Trabajo Práctico N^{o}2

Ada Lovelace

07/10/16 Problemas, Algoritmos y Programación

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

Índice

1. Problema A:

1.1. Introducción:

El problema consiste en, dadas N esquinas y M calles bidireccionales que conectan pares de esas esquinas, hallar la mínima cantidad de esquinas en las que podemos poner a un estudiante del departamento a contar sobre la carrera de Ciencias de la Computación, de manera que todo alumno que se dirija a su escuela se cruce con al menos uno de estos estudiantes. En cada esquina puede haber una escuela, un alumno o estar vacía. Sabemos que para cada esquina existe un camino entre ella y el resto, pero no sabemos a qué colegio va cada chico.

A continuación desarrollamos la idea que sigue el algoritmo que construimos, basado en grafos y redes de flujo.

1.2. Desarrollo:

Redujimos el problema propuesto a un problema de grafos. A partir de las calles y esquinas dadas como entrada construimos un grafo dirigido de la siguiente manera. Por cada esquina utilizamos dos nodos; a uno de ellos lo identificamos como *in* y al otro como *out*. Además usamos otros dos nodos extra a los que llamamos s y t, que en el contexto de redes de flujo representan una fuente y un sumidero respectivamente. Entonces, utilizamos un total de $2{*}N{+}2$ nodos. Luego, agregamos aristas de la siguiente manera:

- Para cada par de nodos (x_{in}, x_{out}) asociados a la misma esquina agregamos una arista en el sentido $\mathrm{x}_{in} \longrightarrow \mathrm{x}_{out}$.
- Para cada par de esquinas i y j que están conectadas por una calle agregamos las aristas $i_{out} \longrightarrow j_{in}$ y $j_{out} \longrightarrow i_{in}$.
- Para cada esquina a que contiene un alumno agregamos la arista $s \longrightarrow a_{in}$.
- Para cada esquina e que contiene una escuela agregamos la arista $e_{out} \longrightarrow t$.

Veamos un ejemplo que muestre más gráficamente lo que hacemos:

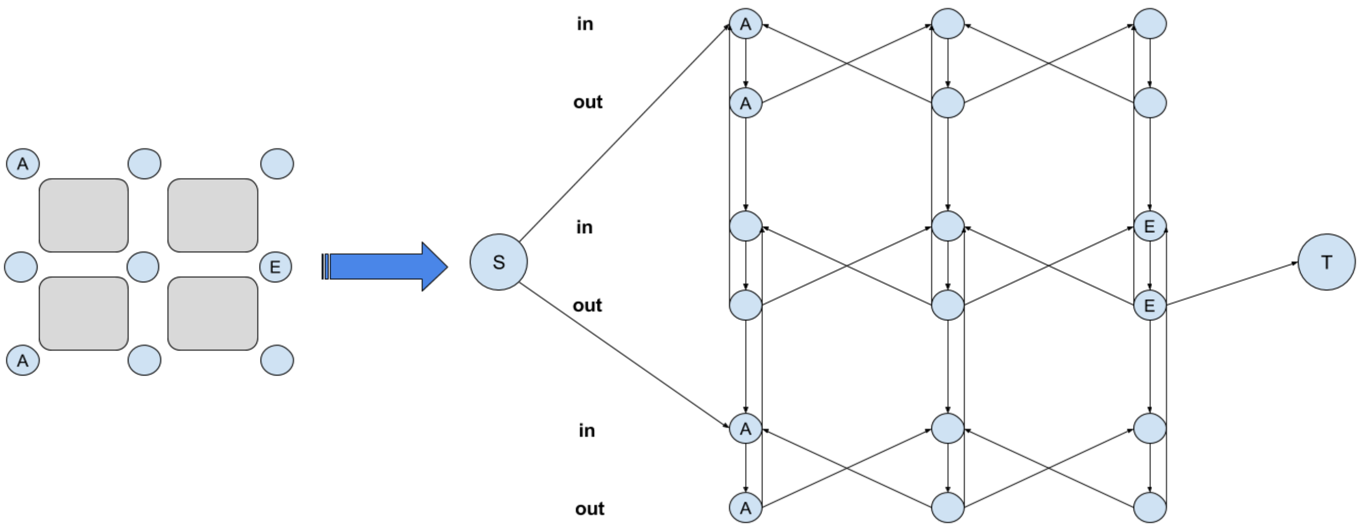

Figura 1: Representacion del problema utilizando grafos dirigidos.

De esta manera, ya esta listo el grafo dirigido que nos permite calcular la solución buscada. Para trabajar con el concepto de redes de flujo nos faltaría asignarle una capacidad a cada arista del grafo. En este caso, le asignamos a todas las aristas capacidad 1.

Luego, hallar la solución a nuestro problema es equivalente a hallar el flujo máximo dentro de la red construida (justificado en la sección de correctitud). Para obtener el flujo máximo utilizamos el método de Ford Fulkerson, que consiste en iniciar con una red de flujo vacía y en cada iteración buscar un camino de aumento en la red residual para incrementar el valor del flujo. Una vez que no existen caminos de aumento, el flujo es máximo.

Para representar el grafo utilizamos una implementación basada en listas de adyacencia, y para las capacidades y el flujo usamos matrices donde la posición $(i,\ j)$ contiene la capacidad o el flujo asociado a la arista $i \longrightarrow j$. Si esa arista no existe entonces el valor es 0 en ambos casos.

El algoritmo que construimos para calcular el flujo máximo recibe como parámetros el grafo, los nodos que representan a s y t, y la matriz de capacidades. Dividimos la explicación en varios items para que sea más clara:

- En primer lugar, en base al grafo recibido construimos la red residual. Para ello iteramos a través de cada nodo del grafo, y para cada uno de sus vecinos agregamos la arista inversa, es decir, la arista que va desde el vecino hacia el nodo en que estamos. Por la forma en que construimos el grafo previamente sabemos que nunca van a existir aristas inversas, por lo tanto no es necesario verificar si ya existen.
- Construimos un flujo vacío (matriz llena de ceros) para comenzar a obtener los caminos de aumento.
- Iniciamos un ciclo que en cada iteración busca un nuevo camino de aumento. Este finaliza una vez que no hayan nuevos caminos de aumento. Para encontrar los caminos, implementamos un algoritmo basado en BFS, que a la hora de encolar vecinos antes verifica si el flujo de la arista (que va del nodo hacia el vecino) es menor que su capacidad. Si es así lo encola, y se guarda en un arreglo quién es el padre de ese vecino. Caso contrario sigue con los otros vecinos, sin hacer nada. Luego, en base al arreglo de "padres" se reconstruye el camino y se devuelve como resultado de la función.
- A medida que vamos obteniendo los caminos de aumento vamos actualizando la función de flujo.
- Finalmente, cuando ya no hay camino de aumento, el resultado buscado es la cantidad de flujo que ingresa a t en la red final.

1.2.1. Pseudocódigo:

```
Sea Q el conjunto de esquinas y C el conjunto de calles
    // construimos grafo dirigido g = (V,E)
    g <-- grafo(2*N+2)
    para cada nodo n de los nodos 'in' de V
        g.agregarArista(n_in, n_out)
    para cada calle de C que une esquinas a y b
        g.agregarArista(a_out, b_in)
        g.agregarArista(b_out, a_in)
    para cada esquina e de Q que tiene un alumno
        g.agregarArista(s, e_in)
    para cada esquina e de Q que tiene una escuela
        g.agregarArista(e_out, t)

    // armamos la matriz de capacidades
    capacidades[2*N+2][2*N+2] <-- matriz con 0's
    para cada arista (a,b) de E
        capacidades[a][b] <-- 1

    resultado <-- flujoMaximo(g, s, t, capacidades)
    devolver resultado
```

```
flujoMaximo(g, s, t, capacidades)
    // construimos red residual
    para cada arista (i,j) de E
        g.agregarArista(j, i)

    // construimos flujo vacío
    flujo[g.size][g.size] <-- matriz con 0's

    // buscamos caminos de aumento y actualizamos el flujo por cada camino
    mientras (true)
        caminoAumento <-- bfs(g, s, t, capacidades, flujo)
        si (caminoAumento == vacio)
            salir del ciclo

        // actualizamos la función de flujo
        para i entre 0 y caminoAumento.size-1
            a = caminoAumento[i]
            b = caminoAumento[i+1]
            flujo[a][b] <-- flujo[a][b] + 1
            flujo[b][a] <-- flujo[b][a] - 1

    // el resultado es la cantidad de flujo que entra a t
    res <-- 0
    para i entre 0 y g.size-1
        res <-- res + flujo[i][t]

    devolver res
```

```
bfs(g, nodoInical, nodoFinal, capacidad, flujo)
    visitados[g.size] <-- arreglo lleno de 'false'
    // aca me guardo el padre de cada nodo visitado, es decir, de que nodo vengo
    padre[g.size] <-- arreglo lleno de -1
    // inicializamos cola vacia
    cola <-- {}
    padre[nodoInical] <-- nodoInical
    visitados[nodoInical] <-- true
    cola.push(nodoInical)
    mientras(!cola.empty)
        nodo <-- cola.front
        cola.pop
        si (nodo == nodoFinal){ // si llegue a nodoFinal termino
         salir del ciclo
        vecinos = g.nodosAdyacentes(nodo)
        para cada nodo v de vecinos
            // si no fue visitado, y la red residual me permite utilizar esa arista
            si (!visitados[v] && capacidad[nodo][v]-flujo[nodo][v] > 0)
                cola.push(v)
                padre[v] <-- nodo
                visitados[v] <-- true
```

```
    // inicializamos vector vacio
    camino <-- {}
    si (visitados[nodoFinal]) // si llegue al nodoFinal
        nodo <-- nodoFinal // reconstruyo el camino desde nodoInical a nodoFinal
        camino.push_back(nodo)
        mientras (nodo != padre[nodo])
            nodo <-- padre[nodo]
            camino.push_back(nodo)

    // invertimos el camino, porque sino queda al reves
    reverse(camino.begin,camino.end);
    devolver camino
```

1.2.2. Correctitud:

En esta sección explicaremos por qué el algoritmo que utilizamos resuelve el problema propuesto. Recordemos lo que nos pide el enunciado: la mínima cantidad de estudiantes de computación necesarios para que todos los alumnos se crucen con al menos uno de ellos durante su camino al colegio. Entonces, de alguna manera buscamos detectar los posibles caminos que puede tomar cada alumno, e ir "cortandolos" con estudiantes. Justamente eso es lo que hacemos con la red de flujo construida.

El flujo representa la cantidad de estudiantes que son necesarios. Cada camino de aumento que vamos encontrando a medida que recorremos la red residual representa un camino desde la casa de un alumno hasta alguna escuela, y tiene la particularidad de ser disjunto (en aristas y nodos) a los caminos de aumento previamente encontrados. Esto ocurre gracias a que utilizamos dos nodos por cada esquina, con una arista de capacidad 1 entre ellos, lo cual obliga a que ningún camino de aumento pase por una esquina que ya fue visitada.

Por otra parte, los caminos de aumento siempre aumentan en 1 el flujo. Nuevamente, esto se debe a la arista intermedia que ubicamos entre los dos nodos asociados a una misma esquina, de manera que por cada camino disjunto incrementemos en uno la cantidad de estudiantes. Si no usáramos esta técnica, el flujo no representaría lo que queremos, y no obtendríamos la solución al problema.

Por último, cuando no hay más caminos de aumento quiere decir que no existe ningún camino desde la casa de un alumno hasta alguna escuela, que pase por esquinas a partir de las cuales podemos no cruzarnos con algún estudiante, de manera que resolvemos el problema. Por esta razón es que buscamos el flujo máximo, lo cual es equivalente, a que no existan caminos de aumento en la red residual del grafo construido (propiedad).

1.2.3. Complejidad:

Para analizar la complejidad del algoritmo que utilizamos, vamos analizando cada una de sus partes y luego calculamos la complejidad total:

- Construcción del grafo: construimos un grafo de 2∗N+2 nodos con costo O(N). Agregamos N aristas, una por cada par de nodos (*in*, *out*), que toma O(N). Luego agregamos dos aristas por cada calle que conecta dos esquinas con costo O(M), y por último por cada escuela o alumno se agrega otra arista mas. En el peor de los casos todos las esquinas tienen escuela o alumno, por lo tanto tiene costo O(N).
  Costo total = O(N) + O(N) + O(M) + O(N) = O(N + M)
- Construimos la matriz que contiene la capacidad de cada arista (1 si existe la arista, 0 sino) con costo $O(N^2)$ al ser una matriz de tamaño (2∗N+2) x (2∗N+2).
  Costo total = $O(N^2)$

- Transformamos el grafo en su red residual, agregando un arista invertida, por cada arista que haya. Vimos en el primer ítem que el costo de agregar todas las aristas es O(N+M), por lo tanto como ahora agregamos la misma cantidad de aristas, pero invertidas, también toma O(N+M).
Costo total = O(N+M)

- Construimos flujo vacío (matriz llena de ceros) con costo O(N^2).
Costo total = O(N^2)

- Comenzamos a buscar caminos de aumento. Para hallar cada uno de ellos recorremos el grafo mediante BFS. Recordemos que el grafo tiene 2∗N+2 nodos y N+A+E+2M aristas, ya que agregamos N aristas, una por cada esquina; A aristas, una por cada esquina que contiene un alumno; E aristas, una por cada esquina que contiene una escuela; y 2M aristas, una por cada calle. Luego acotando tenemos:

  $$N + A + E + 2M \leq N + N + N + 2M = 3N + 2M \leq 3(M + 1) + 2M = 5M + 3 \in O(M)$$

  Observemos que acotamos N por M+1 ya que a partir de una esquina podemos llegar a cualquier otra, de manera que si lo vemos como un grafo donde las esquinas son nodos y las calles son aristas, son un grafo conexo (G conexo ⇒ M ≥ N-1 ⇒ M+1 ≥ N) . Luego, como la cantidad de caminos de aumento que podemos encontrar está acotada por N, la cantidad de BFS que vamos a hacer queda acotada por N. Entonces hacemos a lo sumo N BFS con costo O(M+N) cada uno.
Costo total = O(N∗(N+M)) = O(N^2+N∗M) ⊆ O(N∗M) (por ser conexo)

- Además, cada vez que encontramos un camino de aumento, debemos actualizar la función de flujo. Podemos considerar que este paso toma O(N) ya que nos basta con recorrer el camino de aumento para actualizar el flujo. Luego, como la cantidad de caminos de aumento es O(N), ya que cada camino de aumento implicaría agregar un estudiante y no vamos a agregar más de un estudiante por esquina, podemos considerar que este paso toma O(N^2).
Costo total = O(N^2)

Finalmente tenemos la siguiente suma (nuevamente utilizamos el hecho de que el grafo que representa a las esquinas y calles es conexo):

$$O(N + M) + O(N^2) + O(N * M) \subseteq O(N * M) \subseteq O(M^2)$$

De esta manera respetamos la complejidad requerida por el enunciado.

2. Problema B:

2.1. Introducción:

El problema que se desea resolver es el siguiente. Dados los valores de A acciones durante D días, se quieren graficar todos ellos (siendo un eje el tiempo, y el otro eje el valor, uniendo las mediciones por segmentos de recta). Si una acción se interseca con de otra, entonces no pueden estar en el mismo gráfico. Se desea buscar la mínima cantidad de gráficos necesarios para poder graficar todas las acciones. Se pide resolver el problema en $\mathbf{O}(A^2(A+D))$

2.2. Desarrollo:

Vamos a modelar este problema utilizando grafos, de la siguiente manera:

- Tendremos un nodo por cada acción.
- Si la accion i es "menor" a la accion j, entonces nuestro grafo tendra una arista dirigida $i \to j$. Una acción es menor a otra, si para todos los dias, el valor es menor.

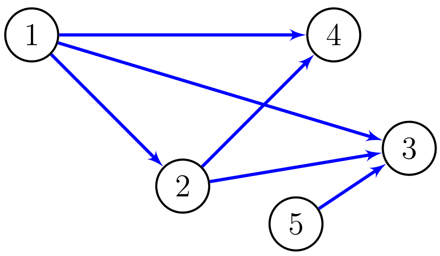

Figura 2: Representacion del ejemplo 2 del problema utilizando grafos dirigidos.

El grafo formado es un DAG, no tiene ciclos.

Si entre dos acciones no hay arista, quiere decir que ninguna es estrictamente menor a la otra, y por lo tanto no es posible meterlas en un mismo gráfico.

Entonces lo que estamos buscando es la máxima cantidad de acciones en las cuales no es posible establecer relación de orden entre ninguna de ellas. Esto es, el tamaño de la mayor anticadena.

Por el teorema de Dilworth, esto es equivalente a encontrar el tamaño del menor recubrimiento por caminos de nuestro grafo. Nuestro problema se reduce a encontrar el tamaño del menor recubrimiento por caminos de un DAG

2.2.1. Pseudocódigo:

```
bool esMenor(vector<int> a, vector<int> b){
    para i desde 0 hasta |a|:
        si a[i] >= b[i]:
            devolver falso
    devolver verdadero
}
```

```
int matchingBipartito(Grafo graph, int cantNodosA, int cantNodosB){
    int s <- cantNodosA+cantNodosB
    int t <- s+1
    graph.agregarNodo()  // agrego al grafo a s
    graph.agregarNodo()      // agrego al grafo a t
    para i desde 0 hasta cantNodosA:
        graph.agregarArista(s,i)

    para i desde 0 hasta cantNodosB:
        graph.agregarArista(cantNodosA+i,t)

    vector<int> ceros(graph.cantNodos(),0)
    vector< vector<int> > capacidad(graph.cantNodos(),ceros)
    para i desde 0 hasta graph.cantNodos():
        vector<int> vecinos = graph.nodosAdyacentes(i)
        para j desde 0 hasta |vecinos|:
            capacidad[i][vecinos[j]] <- 1

    int res <- flujoMaximo(graph,s,t,capacidad)
    devolver res
}
```

```
int cantidadDeGraficos(vector< vector<int> > acciones, int a, int d){
    Grafo graph(a)                // acciones.size() = a, acciones[i].size() = d
    para i desde 0 hasta a:
        para j desde 0 hasta a:
            si i != j:
                si esMenor(acciones[i],acciones[j]):
                    graph.agregarArista(i,j)


    Grafo graphBipartite(2*a)
    para i desde 0 hasta graph.cantNodos():
         vector<int> vecinos = graph.nodosAdyacentes(i)
         para j desde 0 hasta |vecinos|:
             graphBipartite.agregarArista(i,a+vecinos[j])


    int res <- a - matchingBipartito(graphBipartite,a,a)
    devolver res
}
```

2.2.2. Correctitud:

El problema de encontrar el tamaño del menor recubrimiento por caminos en un DAG, se puede resolver mediante matching bipartito. Un recubrimiento por caminos es valido si por cada nodo "entra" a lo sumo una arista y "sale" a lo sumo una arista.

Sea $G = (V,E)$ nuestro grafo original, armamos un grafo bipartito $G' = (V \cup V,E')$, donde por cada arista en E que une los nodos i y j, se agrega una arista en E' donde i pertenece a una particion y j a la otra.

Por cada eje del matching, lo que estamos haciendo es unir dos caminos. Si en un comienzo teniamos todos los nodos separados, o sea, n caminos (siendo n la cantidad de nodos) luego de encontrar un matching de longitud l tendremos $n - l$ caminos.

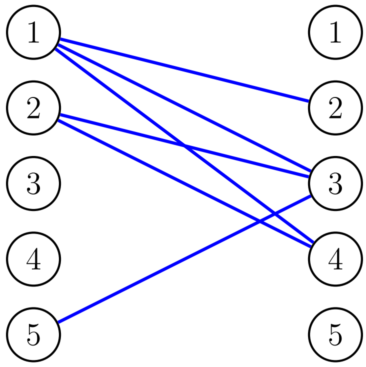

Figura 3: Grafo bipartito respectivo del ejemplo 2 del problema

Entonces el tamaño del menor recubrimiento es igual a $n -$ el matching máximo del grafo bipartito, donde n es la cantidad de nodos de G.

Para hallar el matching máximo del grafo bipartito, utilizamos flujo maximo. Agregamos un nodo s y un nodo t tal que, para todo nodo i de una particion A haya una arista dirigida $s \to i$, para todo nodo j de la otra particion B haya una arista dirigida $j \to t$ y que toda arista en el grafo original sea una arista dirigida de A hacia B. Cada arista tendrá capacidad 1. Entonces el matching máximo es equivalente a hallar el flujo maximo.

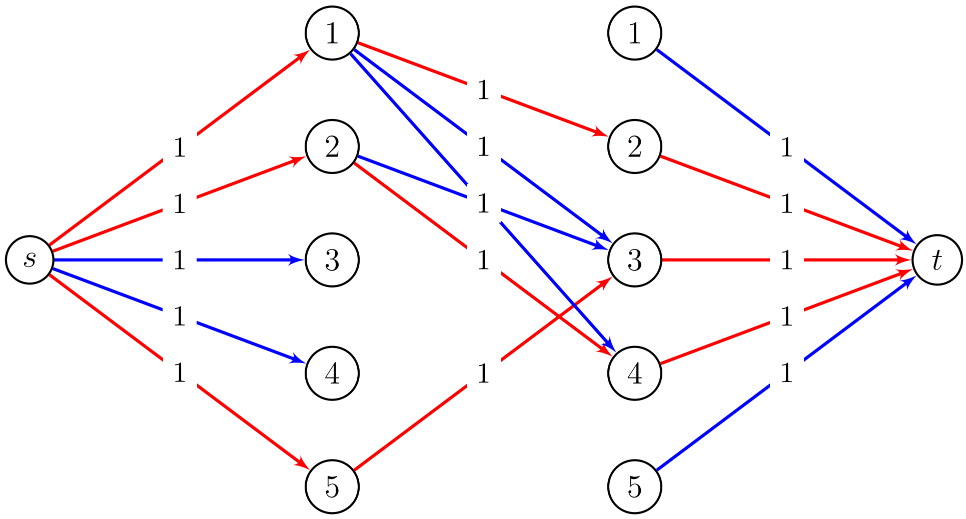

Figura 4: Flujo del ejemplo 2 del problema. Flujo máximo = 3

2.2.3. Complejidad:

- Comenzamos construyendo el grafo dirigido. Para cada acción, hay que verificar si es ”menor” a alguna otra acción. Esto tiene costo $\mathbf{O}(A^2D)$. El grafo resultante tiene A nodos y, en el peor caso, $\mathbf{O}(A^2)$ aristas
- Luego creamos el grafo bipartito para hallar el recubrimiento por caminos minimo, que va a tener $2A$ nodos y $\mathbf{O}(A^2)$ aristas. Esto tiene costo $\mathbf{O}(A^2)$.
- Agregamos 2 nodos más (s y t) y $2A$ aristas (que unen a s y t con sus respectivos nodos) en $\mathbf{O}(A)$.
- Finalmente, buscamos el flujo máximo. Esto tiene costo $\mathbf{O}(mn)$ siendo m la cantidad de aristas y n la cantidad de nodos (con el método de Ford–Fulkerson, pues cada camino de aumento se puede encontrar en $\mathbf{O}(m)$ y el flujo máximo es n). Entonces tenemos un costo $\mathbf{O}(A^3)$

Entonces la complejidad total es $\mathbf{O}(A^2D + A^3) = \mathbf{O}(A^2(A + D))$

3. Problema C:

3.1. Introducción:

Se tiene un mapa con N esquinas y M calles bidireccionales que conectan pares de esquinas en una ciudad. Se sabe que para todo par de esquinas existe un camino en el mapa utilizando algunas de las M calles. Se quiere responder Q preguntas en base a la información que el mapa dispone. Hay tres tipos de preguntas:

A) Dadas e_i, e_j dos esquinas del mapa, se pide saber la cantidad de calles c tales que al quitar c del mapa no exista un camino de e_i a e_j.

B) Dada c_i una calle del mapa, ¿Existen e_i, e_j dos esquinas del mapa tales que no hay camino de una a la otra si quitamos esa calle?

C) Dada e_i una esquina del mapa, se quiere saber a cuantas otras esquinas del mapa se puede llegar quitando una calle cualquiera del mapa.

3.2. Desarrollo:

Se decidió modelar el problema con un grafo simple G donde los vértices son las esquinas del mapa y los ejes sus calles.

Se sabe que para todo par de esquinas existe un camino en el mapa de una a la otra, entonces para todo par de vértices existe un camino en el grafo, entonces el grafo es conexo.

Observación 1. Sea $G = (X, V)$ un grafo conexo $\Rightarrow n - 1 \leq m$, con $|V| = n$ y $|X| = m \Rightarrow n$ es *O(m)*.

Observación 2. Sea $G = (X, V)$ un grafo simple (no dirigido)

- Un eje $e \in X$ es puente $\Leftrightarrow G - x$ tiene más componentes conexas que G.
- Dos vértices $v, v' \in V$ están en una misma componente biconexa $\Leftrightarrow$ pertenecen a un mismo ciclo.

Proposición 1. *Sean $G = (X, V)$ un grafo simple (no dirigido) y $v_1, v_2 \in V$ dos vértices de dos componentes biconexas distintas Rightarrow $\nexists$ C, C' caminos de v_1 a v_2 disjuntos en ejes.*

De existir, C $\cup$ invertirOrden(C') es un ciclo al cual v_1 y v_2 pertenecen $\Rightarrow$ v_1 y v_2 son vértices de una misma componente biconexa (por observación 2), lo cual es un absurdo.

Una vez formado el grafo, se utilizó *DFS* de la forma vista en la clase 5 para hallar componentes biconexas, puntos de articulación y puentes en G (visto en clase). Lo cual es información suficiente para responder todas las queries de tipo **B** y **C** (ver sección *Correctitud*, página 14).

Para responder las queries de tipo **C** se planteó contar la cantidad de vértices en los subgrafos existentes en $G - P$, con P el conjunto de ejes puente en G.

Para responder las queries de tipo **A** se precisó calcular la cantidad de puentes existentes en un camino simple entre los vértices en cuestión. Conociendo los puentes en G se puede realizar un *BFS* desde alguno de los dos vértices y verificar al llegar al otro cuantos se cruzaron. Para esto se persiste durante el *BFS* en cada vértice la cantidad de puentes atravesados para llegar hasta ese punto.

3.2.1. Pseudocódigo:

Sea $|V| = N$, $|X| = M$. El grafo se representó con listas de adyacencias.

Para poder responder queries de tipo **B** en $\mathbf{O}(1)$ se tomó un arreglo de tamaño M donde el *i-ésimo* elemento representa la calle c_i.

Reducir el problema a grafos es leer las ejes y agregarlas tanto al grafo como al arreglo de calles, lo que concluye en una complejidad temporal de $\mathbf{O}(N + M)$.

Sobre el grafo se implementó un *DFS* que completa más información sobre el *DFS Tree* generado a partir de cada nodo.

```
depth[N], low[N], parent[N] inicializados en -1

BCCRecursivo( vertice, profundidad, pila calles_visitadas)

depth[v] <- low[v] <- profundidad
c_hijos <- 0

Para i : 0 -> N
    w <- vecinoDe(v)
    Si depth[w] == -1 entonces hacer
        c_hijos <- c_hijos+1
        parent[w] <- v;

        calles_visitadas.Agregar( Calle (v, w) )

        BCCRecursivo( w, profundidad+1, calles_visitadas)

        low[v] <- minimo( low[v], low[h] )

        Si ( depth[v] == 0 && c_hijos > 1 ) || ( depth[v] > 1 && low[h] >= depth[v] )
        Hacer:
            punto_articulacion[v] <- true
            desenconlar calles_visitadas hasta Calle(v,h)
        Fin hacer
    Fin hacer
    Si no
        Si w != parent[v] && depth[w] < low[v] Hacer:
            low[v] <- min( low[v], depth[w] )
            calles_visitadas.Agregar( Calle (v, w) )
        Fin Hacer
    Fin Si no
Fin Para
```

De aplicar este algoritmo se pueden obtener los puentes, escencial para resolver las diferentes consultas.

Para responder queries de tipo **A** se implementó un *BFS* que persiste más información en los vértices, puntualmente la cantidad de puentes que se atravesaron desde el vértice inicial.

```
Cola.Encolar (vertice inicial)
Marcar visitado (vértice inicial)
Puentes_Cruzados_Hasta[vértice inicial] <- 0

Mientras  !Cola.Vacia()
    v = Cola.Popear()
    Cola.Desencolar()

    Para Todo Vecino w en VecinosDe(v)
        Si ! FueVisitado(w) Hacer
            MarcarVisitado(w)
            Puentes_Cruzados_Hasta[w] = Puentes_Cruzados_Hasta[v] + esPuente(v,w)
            Cola.Encolar(w)
        Fin Hacer
    Fin Para Todo
Fin Mientras
```

Donde *esPuente* chequea que un vértice sea el padre del otro (dado el orden en el que se recorrió en el *DFS*) y que el hijo tenga un valor *low* mayor o igual a su *depth*, lo que significa que en el subárbol que comienza en el hijo, no hay *back edges* que permitan cerrar un ciclo con sus ancestros.

Ambos algoritmos tienen una complejidad temporal $\mathbf{O}(N + M)$.

Para poder responder la *query C* se copio el grafo existente sin incluir los puentes del grafo, y luego se corrió *BFS* desde cada una de las componentes conexas resultantes.

Todas estas operaciones tienen también una complejidad temporal $\mathbf{O}(N + M)$.

Dada la observación 1, se concluye en una complejidad temporal de $\mathbf{O}(N + M)$.

3.2.2. Correctitud:

Sea $G = (X, V)$ el grafo simple con el cual modelamos el mapa con V sus vértices y X sus ejes. G es conexo.

Query B

Sea $e \in X$ una calle a quitar del mapa.

$\exists\, v, v' \in V$ tal que no hay camino entre ellos en $G - e \Leftrightarrow G - e$ no es conexo $\Leftrightarrow e$ es puente (G es conexo).

Por lo que basta saber si la calle en cuestión es o no un eje *puente* en G.

Query C

Sean $G_1, G_2, \ldots, G_k$ las componentes conexas (C.C.) del grafo $G' = G - P$, con $P \subseteq X$ el conjunto de ejes puentes en G, $\forall\, v, v' \in G_i$, $\exists$ C, C' dos caminos con diferentes ejes (por observación 2).

Sea $e \in X$ existe al menos uno de estos dos caminos en $G - e$ ya que no comparten ejes, por lo que $v \in G_i$ tiene al menos tantos otros posibles destinos como vértices en $G_i - v$.

Pero no pueden haber más ya que $\forall v' \in V - G_i = \cup_{j=1\,,j\neq i}^{k} G_j$ no hay dos caminos de v a v' disjuntos en ejes (propiedad 1) y están en componentes conexas distintas en G', por lo que $\exists\, e \in P$ tal que no hay camino de v a v' en $G - e$ y la condición de la consulta es que quitando cualquier eje exista uno.

Por lo que la consulta se reduce a dada una esquina, saber cuantos otras pertenecen a la misma componente conexa en el grafo $G - P$.

Query A

Sean $v, v' \in V$ dos esquinas del mapa.

Si v y v' pertenecen a la misma componente biconexa en G entonces la cantidad es 0 ya que están en un ciclo (por lo visto en la observación 2).

Si v y v' no pertenecen a la misma componente biconexa en G, $\exists$ C camino de v a v' (por ser G es conexo) por lo que se puede hacer *BFS* desde uno de los dos y se puede asegurar que se llegará al otro.

Proposición 2. *Sea $G = (V, X)$ un grafo simple conexo. $v, v' \in V$. $e \in X$. $\nexists$ C camino de v a v' en $G - e \Rightarrow e$ es puente.*

Por absurdo, $\exists\, e \in X$ tal que e no es puente y $\nexists$ C camino de v a v' en $G - e$. e no es puente $\Rightarrow e$ es un eje en B una componente biconexa de $G \Rightarrow$ es un eje en un ciclo. Pero para todo v_1, v_2 vértices en B, $\exists$ C' camino de v_1 a v_2 en $B - e$. Se toma v_1 y v_2 como los vértices a los cuales inciden los ejes puentes que se utilizaron para cruzar de una componente a otra en C. Suponiendo sin pérdida de generalidad que C comenzaba en v pasa primero por v_1 y luego por v_2 en $G \Rightarrow$ $\exists$ camino de v a v' en $G - e$, ya que se puede tomar $C_{v \to v_1} \cup C' \cup C_{v_2 \to v'}$, donde $C_{v \to v_1}$ y $C_{v_2 \to v'}$ son respectivamente los camino de v a v_1 y de v_2 a v', contenidos en $C \Rightarrow$ Absurdo!

Observación 3. Es trivial ver que si $\exists\, e \in X$ puente y $e \nexists$ C un camino de v a v' $\Rightarrow \notin$ C' camino de v a v' en $G - e$. En especial $C' = C$. Por lo que no interesan los puentes que no estén en el camino.

Proposición 3. *No hay dos caminos de v a v' que tengan diferentes puentes.*

$\exists$ C, C' caminos de v a v' tal que tienen al menos un puente diferente. Se puede colapsar cada componente biconexa en un solo vértice y así reescribir cada camino como una secuencia de componentes biconexas $< G_1, G_2, \ldots, G_l >$. Sea G_i como el primer elemento a partir del cual ambas secuencias dejan de ser iguales (al menos ambas contienen a G_1 componente biconexa de v). Sea G_j el primer elemento a partir del cual ambas secuencias son iguales (al menos ambas contienen a G_l componente biconexa de v'). $\Rightarrow$ hay dos caminos distintos entre las componentes biconexas G_i y G_j $\Rightarrow$ están en un ciclo $\Rightarrow$ son la misma componente biconexa $\Rightarrow$ ambos caminos pasan por las mismas componentes biconexas y mismos puentes. $\Rightarrow$ Absurdo!

Se deduce entonces que tomando cualquier camino entre dos vértices v, v' se atraviesan los mismos puentes (propiedad 3) y que al quitar uno ya no existirá camino entre ambos. La propiedad 2 implica que solo interesan ver los ejes que son puentes y se puede ver fácilmente que no importan aquellos no estén en el camino entre los vértices en cuestión (observación 3).

Por lo tanto si se persiste más información en los nodo al momento de correr el algoritmo *BFS*, como la cantidad de *puentes* que se cruzaron para llegar hasta el vértice actual, se tiene la respuesta.

3.2.3. Complejidad:

Sea el grafo $G = (X, V)$, con $|V| = N = cantidad\ de\ esquinas$, $|X| = M = cantidad\ de\ calles$.

Sean Q_A, Q_B y Q_C respectivamente la cantidad de queries a responder de cada tipo.

- Armar el grafo como una lista de adyacencias en base al *input* del problema tiene un costo temporal de $\mathbf{O}(N + M)$.
- Se aplica *DFS* con un costo temporal de $\mathbf{O}(N + M)$.
- Resolver una query de tipo B o C es $\mathbf{O}(1)$.
- Resolver una query de tipo A es el costo de aplicar *BFS* que es $\mathbf{O}(N + M)$.

Por lo que la complejidad total de la solución es de $\mathbf{O}(N + M + (N + M) \cdot Q_A + Q_B + Q_C)$.

Como N es $\mathbf{O}(M)$, se concluye en una compejidad de $\mathbf{O}(M + M \cdot Q_A + Q_B + Q_C)$.

4. Problema D: Desocupando el pabellón

4.1. Introducción:

Se presenta la siguiente situación: en una facultad en la que hay A aulas y P pasillos, luego se hacen Q preguntas del estilo: ¿se puede ir de un aula a_i a un aula a_j con $i \neq j$ y luego volver al aula a_i?. La complejidad del problema radica en que los pasillos de la facultad son unidireccionales, por ende el hecho de poder ir a un aula no implica que se pueda volver al aula desde la que se partió. Este problema debe ser resuelto dentro de la complejidad O(A + P + Q)

4.2. Desarrollo:

Para resolver el problema planteado anteriormente, se realizo una abstracción a un grafo dirigido. Dicha abstracción fue realizada de la siguiente manera:

- Las aulas pasan a ser los nodos del grafo.
- Los pasillos representan las aristas dirigidas del grafo.

Ejemplo:

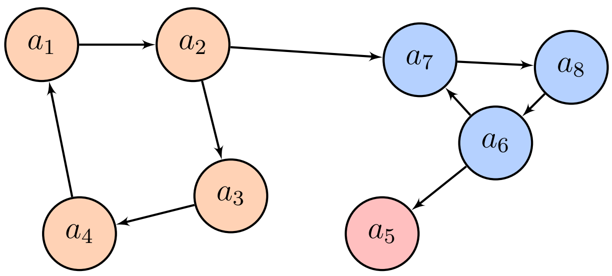

Figura 5: Representacion del problema utilizando grafos dirigidos.

De esta manera, se puede ir del aula a_i al aula a_j y volver $\Leftrightarrow$ existe un camino de a_i a a_j y de a_j a a_i $\Leftrightarrow$ a_i y a_j pertenecen a la misma componente fuertemente conexa.

Luego el problema se reduce a identificar las componentes fuertemente conexas del grafo y marcar a cual pertenece cada aula. En el ejemplo, cada color representa una componente fuertemente conexa distinta. Para identificar las componentes fuertemente conexas se utilizara el algoritmo de *Kosaraju* visto en clase.

4.2.1. Pseudocódigo:

```
array[int] kosaraju(grafo g)
    pila ordKosaraju <- ordenKosaraju(g)
    invertirAristas(g)
    compConex[int] <- componentesKosaraju(ordKosaraju)
    devolver compConex
```

```
pila ordenKosaraju(grafo g)
    pila res
    visitados[bool] <-(false*g.cantNodos)
    para (todos los nodos)
        si(no esta visitado)
            pila hijos <- nueva pila
            ordenKosaraju[int] <- nueva lista
            alcanzables[nodo] <- nueva lista
            dfsRecursion(i,hijos,visitados,alcanzables,ordenKosaraju)
            para (cada nodo en ordenKosaraju)
                res.agregarAtras(ordenKosaraju[j])
    devolver res
```

```
dfsRecursion(int nodo,pila hijos,visitados[bool],alcanzables[nodos],odenKosaraju[int])
    si(el nodo no esta visitado)
        alcanzables.agregarAtras(nodo)
        visitados[nodo] <- true
        para (cada vecino del nodo)
            hijos.apilar(vecinos del nodo)
        mientras(la pila hijos no este vacía)
            aux <- hijos.devolverYsacarTope
            dfsRecursion(aux,hijos,visitados,alcanzables,ordenKosaraju)
        ordenKosaraju.agregarAtras(nodo)
```

```
array[int] componentesKosaraju(pila ordenKosaraju)
    componentesConexasDeCadaNodo[int] <-(cantidadNodos*-1)
    componenteConexa <- -1
    visitados[bool] <- (cantidadNodos*false)
    mientras(ordenKosaraju no este vacío)
        nodoActual <- ordenKosaraju.devolverYsacarTope
        si(el nodoActual no esta visitado)
            componenteConexa <- componenteConexa+1
            alcanzables[nodos] <- dfs(nodo)
            para (todos los nodos alcanzables desde el nodoActual){
                si(no esta visitado el nodo)
                    visitados[alcanzables[i]] = true
                    componentesConexasDeCadaNodo[alcanzables[i]] = componenteConexa
    devolver componentesConexasDeCadaNodo
```

4.2.2. Correctitud:

Se aplica el algoritmo de $Kosaraju$ al grafo para identificar las componentes fuertemente conexas del mismo. La correctitud de dicho algoritmo ya fue demostrada en la clase numero 6.

Este algoritmo, en pocas palabras, corre dos dfs sobre un grafo dado. Con el primero busca obtener el finishing time de cada nodo, para de esta forma guardar un orden que va a ser utilizado en el segundo dfs. Se llamara a este orden, $ordenKosaraju$. Este primer dfs va marcando como visitados los nodos por los que ya paso, por lo que pasa una única vez por cada nodo.

Luego invierte las aristas, y por ultimo realiza el segundo dfs. Este dfs parte desde el primer nodo, k_1, en $ordenKosaraju$. Este segundo dfs marca los nodos a los que llega desde k_1 como visitados. Además el algoritmo asegura que todos estos nodos pertenecen a la misma componente fuertemente conexa y que todo nodo que pertenece a esta componente fue alcanzado por el dfs, osea no falta ninguno. Luego, continua por el segundo nodo en $ordenKosaraju$ que no haya sido visitado, k_k, y de nuevo, asegura que todos los nodos a los que se llega a partir de k_k pertenecen a la misma componente fuertemente conexa y que se alcanzo a todos los nodos que pertenecen a dicha componente. De esta manera, repitiendo este proceso con los nodos en $ordenKosaraju$ que todavía no haya sido visitados, se obtiene la información necesaria para saber a que componente fuertemente conexa pertenece cada nodo.

En el problema ,luego de obtener $ordenKosaraju$, y antes de ejecutar el segundo dfs al primer nodo de esta pila, se crea un vector al cual se referira con el nombre $componentesConexas$. Este vector tiene tamaño igual a la cantidad de nodos que tenga el modelo del problema, y tiene el siguiente propósito:

- Una posición del vector representa a un nodo.
- El contenido de dicha posición del vector representa la componente conexa a la que pertenece el nodo.

Entonces, al correr el dfs al primer nodo de $ordenKosaraju$, además de marcar como visitados los nodos a los que se llega con dicho dfs, se guarda en $componentesConexas$ que todos estos nodos pertenecen a la misma componente fuertemente conexa.

De esta manera, cuando el algoritmo de $Kosaraju$ termina, se tiene en $componentesConexas$, a que componente fuertemente conexa pertenece cada nodo. Esto permite que, dada una pregunta q_i, que pregunta si se puede ir y volver del aula a_i al aula a_j, simplemente se consulta en $componentesConexas$ que el contenido de las posiciones a_i y a_j sea el mismo. Si el contenido es igual, las aulas pertenecen a la misma componente fuertemente conexa y entonces se procede a devolver ”S” y si difiere, se devuelve ”N”.

4.2.3. Complejidad:

- El algoritmo lo primero que hace es partir de un nodo arbitrario k y desde ese nodo hace un dfs. Marca todos los nodos que alcanza con dicho bfs como visitados. Además va pusheando en una pila, $ordenKosaraju$, los nodos a los que ya le termino de visitar todos sus hijos. De esta manera, la posición de los nodos en la pila, representa el finishing time de cada nodo. Por ejemplo, suponiendo que desde k se puede llegar a todos el resto de los nodos, entonces el dfs va a terminar de visitar todos los hijos de k cuando haya pasado por todo el resto de los nodos, esto implica que k va a tener el finishing time mas alto que el resto y por ende va a ser el primero en la pila.

  Como pasa una única vez por cada nodo, ya que marca como visitados los nodos por los que paso, la complejidad hasta aquí es de O(A+P).

  Resumen: O(A+P).

- Luego, invierte las aristas del grafo, para esto se pasa por cada arista y se cambia su sentido. Como el grafo esta implementado con una lista de adyacencias, la complejidad consiste exactamente en pasar una vez por cada arista.

  Resumen: O(A+P+P) = O(A+P).

- Por último, utilizando el orden calculado al principio, osea $ordenKosaraju$, se toma el primer nodo de la pila y a partir de ese nodo se lleva a cabo un dfs o bfs. Se marcan todos los nodos alcanzados por dicho dfs como visitados y además se los asocia con la misma componente conexa. Luego se continua con el siguiente nodo en la pila que no haya sido marcado como visitado en el anterior dfs y se repite el proceso. De nuevo, solo se visita una vez a cada nodo, por ende la complejidad de este paso también es de O(A+P).

  Resumen: O(A+P+A+P) = O(A+P).

- Con la información calculada anteriormente, por cada pregunta Q, se puede contestar en O(1) si dadas dos aulas, estas pertenecen a la misma componente conexa. Esto es así porque que se tiene un vector, donde cada posición representa un nodo y el contenido de esa posición, la componente conexa a la que pertenece dicho nodo. Por ende la complejidad final del problema termina siendo la siguiente:

  Resumen: O(A+P+Q).