Problemas, Algoritmos, y Programación
Trabajo Práctico 2

Manindra Agrawal

Segundo Cuatrimestre 2016

Índice

1. Problema A:Alumnos de secundario **3**
- 1.1. Descripción del problema . . . 3
- 1.2. Soluciones al problema . . . 3
- 1.3. Algoritmo . . . 4
- 1.4. Código de la solución . . . 4
- 1.5. Casos de prueba . . . 8

2. Problema B: Buenos gráficos **9**
- 2.1. Descripción del problema . . . 9
- 2.2. Solución al problema . . . 9
- 2.3. Algoritmo . . . 9
- 2.4. Código de la solución . . . 11

3. Problema C: Cortes programados **14**
- 3.1. Descripción del problema . . . 14
- 3.2. Soluciones al problema . . . 14
- 3.3. Algoritmo . . . 16
- 3.4. Casos de prueba . . . 18
- 3.5. Código de la solución . . . 19

4. Problema D: Desocupando del pabellón **22**
- 4.1. Descripción del problema . . . 22
- 4.2. Soluciones al problema . . . 22
  - 4.2.1. Enfoque naïve . . . 22
  - 4.2.2. Enfoque con componentes fuertemente conexas . . . 22
- 4.3. Algoritmo y Análisis de Complejidad . . . 22
- 4.4. Código de la solución . . . 24

5. Bibliografía **26**

1. Problema A: Alumnos de secundario

Peso del ejercicio: 8

1.1. Descripción del problema

El problema consiste en una ciudad con N esquinas y M calles bidirecciones que las conectan. Con esta primera frase en la descripción del problema, ya se puede asumir el problema se trata de un problema dentro de la **Teoría de Grafos**. De esta manera, representamos a la red con un grafo $G = \langle V, E\rangle$ con N nodos y M aristas, que va a ser conexo y donde todos los nodos van a estar en alguno de los tres conjuntos disjuntos A, E, y X.

Con esta abstracción de los datos, lo que queremos encontrar es el tamaño del menor conjunto de nodos D tal que por cada camino entre escuelas $x_1, \ldots, x_n$ donde $x_1 \in A$, $x_n \in E$, $x_i \in V\ \forall i \in [1, n]$, y $\langle x_i, x_{i+1}\rangle \in E\ \forall i\, [1, n-1]$, exista in i tal que $x_i \in D$.

1.2. Soluciones al problema

La descripción de este problema se presta facilmente al problema de buscar al tamaño del mínimo corte, que es igual al del flujo máximo[1, *theorem 26.6*]. Sin embargo, la versión intuitiva del problema — que conecta todas las esquinas con alumnos $x \in A$ con un nodo fuente S, y todas las esquinas con escuelas $x \in E$ con un nodo sumidero T, se le asigna 1 de capacidad de todas las aristas, y se calcula el flujo máximo de S a T — no funciona, porque estamos intentando poner alumnos de ciencias de la computación en esquinas (nodos), y este problema busca el corte mínimo por aristas.

Para resolver este problema, lo que sí se puede hacer es separar a las esquinas entre dos nodos, llamados *left* y *right* que tengan una arista dirigida entre ellos, y que todas las aristas entrantes a un nodo pasen por *left*, y las salientes salgan por *right*. De esta manera, se genera el grafo G' de la siguiente forma.

$$G' = \langle V', E'\rangle \tag{1}$$

$$V' = \{x_{\text{left}} \mid x \in V\} \cup \{x_{\text{right}} \mid x \in V\} \cup \{S, T\} \tag{2}$$

$$E' = \begin{pmatrix} \{\langle x_{\text{right}}, y_{\text{left}}\rangle \mid \langle x, y\rangle \in E\} \\ \cup \{\langle x_{\text{left}}, x_{\text{right}}\rangle \mid x \in V\} \\ \cup \{\langle S, x_{\text{left}}\rangle \mid x \in A\} \cup \{\langle x_{\text{right}}, T\rangle \mid x \in E\} \end{pmatrix} \tag{3}$$

Se definen las capacidades de todas las aristas de la forma $\langle x_{\text{left}}, x_{\text{right}}\rangle$ como 1. Esto significa que, como las únicas aristas entrantes a los nodos *right* en G' tienen capacidad 1, todas las aristas salientes que no salen de S salen de un nodo *right*, y no hay una arista entre S y T, cualquier arista que no salga de S va a tener flujo máximo 1, y la cantidad de flujo que pase por las que sí nunca se va a pasar este valor. Por esta razón, aunque según la definición del problema todas las aristas que no sean de esa forma deberían tener capacidad ∞, por simplicidad les asignamos capacidad 1 con los mismos resultados.

1.3. Algoritmo

Para resolver el problema de corte mínimo/flujo máximo en G', usamos el algoritmo de **Edmonds-Karp**[1], que usa breadth-first search para encontrar el camino de augmento. Este algoritmo nos garantiza una complejidad de $\mathcal{O}(|V| \cdot |E|^2)$[1, *theorem 26.8*], que en nuestro caso sería igual a la ecuación 4.

$$\mathcal{O}\left(|V'| \cdot |E'|^2\right) = \mathcal{O}\left((2 + 2 \cdot N) \cdot (N + M + |A| + |E|)^2\right) = \mathcal{O}(N \cdot M^2) \quad (4)$$

Por otro lado, Edmonds-Karp está acotado por la complejidad del algoritmo de Ford-Fulkenson $\mathcal{O}(f \cdot |E|)$, donde f es la cantidad de flujo máximo que pasa por la red[1, *theorem 26.8*]. Como la capacidad de todas las aristas es 1, el flujo máximo está acotado por N, y la complejidad final pasa a ser $\mathcal{O}(N \cdot M)$.

1.4. Código de la solución

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <queue>
#include <experimental/optional>

using namespace std;
using namespace std::experimental;

typedef int Node;
typedef vector <vector <Node>> Graph;

// Simula una red de flujo en un grafo sin pesos.
// Esta red es igual a la del grafo original excepto por agrega dos nodos como
// fuente y sumidero, y todos los otros nodos se separan entre "left" y
// "right", con una arista del primero al segundo.
// Para todas las aristas entrantes a cualquier nodo n del grafo original hay
// una arista entrante a left(n); para todas las aristas salientes hay una
// arista saliente de right(n).
// Tambien se pueden agregar aristas de la fuente y al sumidero.
struct FlowNetwork
{
  Graph G;
  const Node S, T;

  // Dar el corte minimo del grafo usando el algoritmo de Edmonds-Karp.
  int MinCut()
  {
    // Flujo que hay entre cada par de nodos <x, y>.
    // Cada elemento puede ser igual a 0 (no pasa flujo), 1 (pasa flujo
    // de x a y), o -1 (pasa flujo de y a x).
    // flow[x][y] == 1 sii flow[y][x] == -1.
    vector <vector <int>> flow(G.size(), vector <int> (G.size(), 0));
```

```
  // Buscar caminos de augmento que mejoren el flujo maximo usando BFS.
  // La cantidad de caminos de augmento esta acotada en N * M en el
  // caso general (Cormen, teorema 26.8). De hecho, como hay una nueva
  // arista dirigida con peso 1 por cada uno de los N nodos originales
  // el flujo maximo F esta acotado por N, y esto tambien acota la
  // cantidad de caminos en N. Por lo tanto, hay O(N) caminos.
  // Cada iteracion tarda O(M + N) = O(M), por lo que esta parte
  // del algoritmo tiene una complejidad temporal de O(MN).
  int maxFlow = 0;
  while (true)
  {
    // El predecesor de cada nodo en el camino de augmento.
    vector<optional<Node>> pred(G.size());

    // Hacer DFS desde la fuente hasta el sumidero.
    // Esto visita una vez cada nodo, y por cada nodo recorre
    // todas sus aristas, por lo que tarda O(M) iteraciones.
    queue<Node> q;
    q.push(S);
    while (!q.empty())
    {
      Node c = q.front(); q.pop();
      for (auto &e : G[c])
      {
        // El siguiente nodo puede ser parte del camino de augmento
        // si no es parte del camino actual ni de alguno anterior.
        if (!pred[e] && e != S && flow[c][e] < 1)
        {
          pred[e] = c;
          q.push(e);
        }
      }
    }

    // Si no se encontro ningun camino de augmento hasta el sumidero,
    // el algoritmo ya tiene el corte optimo.
    if (!pred[T])
      break;

    // Recorrer el camino de augmento que creamos y buscar la minima capacidad
    // disponible. Esta va a ser la capacidad del camino.
    // Como un camino no puede pasar dos veces por el mismo nodo, esto tarda
    // O(N) iteraciones.
    int df = 1 - flow[T][*pred[T]];
    for (Node e = *pred[T]; pred[e]; e = *pred[e])
      df = min(df, 1 - flow[e][*pred[e]]);

    // Cambiar las capacidades de estas aristas.
    for (Node e = T; pred[e]; e = *pred[e])
    {
      flow[*pred[e]][e] += df;
      flow[e][*pred[e]] -= df;
    }

    maxFlow += df;
  }
```

```
    return maxFlow;
  }

  // Un nodo "left" es un nodo que tiene muchas aristas entrantes y una sola
  // arista saliente, al correspondiente nodo "right".
  // Por simplicidad, la fuente y el sumidero cuentan como nodos
  // "left" y "right" a la vez.
  Node left(int n)
  {
    if (n == S || n == T)
      return n;

    return 2 * n;
  }

  // Un nodo "right" es un nodo que tiene muchas aristas salientes y una sola
  // arista enrante, al correspondiente nodo "left".
  // Por simplicidad, la fuente y el sumidero cuentan como nodos
  // "left" y "right" a la vez.
  Node right(int n)
  {
    if (n == S || n == T)
      return n;

    return 2 * n + 1;
  }

  // Agrega un par de aristas dirigidas entre el nodo "right" correspondiente a
  // un valor y el nodo "left" correspondiente a otro, y viceversa.
  void addEdge(int f, int t)
  {
    G[right(f)].push_back(left(t));
    G[right(t)].push_back(left(f));
  }

  // Crea una fuente, un sumidero, y un par de nodos por cada uno de los n nodos.
  FlowNetwork(int n)
  : S(2 * n), T(2 * n + 1)
  {
    G = Graph(2 * n + 2);
    for (int i = 0; i < n; i++)
      G[left(i)].push_back(right(i));
  }
};

int main()
{
  int n, m;
  while (cin >> n >> m)
  {
    FlowNetwork fn(n);
    for (int i = 0; i < n; i++)
    {
      // Si un nodo es una escuela, agregar una arista de su nodo
      // correspondiente al sumidero. Si es un alumno, conectarla
      // a la fuente.
```

```
      char c; cin >> c;
      if (c == 'E')
        fn.addEdge(i, fn.T);
      if (c == 'A')
        fn.addEdge(fn.S, i);
    }

    for (int i = 0; i < m; i++)
    {
      int a, b; cin >> a >> b; a -= 1; b -= 1;
      fn.addEdge(a, b);
    }

    cout << fn.MinCut() << endl;
  }

  return 0;
}
```

1.5. Casos de prueba

Caso de prueba	Archivo de entrada	Salida esperada
Hay solo un alumno y una escuela	2 1 A E 1 2	1
Hay un solo alumno y una sola escuela con muchos posibles caminos	7 10 A X X X X X E 1 2 1 3 1 4 1 5 1 6 2 7 3 7 4 7 5 7 6 7	1
Hay muchos caminos posibles, pero una solución greedy suele no lograr un corte óptimo	12 14 A A A A A X X E X E E E 1 6 6 7 7 8 2 6 3 6 4 6 5 6 2 9 3 9 4 9 5 9 9 10 9 11 9 12	2
Hay una cantidad muy grande de nodos[3]	`ej1NoTanGrande.in`	El programa termina en menos de 1 segundo y con un resultado razonable.
Hay una cantidad muy grande de nodos y de aristas en un grafo fuertemente conexo	`ej1Grande.in`	El programa termina en menos de 1 segundo[4] y con un resultado razonable.

[3] No probamos que este grafo es conexa, pero el programa da la solución correcta igualmente.
[4] De hecho, termina en 0.4 segundos, lo que es más rápido que lo que indica la complejidad.

2. Problema B: Buenos gráficos

Peso del ejercicio: 9

2.1. Descripción del problema

Dados los precios de A acciones durante D dias, se nos pregunta la mínima cantidad de gráficos para graficar los precios de estas acciones en el tiempo de tal manera que los precios no se intersecten. Si los precios de dos acciones no se intersectan pueden dibujarse en un mismo gráfico.

2.2. Solución al problema

Nos construimos un grafo donde los vertices representan las acciones y el vertice i esta conectada con el vertice j si los precios de la acción j son estrictamente mayores que los precios de la acción i. Simbolicamente $G = (V, E)$ donde $V = \{a_1, \dots, a_A\}$ y si $P_{i,t}$ denota al precio de la accioń i-esima a tiempo t entonces $(i, j) \in E$ si y solo si $P_{i,t} < P_{j,t}$ para todo $1 \leq t \leq D$.

El grafo es dirigido claramente, es transitivo pues si (i, j),$(j, k) \in E$ entonces $P_{i,t} < P_{j,t}$ y $P_{j,t} < P_{k,t}$ para todo $1 \leq t \leq D$ entonces $P_{i,t} < P_{k,t}$ para todo $1 \leq t \leq D$ entonces $(i, k) \in E$. También es aciclico pues si i y j estan en un ciclo por transitividad vale que (i, j),$(j, i) \in E$ entonces $P_{i,t} < P_{j,t}$ y $P_{j,t} < P_{i,t}$, absurdo. Luego G es un DAG.

Si llamamos r a la cantidad mínima de caminos para cubrir el DAG y s a la cantidad mínima de gráficos para repartir las acciones sin que haya solapamiento, quiero ver que $r = s$.

Notamos $g_1, \dots, g_s$ a los gráficos, a cada gráfico g_k le corresponde un conjunto de acciones $a_{j_1}, \dots, a_{j_{N_k}}$ que no se solapan por lo que puedo suponer que estan ordenadas de menor a mayor, con respecto a los precios, entonces $c_k = ((j_1, j_2), \dots, (j_{N_k-1}, j_{N_k}))$ es un camino en G. De esta manera me construyo $c_1, \dots, c_s$ caminos y cubren G pues toda acción esta algún gráfico entonces toda acción esta en algun camino. Luego $r \leq s$.

Sea $c_1, \dots, c_r$ el menor cubrimiento por caminos del grafo. Si $a_{j_1}, \dots, a_{j_{N_k}}$ son los elementos de camino c_k ordenados de menor a mayor entonces $P_{i,t} < P_{i+1,t}$ para todo $1 \leq t \leq D$ y para todo $1 \leq i \leq N_k - 1$, luego me construyo el grafico g_k con estas acciones, no se solapan pues la diferencia entre los precios es positiva. Luego $s \leq r$. Luego $r = s$.

2.3. Algoritmo

Dadas n acciones $v_1, \dots, v_n$ me construyo el grafo bipartito duplicando los vertices originales. Ahora tengo dos conjuntos de vertices $\{v_1, \dots, v_n\}$ y $\{u_1, \dots, u_n\}$ y (i, j) es una arista del grafo si $P_{i,t} < P_{j,t}$ para todo $1 \leq t \leq D$. Si m es el matching maximo de este grafo bipartito, que encontramos usando el algoritmo

Esto es resultado de que los algoritmos de flujo suelen ser rápidos en casos no patológicos.

de Edmonds Karp, entonces por los visto en la clase de flujo la minima cantidad de caminos para cubrir G, el DAG de las acciones, es $n - m$.

Detectar cuales vertices se conectan cuesta $\mathcal{O}(A^2 D)$ y como el grafo bipartito tiene $2A$ nodos y A^2 aristas la complejidad de calcular matching maximo es $\mathcal{O}(A^3)$. En total el algoritmo tiene una complejidad de $\mathcal{O}(A^2(A + D))$ como se pedia en el enunciado.

2.4. Código de la solución

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <queue>
#include <unordered_map>
#include <experimental/optional>

using namespace std;
using namespace std::experimental;

typedef int Node;
typedef vector <vector <Node>> Graph;

// Dar el corte mínimo del grafo usando el algoritmo de Edmonds-Karp.
int MinCut( Graph &G,Node S,Node T)
{
  // Flujo que hay entre cada par de nodos <x, y>.
  // Cada elemento puede ser igual a 0 (no pasa flujo), 1 (pasa flujo
  // de x a y), o -1 (pasa flujo de y a x).
  vector <vector <int>> flow(G.size(), vector <int> (G.size(), 0));

  // Buscar caminos de augmento que mejoren el flujo máximo usando BFS.
  // La cantidad de caminos de augmento está acotada en N * M en el
  // caso general (Cormen, teorema 26.8). De hecho, como hay una nueva
  // arista dirigida con peso 1 por cada uno de los N nodos originales
  // el flujo máximo F está acotado por N, y esto también acota la
  // cantidad de caminos en N. Por lo tanto, hay O(N) caminos.
  int maxFlow = 0;
  while (true)
  {
    // El predecesor de cada nodo en el camino de augmento.
    vector <optional<Node>> pred(G.size());

    // Hacer DFS desde la fuente hasta el sumidero.
    // Esto visita una vez cada nodo, y por cada nodo recorre
    // todas sus aristas, por lo que tarda O(M) iteraciones.
    queue <Node> q;
    q.push(S);
    while (!q.empty())
    {
      Node c = q.front(); q.pop();
      for (auto &e : G[c])
      {
        // El siguiente nodo puede ser parte del camino de augmento
        // si no es parte del camino actual ni de alguno anterior.
        if (!pred[e] && e != S && flow[c][e] < 1)
        {
          pred[e] = c;
          q.push(e);
        }
      }
    }

    // Si no se encontró ningún camino de augmento hasta el sumidero,
    // el algoritmo ya tiene el corte óptimo.
    if (!pred[T])
```

```
        break;

      // Recorrer el camino de augmento
      int df = 1 - flow[T][*pred[T]];
      for (Node e = *pred[T]; pred[e]; e = *pred[e])
        df = min(df, 1 - flow[e][*pred[e]]);

      for (Node e = T; pred[e]; e = *pred[e])
      {
        flow[*pred[e]][e] += df;
        flow[e][*pred[e]] -= df;
      }

      maxFlow += df;
    }

    return maxFlow;
}

int main(){
    int a,d;cin>>a>>d;
    vector<vector<int> > p(a,vector<int>(d));
    for(int i=0;i<a;i++)
      for(int j=0;j<d;j++)
        cin>>p[i][j];

    Graph G(2*a+2);
    Node S=0;Node T=2*a+1;

    //Conectamos la fuente y el sumidero
    for (int i = 0; i < a; i++)
    {
      G[S].push_back(i+1);
      G[i+1].push_back(S);
      G[i+1+a].push_back(T);
      G[T].push_back(i+1+a);
    }


    //Conectamos la primera columna con la segunda
    //Este procesamiento tiene complejidad O(a*a*d)
    for(int i=0;i<a;i++)
      for(int j=0;j<a;j++){
        if(j==i)continue;
        bool loAgrego = true;
        for(int t=0;t<d-1;t++){
          if(p[i][t]>p[j][t] && p[i][t+1]<p[j][t+1]){
            //si se cruzan en el medio
            loAgrego = false;
            break;
          }
          else if(p[i][t]<p[j][t] && p[i][t+1]>p[j][t+1]){
            //si se cruzan en el medio
            loAgrego = false;
            break;
          }else if(p[i][t]==p[j][t] || p[i][t+1]==p[j][t+1]){
            //si se cruzan en las puntas
```

```
          loAgrego = false;
          break;
        }else if(p[i][t]>p[j][t]){
          //solo agrego si la acción i-esima esta por debajo de la acción j-esima
          loAgrego = false;
          break;
        }
      }
      if(loAgrego){
        //Agrego solo si estoy por debajo de j
        G[i+1].push_back(j+1+a);
        G[j+1+a].push_back(i+1);
      }
    }
  //Calculamos el flujo maximo, como la cantidad de vertices
  //es 2a y la cantidad de aristas como máximo es a*a entonces
  //la complejidad es O(a*a*a)
  int k = MinCut(G,S,T);
  cout <<a-k<<endl;
  return 0;
}
```

3. Problema C: Cortes programados

Peso del ejercicio: 9

3.1. Descripción del problema

Dada una ciudad con N esquinas y M calles bidereccionales que conectan pares de esquinas, donde se puede viajar entre cualquier par de esquinas usando las calles, se pide responder una serie de queries de varios tipos:

- Tipo A: dadas 2 esquinas e_1 y e_2, dar la cantidad de calles tales que, si cortáramos únicamente esa calle, impediría viajar desde e_1 hasa e_2.
- Tipo B: dada una calle, devolver si existen al menos 2 esquinas entre las que dejaría de haber camino si cortáramos la calle.
- Tipo C: dada una esquina e, devolver la cantidad de esquinas e_2 tales que, de cortar una sola calle cualquiera, seguiría habiendo camino entre e_1 y e_2.

Se nos pide diseñar un algoritmo que resuelva el problema en complejidad temporal $\mathcal{O}(M + MQ_A + Q_B + Q_C)$, donde Q_A, Q_B y Q_C son las queries de tipo A, B y C respectivamente.

3.2. Soluciones al problema

Modelaremos la ciudad como un grafo, de la manera clásica, donde los nodos serán las esquinas de la ciudad, y las aristas serán las calles (esto tiene sentido porque las calles conectan *pares* de esquinas). Como sabemos que entre todo par de esquinas hay un camino, esto quiere decir que el grafo que obtenemos es conexo, y por tanto además $\mathcal{O}(N) \in \mathcal{O}(M)$.

Para resolver el problema, lo que haremos será calcular los **puentes** en el grafo resultante, y responderemos las queries utilizando información que obtendremos en base a los puentes.

Veamos cómo se pueden responder los distintos tipos de queries usando los puentes el grafo:

Tipo A:

La query nos habla de aristas que al ser removidas individualmente impedirían llegar desde e_1 hasta e_2. De alguna manera nos habla de los puentes del grafo, porque esas son las aristas que al ser removidas aumentan la cantidad de componentes conexas y cortan caminos entre nodos.

De modo que la query nos habla de los puentes, pero no de todos, sino de **los puentes que se encuentran en un camino simple desde e_1 hasta e_2**.

Si bien puede haber más de un camino simple entre e_1 y e_2, **no pueden existir dos que usen distintos puentes**. Si existiesen dos, entonces tendría que haber un ciclo conteniendo a esos puentes, lo cual es un absurdo. De modo que los puentes en un camino simple de e_1 a e_2 son únicos, y es claro que cumplen con la condición que buscamos porque necesariamente al remover uno quedan separados e_1 y e_2, de lo contrario habría otro camino que los mantiene conectados y esto haría que el puente pertenezca a un ciclo, que también es un absurdo.

Para responder las queries entonces teniendo los puentes del grafo precomputados podemos encontrar un camino simple de e_1 a e_2 con un BFS, y luego iterar sobre dicho camino y contar la cantidad de puentes en el. El BFS siempre encuentra camino (porque el grafo es conexo) y tiene complejidad $\mathcal{O}(N+M) = \mathcal{O}(M)$. Luego Q_A queries de tipo A tienen complejidad $\mathcal{O}(MQ_A)$.

Tipo B:

La query nos pide devolver para una arista dada, si al removerla hay al menos 2 nodos que quedan desconectados. Como todos los nodos empiezan conectados por ser el grafo conexo, la única manera de que al remover una arista se desconecten nodos es que aumente la cantidad de componentes conexas, y **esto ocurre únicamente al remover un puente** (por definición).

Teniendo los puentes del grafo precomputados, dada una arista podemos responder en $\mathcal{O}(1)$ si es o no un puente, luego Q_B queries de tipo B tienen complejidad $\mathcal{O}(Q_B)$.

Tipo C:

En esta query queremos dado un nodo e_1, responder cuántos otros nodos se mantienen conectados a e_1 sin importar qué arista sea removida (sólamente una arista).

Es claro que los nodos alcanzables desde e_1 con un camino simple que contenga puentes no pueden estar incluidos en la respuesta, dado que si cortamos alguno de los puentes en dicho camino el nodo queda desconectado (por lo que vimos con las queries de Tipo A, el conjunto de puentes en un camino simple entre 2 nodos es único).

Por otro lado, todos los nodos alcanzables desde e_1 con un camino simple que no utilicen ningún puente estarán incluidos en la respuesta. Al ser todas las aristas no puentes, están cada una contenida en algún ciclo simple, removiendo cualquiera de ellas, podemos reconstruir otro camino que los mantenga conectados usando el ciclo.

Esto último es equivalente a **considerar las componentes conexas del grafo resultante de quitar todos los puentes**. Los nodos en cada componente son alcanzables entre sí cumpliendo la condición de la query, y dos nodos de distintas componentes no cumplen la condición por lo visto antes.

Tenemos entonces una equivalencia entre los nodos que queremos contar y su alcanzabilidad en el grafo. Para responder las queries, podemos tener precomputadas las componentes conexas del grafo sin puentes. Sabiendo para cada nodo a qué componente de estas pertenece y el tamaño de las componentes, podemos responder cada query en $\mathcal{O}(1)$. La respuesta será el tamaño de dicha componente restado 1, porque debemos descontar al propio nodo por el cual estamos preguntando. Luego Q_C queries de tipo B tienen complejidad $\mathcal{O}(Q_C)$.

3.3. Algoritmo

Describimos a continuación el algoritmo completo en forma de pseudocódigo que resuelve el problema en base a lo presentado en el análisis.

```
dfsPuentes(0,0,0);
for cada nodo no visitado por dfsComponentes do
    componente ← {};
    dfsComponentes(nodo, componente);
    for cada nodo de componente do
        tamaño[nodo] ← componente;
    end
end
for cada query do
    if tipo A then
        c ← bfsCaminoSimple(u,v);
        return cantidad de puentes en c;
    end
    if tipo B then
        return si la arista es puente o no;
    end
    if tipo C then
        return tamaño[nodo]-1;
    end
end
```

Algorithm 1: Algoritmo general

El siguiente es el pseudocódigo de `dfsPuentes`, algoritmo que se encarga de calcular los puentes del grafo.

```
Data: v, d, padre
profundidad[v] = low[v] = d;
for cada vecino w de v distinto del padre do
    if profundidad[w] == -1 then
        dfsPuentes(w,d+1,v);
        low[v] = min(low[v], low[w]);
        if low[w] >= profundidad[w] then
            marcar (v,w) como puente;
        end
    else
        low[v] = min(low[v], low[w]);
    end
end
```

Algorithm 2: `dfsPuentes`

Por último mostramos el pseudocódigo de `dfsComponentes`, para calcular las componentes conexas del grafo sin puentes. El algoritmo es una variación de un DFS, con la única diferencia de evitamos visitar un nodo si llegamos a el por un arista puente.

```
Data: v, componente
visitar v;
agregar v a componente;
for cada vecino w de v no visitado do
    if la arista (v,w) no es puente then
        dfsComponentes(w, componente);
    end
end
```

Algorithm 3: `dfsComponentes`

Análisis de complejidad

En base a lo visto en clase, la complejidad de `dfsPuentes` (o el DFS modificado para calcular los puentes de un grafo), tiene la misma complejidad que un DFS, que es $\mathcal{O}(N+M)$, que en nuestro contexto es $\mathcal{O}(M)$. La complejidad de hacer los `dfsComponentes` es también $\mathcal{O}(N+M) = \mathcal{O}(M)$, porque cada nodo se visita una vez, al igual que en un DFS clásico. Por tanto, todo el precómputo tiene complejidad $\mathcal{O}(M)$.

Luego por lo que vimos antes, responder Q_A queries de tipo A podemos hacerlo en complejidad $\mathcal{O}(MQ_A)$, Q_B queries de tipo B en $\mathcal{O}(Q_B)$, y Q_C queries de tipo C en $\mathcal{O}(Q_C)$.

Finalmente, la complejidad total de nuestro algoritmo es $\mathcal{O}(M + MQ_A + Q_B + Q_C)$.

3.4. Casos de prueba

Caso de prueba	**Archivo de entrada**	**Salida esperada**
El grafo es un árbol, por tanto todas las aristas son puentes.	6 5 1 2 3 1 5 2 2 4 4 6 6 A 3 6 A 5 6 B 1 B 5 C 1 C 3	4 3 1 1 0 0
El grafo es un ciclo simple, por tanto no tiene ningún puente.	5 5 1 2 2 3 3 4 4 5 5 1 3 A 1 5 B 1 C 1	0 0 4
El grafo es una serie de ciclos simples conectados por aristas. Entre los ciclos simples no se forman otros ciclos. Las aristas de los ciclos no son puentes, y las que conectan los ciclos sí.	11 13 1 2 1 3 2 3 3 4 4 5 4 6 5 7 6 7 6 8 8 9 8 11 9 10 10 11 9 A 10 8 A 1 7 A 2 9 B 1 B 4 B 9 C 1 C 6 C 10	0 1 2 0 1 1 2 3 3

3.5. Código de la solución

```cpp
#include <iostream>
#include <vector>
#include <queue>

using namespace std;

struct Edge {
    int node, id;
    Edge() { node = -1, id = -1; }
    Edge(int n, int i) { node = n, id = i; }
};

typedef vector<int> vi;
typedef vector<bool> vb;
typedef vector<Edge> vEdge;

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

vector<vEdge> graph;
vi depth;
vi low;
vb visit;
vb bridges;
vi CCSizes;

// Hace el DFS calculando los puentes.
// Ademas al encontrar un puente, genera la nueva componente conexa
// delimitada por el puente.
void dfsBridges(int v, int d, int p){
    depth[v] = low[v] = d;
    for(auto adj : graph[v]){
        int w = adj.node;
        if(w != p){
            if(depth[w] == -1){
                dfsBridges(w,d+1,v);
                low[v] = min(low[v], low[w]);
                // Puente
                if(low[w] >= depth[w]) bridges[adj.id] = true;
            }else{
                low[v] = min(low[v], depth[w]);
            }
        }
    }
}

// Hace DFS ignorando pasar por aristas que sean puentes, y va llevando
// la cuente de la componente.
void dfsComponents(int v, vi &nodes){
    visit[v] = true;
    nodes.pb(v);
    for(auto adj : graph[v]){
        int w = adj.node;
        int edge = adj.id;
        if(!bridges[edge] && !visit[w]) dfsComponents(w, nodes);
```

```
    }
}

int main(){
    int N, M; cin >> N >> M;
    graph = vector<vEdge>(N, vEdge());
    forn(i,M){
        int u, v; cin >> u >> v; u--; v--;
        graph[u].pb(Edge(v,i));
        graph[v].pb(Edge(u,i));
    }

    // Preproceso
    depth = vi(N, -1);
    low = vi(N, -1);
  visit = vb(N, false);
    bridges = vb(M, false);
    CCSizes = vi(N, 0);

    // DFS para calcular los puentes
    dfsBridges(0,0,0);
    // DFS para calcular las componentes conexas sin puentes
    forn(i,N) if(!visit[i]){
        vi nodes;
        dfsComponents(i,nodes);
        for(auto v : nodes) CCSizes[v] = nodes.size();
    }

    // Queries
    int QS; cin >> QS;
    forn(i,QS){
        char qi; cin >> qi;
        if(qi == 'A'){
            int u, v; cin >> u >> v; u--; v--;
            // Lanzamos un BFS desde 'u' hasta 'v', reconstruimos el camino
            // y contamos la cantidad de puentes en el camino.
            vi predBFS(N, -1);
            vi predEdge(N, -1);
            queue<int> Q; Q.push(u);
            predBFS[u] = u;
            predEdge[u] = -1;
            while(!Q.empty()){
                int x = Q.front(); Q.pop();
        // Al llegar a 'v' cortamos el BFS.
                if(x == v) break;
                for(auto adj : graph[x]){
                    int y = adj.node;
                    if(predBFS[y] == -1){
                        predBFS[y] = x;
                        predEdge[y] = adj.id;
                        Q.push(y);
                    }
                }
            }
            int RTA = 0;
            int currNode = v;
            while(currNode != u){
```

```
                if(bridges[predEdge[currNode]]) RTA++;
                currNode = predBFS[currNode];
            }
            cout << RTA << endl;
        }else if(qi == 'B'){
            // Devolvemos 1 cuando la calle 'e' es un puente, y
            // 0 en el caso contrario.
            int e; cin >> e; e--; cout << bridges[e] << endl;
        }else if(qi == 'C'){
            // Devolvemos el tamanio de la componente conexa libre
            // de puentes que contiene a 'v' (menos 1, para no contar 'v').
            int v; cin >> v; v--; cout << CCSizes[v]-1 << endl;
        }
    }
    return 0;
}
```

4. Problema D: Desocupando del pabellón

Peso del ejercicio: 8

4.1. Descripción del problema

El problema puede modelarse de la siguiente forma si consideramos como vértices a las aulas y como aristas dirigidas a los pasillos. Dado un grafo dirigido de $\mathbf{A}$ vértices y $\mathbf{P}$ aristas, responder $\mathbf{Q}$ preguntas de la forma: "¿Existe un camino de v_1 a v_2 y otro de v_2 a v_1?"

4.2. Soluciones al problema

En lo que sigue llamaremos $\mathcal{A}$ (aulas) al conjunto de vértices y $\mathcal{P}$ (pasillos) al conjunto de aristas. Por enunciado tenemos que $\#\mathcal{A} = \mathbf{A}$ y que $\#\mathcal{P} = \mathbf{P}$

4.2.1. Enfoque naïve

Una vez obtenida la representación del grafo con listas de adyacencia, podríamos responder a cada query lanzando un algoritmo de recorrido de grafo ("BFS.º "DFS"por ejemplo) desde v_1 chequeando si alcanza a v_2, y otro desde v_2 chequeando si se alcanza a v_1.

De esa forma tendríamos una complejidad de $\mathcal{O}(\mathbf{A} + \mathbf{P})$ para leer la entrada y luego $\mathcal{O}(\mathbf{A} + \mathbf{P})$ para responder cada query. Obteniendo en total un algoritmo de complejidad $\mathcal{O}(\mathbf{A} + \mathbf{P} + \mathbf{Q} \cdot (\mathbf{A} + \mathbf{P}))$, que claramente no satisface la complejidad pedida por el enunciado.

4.2.2. Enfoque con componentes fuertemente conexas

Notemos primero que nada que no se realiza ningún cambio en el grafo entre las queries, eso nos da la pauta de que si realizamos un precómputo en $\mathcal{O}(\mathbf{A}+\mathbf{P})$ que nos permita resolver cada query en $\mathcal{O}(1)$, entonces tendríamos un algoritmo con complejidad $\mathcal{O}(\mathbf{A} + \mathbf{P} + \mathbf{Q})$ como pide el enunciado.

Estudiemos lo que se nos pide en las queries, para entender entonces qué precomputar (aunque el título de la sección ya es un spoiler). Dados dos vértices v_1 y v_2, queremos saber si existe un camino de v_1 a v_2 y viceversa, pero por lo visto en clase esto ocurre si y solo si v_1 y v_2 están en una misma componente fuertemente conexa del grafo.

Por ende, si calculamos en $\mathcal{O}(\mathbf{A} + \mathbf{P})$ a qué componente fuertemente conexa pertence cada nodo, y nos guardamos ese resultado en un arreglo llamado `scc` de tamaño $\mathbf{A}$, responder una query se reduce a responder si ocurre que $\texttt{scc}[v_2] = \texttt{scc}[v_2]$, lo cual se realiza en $\mathcal{O}(1)$ y cumple con lo estudiado en el primer párrafo.

4.3. Algoritmo y Análisis de Complejidad

1. Leemos $\mathbf{A}$ y $\mathbf{P}$. Complejidad $\mathcal{O}(1)$

2. Leemos las **P** aristas y las guardamos como lista de adyacencia del grafo original, y también del grafo transpuesto. Creamos `orden` como una cola de dos puntas (solo para evidenciar que podemos apilar al frente de la cola) vacía. Complejidad $\mathcal{O}(2 \cdot \mathbf{P}) + \mathcal{O}(1) = \mathcal{O}(\mathbf{P})$

3. Corremos DFS, y al finalizar la visita a un nodo, empujamos dicho nodo al frente de `orden`. Como este último paso agrega una operación $\mathcal{O}(1)$ por cada vértice, en total tenemos la misma complejidad de DFS. $\mathcal{O}(\mathbf{A} + \mathbf{P})$

4. Corremos DFS, pero en el orden dado por `orden` y en el grafo transpuesto. Además en el ciclo central del DFS vamos actualizando un contador que aumentamos cada vez que lanzamos una búsqueda nueva, y cada vez que terminamos la visita de un nodo, guardamos en `scc` a qué componente pertence el nodo. Nuevamente la complejidad de esto es $\mathcal{O}(\mathbf{A} + \mathbf{P})$, pue solo agregamos un contador y la actualización de un lugar en el arreglo, que se hacen a lo sumo una vez por nodo.

5. Leemos **Q**. Complejidad $\mathcal{O}(1)$

6. Leemos las **Q** queries y en cada query, leemos a v_1 y v_2. Si $\texttt{scc}[v_1] = \texttt{scc}[v_2]$ imprimimos "S", si no, imprimimos "N". Complejidad $\mathcal{O}(\mathbf{Q})$, pues se responde en $\mathcal{O}(1)$ cada query.

- **Observación** : Notar que los puntos 3 y 4 corresponden al Algoritmo de Kosaraju visto en clase.

Nos queda una complejidad de $\mathcal{O}(\underbrace{\mathbf{P}}_{1 \text{ y } 2} + \underbrace{(\mathbf{A} + \mathbf{P})}_{3} + \underbrace{(\mathbf{A} + \mathbf{P})}_{4} + \underbrace{\mathbf{Q})}_{5 \text{ y } 6}$. Lo cual nos da como complejidad final $\mathcal{O}(\mathbf{A} + \mathbf{P} + \mathbf{Q})$, que cumple con la complejidad pedida en el trabajo práctico.

4.4. Código de la solución

```
#include <iostream>
#include <cassert>
#include <vector>
#include <deque>

using namespace std;

typedef long long tint;

#define forn(i,n) for(tint i=0;i<(tint)(n); i++)

void dfsVisita (deque<tint> &ordenDfs, vector<bool> &visitado,
        vector<vector<tint> > &listaVecinos, tint nodo,
        tint k, vector<tint> &scc)
{
  // ACA COMIENZA LA VISITA
  visitado[nodo] = true;
  for (auto vecino : listaVecinos[nodo])
    if (!visitado[vecino])
      dfsVisita(ordenDfs, visitado, listaVecinos, vecino, k, scc);
  // ACA FINALIZA LA VISITA
  scc[nodo] = k;
  ordenDfs.push_front(nodo); /* Notar que lo ponemos al principio para
                 * que quede directamente ordenado por
                 * orden decreciente de finalizacion */
}

deque<tint> dfs (vector<bool> &visitado, deque<tint> &orden,
        vector<vector<tint> > &listaVecinos, vector<tint> &scc)
{
  tint k = 0;
  deque<tint> ordenDfs;
  for(auto nodo : orden)
    if (!visitado[nodo])
      dfsVisita(ordenDfs, visitado, listaVecinos, nodo, k++, scc);
  return ordenDfs;
}

int main()
{
  #ifdef ACMTUYO
    assert(freopen("ej4.in", "r", stdin));
  #endif
  ios_base::sync_with_stdio(0);
  cin.tie(NULL);

  tint a,p;
  while (cin >> a >> p)
  {

    /* Leemos la entrada, y guardamos la lista de adyacencia del
     * grafo original y su transpuesto (vamos a usar Kosaraju) */
    vector<vector<tint> > listaVecinosOriginal (a);
    vector<vector<tint> > listaVecinosTranspuesto (a);


```

```
    forn(i,p)
    {
      tint u,v;
      cin >> u >> v;
      // Las aulas vienen numeradas desde 1, de ahi el "-1"
      listaVecinosOriginal[u-1].push_back(v-1);
      listaVecinosTranspuesto[v-1].push_back(u-1);
    }
    // La funcion dfs, toma un orden en el que genera el dfs-forest
    deque<tint> ordenUsual (a);
    forn(i,a)
      ordenUsual[i] = i;

    // Indica para cada nodo si fue visitado
    vector<bool> visitado (a,false);

    // Guarda la componente fuertemente conexa de cada nodo
    vector<tint> scc (a);

    // La primera vez no importa que queda en scc
    deque<tint> ordenDfs = dfs(visitado, ordenUsual, listaVecinosOriginal,scc);

    // Reiniciamos el vector de visitados para el nuevo dfs
    forn(i,a)
      visitado[i] = false;

    /* Corremos dfs en el grafo transpuesto, mirando los nodos en
     * el orden obtenido y ahora si llenamos el vector scc */

    ordenDfs = dfs(visitado,ordenDfs,listaVecinosTranspuesto,scc);


    // RESPONDEMOS LAS QUERIES

      /* Para cada query se quiere saber si las 2 aulas estan en
       * una misma componente fuertemente conexa */
    tint q;
    cin >> q;
    forn(i,q)
    {
      tint a1,a2;
      cin >> a1 >> a2;
      // Otra vez, restamos uno por la numeracion
      if (scc[a1-1] == scc[a2-1])
        cout << "S\n";
      else
        cout << "N\n";
    }
  }
  return 0;
}
```

5. Bibliografía

Referencias

[1] Thomas H. Cormen et. al
Introduction to Algorithms
Third Edition
MIT Press
2009
ISBN 978 0 262 03384 8