in de la esq2
    graph[2*esq1][2*esq2-1] = 1;
    // el out de la esq2 lo conecto con el int de la esq1
    graph[2*esq2][2*esq1-1] = 1;
  }

  cout << edmons(graph, s, t);
        return 0;
}
```

## 5.2 Buenos Gráficos

```
#include<iostream>
#include<math.h>
#include<algorithm>
#include<vector>
#include<queue>

using namespace std;

#define forn(i,n) for(int i=0;i<(int)(n); i++)
#define forsn(i,s,n) for(int i=(s);i<(int)(n); i++)
#define pb push_back

bool esMayor(vector<int> a1, vector<int> a2){
        forn(i, a1.size()){
                if(a1[i]<=a2[i])return false;
        }
        return true;
}

vector< vector<int> > capacidades, flujos;
vector<int> parent;

int findAugPath(vector< vector<int> > bipartite, int s, int t){
        parent.clear();
        forn(i, bipartite.size()){
                parent.pb(-1);
        }
        parent[s]=-1;
        const int INF_FLOW=3;
        vector<int> M(parent.size(), INF_FLOW);
        queue<int> q;
        q.push(s);
        while(!q.empty()){
                int estoy=q.front(); q.pop();
                forn(i, bipartite[estoy].size()){
                        int vecino=bipartite[estoy][i];
                        if(capacidades[estoy][vecino] - flujos[estoy][vecino]>0 &&
                             parent[vecino]==-1 ){
                                parent[vecino]=estoy;
                                M[vecino]=min(
                                     M[vecino],
                                      capacidades[estoy][vecino]-flujos[estoy][vecino]
                                );
                                if(vecino!=t){
                                        q.push(vecino);
                                }else{
                                        return M[t];
                                }
                        }
                }
        }
}
```

```
int main (){
        int A, D;
        cin>>A>>D;
        vector< vector<int> > bipartite(2*A+2);
        const int SOURCE=0;
        const int SINK=2*A+1;
        vector< vector<int> > accion(A);
        forn(i, A){
                vector<int> ac(D);
                forn(j, D){
                        int precio;
                        cin>>precio;
                        ac[j]=precio;
                }
                accion[i]=ac;
        }
        forn(i, 2*A+2){
                vector<int> newCap(2*A+2, 0), newFlow(2*A+2, 0);
                capacidades.pb(newCap);
                flujos.pb(newFlow);
        }
        forn(i, A){
        // aristas del source a cada accion (lado izquierdo)
                bipartite[SOURCE].pb(i+1); bipartite[i+1].pb(SOURCE);
                capacidades[SOURCE][i+1]=1;
        // aristas de cada accion (lado derecho) al sink
                bipartite[i+A+1].pb(SINK); bipartite[SINK].pb(i+A+1);
                capacidades[i+A+1][SINK]=1;
                forn(j, A){
                        if(esMayor(accion[i], accion[j])){
            // Esta funcion tiene un for de 1 a D con operaciones de O(1),
            // por lo que tiene complejidad O(D).
                                bipartite[i+1].pb(j+A+1);
                // agrego la arista para la vuelta del flujo.
                                bipartite[j+A+1].pb(i+1);
                                capacidades[i+1][j+A+1]=1;
                        }
                }
        }
        int flow=0;
        int path;
        while((path=findAugPath(bipartite, SOURCE, SINK)) && path>0){
                flow+=path; // aca, path siempre va a ser 1
                int v=SINK;
                while(v!=SOURCE){
                        int u=parent[v];
                        flujos[u][v]+=path;
                        flujos[v][u]-=path;
                        v=u;
                }
        }
        cout<<A-flow<<endl;
}
```

### 5.3 Cortes Programados

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <stack>
#include <queue>
using namespace std;
typedef pair<int,int> PII;

void reportarcomponente(stack<PII> &pila,PII par, vector<int> &componente)
{
    stack<int> miembros;
    if (pila.size()>0)
    {
        miembros.push(pila.top().second);
        while ((pila.top().first!=par.first)&&(pila.top().second!=par.second)
                        &&(pila.size()>1)){
            pila.pop();
            miembros.push(pila.top().second);
        }
        pila.pop();
    }
    int k=miembros.size();
    //cout<<par.first+1<<" "<<par.second+1<<" "<<" "<<k<<endl;
    for (int i=0;i<k;i++)
    {
        componente[miembros.top()]=k-1;
        if ((par.first==0)&&(par.second==0)){
            componente[miembros.top()]++;
        }
        miembros.pop();
    }
}

int dfs(vector<vector<PII> > &G, vector<int> &puente,vector<int> &articulacion,
vector<int> &depth, vector<int> &low, vector<int> &componente, int v,int d,
vector <int> &padre, stack <PII> &pila)
{
    depth[v]=d;
    low[v]=d;

    if (v != padre[v]){
        pila.push(make_pair(padre[v],v));
    }
        int w;
    for (unsigned int i=0;i<G[v].size();i++)
    {
        w=G[v][i].first;

        if (w!=padre[v]){
            if (depth[w] == -1){
                padre[w]=v;
                low[v] = min(low[v], dfs(G, puente, articulacion, depth , low,
                        componente, w, d+1,padre , pila));
                if ((low[w]>=depth[v])&&(v!=0)){
                    articulacion[v]=1;
```

```
                }
                if (low[w]>=depth[w]){
                    puente[G[v][i].second]=1;
                    reportarcomponente(pila, make_pair(v,w), componente);
                }
            }
            else{
                low[v] = min(low[v],depth[w]);
            }
        }
    }

    if (v==0){
        reportarcomponente(pila, make_pair(0,0), componente);//me aseguro vaciarla
        if (G[0].size()>1){
            articulacion[0]=1;
        }
    }

    return(low[v]);
}

int queryA (vector<vector<PII> > &G, vector <int> &visitado, vector<int> &puente,
                int v, int a, int b,int &activador)
{
    visitado[v] = 1;
    int resultado=0;
    int entre=0;
    activador=0;
    for (unsigned int i=0;i<G[v].size();i++)
    {
        int w=G[v][i].first;
        if (visitado[w] == 0)
        {
            int x=0;
            resultado=resultado+queryA(G,visitado, puente,w,a,b,x);
            activador=activador+x;
            if ((x==1)&&(puente[G[v][i].second]==1)){
                resultado++;
            }
            entre=1;
        }
    }

    if (entre==0){
        resultado=0;
    }

    if (v==b || v==a){
        activador++;
    }

    return resultado;
}

int main()
{
    int n,m;
```

```
    cin>>n>>m;
    vector<PII> vacio;
    vector<vector<PII> > G (n,vacio);
    vector<int> puente(m,0);
    vector<int> articulacion (n,0);
    vector<int> depth (n,-1);
    vector<int> low (n, n+1);
    vector<int> componente (n,0);
    vector <int> padre (n,0);//quiero que la raiz tenga como padre a si mismo
    stack <PII> pila;//pila vacia
    int u,v;
    for(int i=0;i<m;i++)
    {
        cin>>u>>v;
        G[u-1].push_back(make_pair(v-1,i));
        G[v-1].push_back(make_pair(u-1,i));
    }
    //cout<<"dfs inicio"<<endl;
    dfs(G, puente, articulacion, depth , low, componente, 0, 0,padre , pila);
    int q;
    cin>>q;
    char t;
    int a,b;

    vector<int>visitado (n, 0);

     for(int i=0;i<q;i++){
        cin>>t;
        if (t=='A')
        {
            fill(visitado.begin(), visitado.end(), 0);
            cin>>a>>b;
            int x=0;
            cout<<queryA(G, visitado, puente, 0, a-1, b-1, x)<<endl;
        }
        else if (t=='B'){
            cin>>a;
            cout<<puente[a-1]<<endl;
        }
        else if (t=='C'){
            cin>>a;
            cout<<componente[a-1]<<endl;
        }
    }

    return(0);
}
```

## 5.4 Desocupando el Pabellón

```
#include <bits/stdc++.h>

using namespace std;

typedef vector<vector<int>> grafo;

int cantidadDeAulas, cantidadDePasillos;
grafo pasillosQueSalenDe;
grafo grafoTranspuesto;

vector<bool> visitado;
stack<int> st;

vector<int> componente;

void identificarComponentes(int v, int id){
    visitado[v] = true;
    componente[v] = id;

    for (int i: grafoTranspuesto[v])
        if (!visitado[i]) identificarComponentes(i, id);
}

grafo obtenerTranspuesta(grafo & g){
    grafo transpuesta(g.size());
    for (int s = 0; s < (int)g.size(); s++) {
        for(int d: g[s]) {
            transpuesta[d].push_back(s);
        }
    }
    return transpuesta;
}

void ordenarPorFinalizacion(int s){
    visitado[s] = true;
    for(int d: pasillosQueSalenDe[s]){
        if(!visitado[d])
            ordenarPorFinalizacion(d);
    }
    st.push(s);
}

void kosaraju() {

    fill(visitado.begin(), visitado.end(), false);

    for(int i = 0; i < cantidadDeAulas; i++)
        if(!visitado[i]) ordenarPorFinalizacion(i);

    grafoTranspuesto = obtenerTranspuesta(pasillosQueSalenDe);

    fill(visitado.begin(), visitado.end(), false);

    int id = 0;
    while (!st.empty()) {
```

```
        int v = st.top(); st.pop();
        if (!visitado[v]){
            identificarComponentes(v, id);
            id++;
        }
    }
}


int main(int argc, char **argv) {

        cin >> cantidadDeAulas >> cantidadDePasillos;

        pasillosQueSalenDe.resize(cantidadDeAulas);
        visitado.resize(cantidadDeAulas, false);
        componente.resize(cantidadDeAulas, -1);

        for (int pasillo = 0; pasillo < cantidadDePasillos; pasillo++) {
                int desde, hasta;
                cin >> desde >> hasta;
                pasillosQueSalenDe[desde-1].push_back(hasta-1);
        }

        kosaraju();

        int cantidadDeConsultas;
        cin >> cantidadDeConsultas;
        for (int consulta = 0; consulta < cantidadDeConsultas; consulta++) {
                int origen, destino;
                cin >> origen >> destino;
                cout << (componente[origen-1]==componente[destino-1] ? "S" : "N")
                                                                        << endl;
        }


        return 0;
}
```