Equipo 3

Proyecto Final EDA II

Nadia González Fernández Arrastre
Bryan Machín García C-212
José Alejandro Solís Fernández C-212

Facultad de Matemática y Computación
Universidad de La Habana

1. Problema 1

Sea una red de flujo $G =< V, E, c >$, con fuente y receptor $s, t \in V$ respectivamente y para toda arista $e \in E$ se cumple que $c(e) = 1$. Se desea eliminar k aristas de la red de flujo de tal forma que el valor del flujo máximo disminuya lo más posible. Diseñe un algoritmo que encuentre un conjunto F de k aristas ($|F| = k$) de tal forma que el valor del flujo máximo en la red de flujo $G' =< V, E - F, c >$ es lo más pequeño posible. La complejidad temporal de su algoritmo debe ser de $O(|E| * |f^*|)$.

1.1. Propuesta de Solución

Para seleccionar las k aristas de la red de flujo de forma que el valor del flujo máximo disminuya lo más posible, primero se deben escoger las aristas que cruzan el corte mínimo de la red de flujo. Si k es mayor que la cantidad de aristas que cruzan el corte mínimo entonces se selecciona cualquiera de las restantes de la red de flujo hasta completar la cantidad k de aristas.

1.2. Correctitud

El valor del flujo máximo es igual a la capacidad del corte mínimo de la red de flujo (visto en conferencia).
Como $c(e) = 1 \; \forall e \in G(E)$ podemos establecer una biyección entre la capacidad y la cantidad de aristas que cruzan un corte.
Para obtener un flujo máximo sobre la red de flujo G se utiliza el algoritmo **Ford-Furlkerson(G)**. El algoritmo genera una red residual G_f de G.
La solución consiste en realizar un $DFS\text{-}Visit$ a partir de s en G_f. Todos los vértices alcanzados por s, incluyendo s conformarán una parte del corte y los no alcanzables junto a t la otra parte del corte.
El corte sobre la red de flujo G con flujo máximo f^* es de capacidad mínima.
Demostración:

> Sea el corte (S, T) formado por $S = \{s, v_1, v_2, ..., v_k\}$ y $T = \{v_{k+1}, v_{k+2}, ..., v_{n-2}, t\}$ donde todo vértice $v_i \in S$ es alcanzable por s a través de un camino en la red residual G_f y todo vértice en T serían los no alcanzables por s bajo este criterio. Ambos conjuntos son no vacíos, pues a S al menos pertenece el vértice s y a T pertenece el vértice t, pues si no perteneciera significa que existe un camino de s hacia t en G_f lo que implica que sería un camino aumentativo y sería posible incrementar el flujo utilizando dicho camino lo cual entra en contradicción con el hecho de que f^* es un flujo máximo en G.
>
> Luego, toda arista $< u, v >$ que cruza el corte (S, T) de S hacia T, donde $u \in S$ y $v \in T$ cumple que $f(< u, v >) = c(< u, v >)$, pues si existiese $< u, v >$ que cruza el corte tal que $f(< u, v >) < c(< u, v >)$ entonces la arista $< u, v >$ no estaría saturada y existiera en G_f por lo que el camino de s hacia u de aristas no saturadas, adicionando la arista $< u, v >$ forman un camino s hacia v en G_f lo cual es una contradicción pues $v \in T$.
>
> Además toda arista $< v, u >$ que cruza el corte desde T hacia S donde $u \in S$ y $v \in T$ cumple que $f(< u, v >) = 0$, pues si existiese $< v, u >$ que cruza el corte tal que $f(< v, u >) > 0$ entonces dicha arista da origen a

una arista inversa $< u, v >$ en G_f, como $u \in S$, entonces existe un camino desde s hacia u en G_f que al unirse con la arista $< u, v >$ daría lugar a un camino de s hacia v en G_f lo cual es una contradicción con el hecho de que $v \in T$.

Por lo tanto todas las aristas que salen de S están saturadas y todas las que entran tienen flujo 0. Demostremos entonces que la capacidad del corte (S, T) es igual al valor del flujo máximo f^* en G. Sabemos que $|f^*| = f(S, T)$ por el *Lema* 26.4 visto en conferencia. Luego:

$$|f^*| = f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$$

Pero como demostramos anteriormente que $f(u, v) = c(u, v)$ y $f(v, u) = 0$ donde $u \in S$ y $v \in T$ entonces:

$$|f^*| = f(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{u \in S} \sum_{v \in T} 0 = \sum_{u \in S} \sum_{v \in T} c(u,v) = C(S,T)$$

Luego la capacidad del corte (S, T) es igual al valor del flujo máximo f^* en G por lo que su capacidad es mínima ya que $|f^*|$ es una cota mínima de la capacidad de todo corte por *Corolario* 26.5 visto en conferencia.

Sea C el conjunto de aristas que cruzan el corte mínimo encontrado en el $DFS\text{-}Visit$. Si $k \leq |C|$, se añaden al conjunto F k aristas de C. Si $k > |C|$ se añade a F todas las aristas de C y se completa con $k - |C|$ aristas restantes de G.

Si se quitan de una red de flujo todas las aristas de un corte mínimo, el flujo máximo es igual a cero, s y t quedan desconectados por lo que no pasa ningún flujo hacia t. Por tanto quitar otras aristas de la red residual no cambia el valor del flujo máximo.

1.3. Pseudocódigo

Algorithm 1

```
1: G_f ← Ford-Furlkerson(G,s,t)                    ▷ G_f es el grafo residual
2: visited ← DFS-Visit(G_f,s)
3: Min-Cut ← list
4: for each < a, b > in G.Edges do
5:     if visited[a] and not visited[b] then
6:         Min-Cut.add(¡a,b¿)
7:         ¡a,b¿.Added = True
8:     end if
9: end for
10: F ← list
11: while F.Count < k and Min-Cut.Count ≠ 0 do
12:     F.add(Min-Cut.Pop())
13: end while
14: while F.Count < k do
15:     edge = G.Edges.Pop()
16:     if not edge.Added then
17:         F.Add(edge)
18:     end if
19: end while
20: return F
```

1.4. Análisis de la complejidad temporal

La complejidad temporal del algoritmo **Ford-Furlkerson** es $O(|E| * |f^*|)$. La ejecución del **DFS-Visit** sobre una red de flujo es $O(|E|)$(demostrado en conferencia). Luego, se ejecutan tres ciclos independientes que recorren las aristas del grafo una única vez, por lo tanto cada uno es $O(|E|)$. Por regla de la suma, la complejidad temporal del algoritmo es $O(|E| * |f^*|)$.

2. Problema 2

Sean n ciudades, una de ellas la capital y m carreteras. Cada carretera conecta unidireccionalmente a un par de ciudades. Diseñe un algoritmo que devuelva una lista de carreteras a construir tal que luego de construidas sea posible llegar desde la capital al resto de las ciudades El tamaño de la lista tiene que ser lo menor posible incluso vacía en caso de que inicialmente todas las ciudades sean alcanzables desde la capital. La complejidad temporal del algoritmo debe ser $O(n + m)$.

2.1. Propuesta de Solución

Para hallar la menor cantidad de carreteras que deben ser construidas para llegar desde la capital al resto de las ciudades, modelemos el ejercicio como un grafo dirigido $G =< V, E >$. Sea V el conjunto de vértices formado por las n ciudades, incluida la capital, y E el conjunto de aristas, que representan las m

carreteras, donde la arista $< n_1, n_2 > \in E$ si y solo si existe una carretera de la ciudad n_1 hasta n_2.
Luego, se hallan todas las componentes fuertemente conexas y se crea un grafo reducido de dichas componentes fuertemente conexas.
Después se hace $DFS\text{-}Visit$ tomando como raíz el vértice que representa a la componente fuertemente conexa a la cual pertenece la capital.
Cada vértice tiene la propiedad *in-degree* (cantidad de aristas que inciden sobre él).
Luego, todos los vértices que no forman parte del árbol creado por $DFS\text{-}Visit$, es decir, todos las componentes fuertemente conexas que no son alcanzables desde la componente fuertemente conexa que contiene a la capital. Se analizan los *in-degree* y por cada vértice con *in-degree* $= 0$ se añade la arista formada por el vértice capital, y cualquier vértice de la componente fuertemente conexa donde en el grafo de componentes tiene *in-degree* $= 0$.

2.2. Correctitud

Sea $G^{SCC} = \{V^{SCC}, E^{SCC}\}$ el grafo reducido de G. Cada vértice tiene la propiedad *in-degree*. Esta se actualiza en el vértice w cuando se descubre la arista $< v, w >$. Esto no modifica el funcionamiento del algoritmo.
Sin pérdida de generalidad, sea C_i la componente fuertemente conexa que contiene al vértice de la capital. No es necesario añadir aristas dentro de C_i ya que existe un camino del vértice capital al resto de los vértices de la componente. Entonces se debe analizar si existe un camino que conecte un vértice de C_i con cualquier otro vértice de alguna componente C_j tal que $i \neq j$. Para ello, se hace $DFS\text{-}Visit$ en el grafo reducido G^{SCC}, tomando como raíz el vértice de la componente donde se encuentra la capital. Durante la ejecución, se actualiza una lista global que contiene todos los vértices visitados. Para cada vértice v visitado en este recorrido existe un camino desde la raíz (la capital) hasta v. Ningún vértice no visitado en el $DFS\text{-}Visit$ es alcanzable desde la capital.
Luego de este análisis, se hallan las aristas que deben ser añadidas tal que exista un camino desde la capital a todos los vértices.
Demostremos que solo es necesario añadir aquellas aristas formadas por el vértice capital y cualquier vértice dentro de una componente fuertemente conexa C representada por vértices no visitados tal que *in-degree*$(C) = 0$.

Lema 1: Para cada vértice v de *in-degree* > 0 existe un camino que conecta a un vértice u de *in-degree* $= 0$ a v.

Demostración: Sea $C = v_0, v_1, ..., v_k$ el camino más largo que contiene al vértice v_k. Supongamos que *in-degree*$(v_0) > 0$, entonces existe $< w, v_0 >$ tal que el vértice w cumple que:

- $w \notin c \Rightarrow \exists C'$ tal que C' tiene mayor longitud que C y contiene a v_k. *Contradicción.*
- $w \in C \Rightarrow$ existe un ciclo en C, ya que existe un camino de v_0 a w y la arista $< w, v_0 >$ completa el ciclo. *Contradicción* porque en un grafo reducido no existen ciclos.

Por lo tanto, añadiendo las aristas que van desde el vértice que contiene a la capital a los vétices que tiene *in-degree* $= 0$, la capital quedaría conectada con

todas las ciudades.
Demostremos que se han añadido la cantidad mínima de aristas necesarias. Sea $< u, v >$ una de las aristas añadidas. Por la forma en que se añadió esta arista, v tiene *in-degree* $= 1$. Al quitarla, v tendría *in-degree* $= 0$ y por lo antes expuesto queda desconectado de la capital.

2.3. Pseudocódigo:

Algorithm 2 Carreteras necesarias

```
1: G ← BuildGraph< V, E >
2: G' ←Strongly-Connected-Components(G)
3: G^SCC ← ReducedComponentGraph(G')                    ▷ grafo reducido
4: visited ← DFS-Visit(G^SCC, capital)
5: edges ← list
6: for each vertex in G^SCC do
7:     if not visited[vertex] and vertex.in-degree == 0 then
8:         edges.add(<capital,vertex>)
9:     end if
10: end for
11: return edges
```

2.4. Análisis de la complejidad temporal

Se construyen los grafos G, G' y G^{SCC} utilizando listas de adyacencia, lo cual tiene un costo $O(|V|+|E|)$. La complejidad temporal en la construcción del grafo G^{SCC} no se ve afectada por la actualización de la propiedad *in-degree*, ya que es $O(1)$. La ejecución del $DFS\text{-}Visit$ es $O(|V| + |E|)$ por la manera en que está implementado el grafo. En las líneas $6 - 9$ se recorre a lo sumo una vez cada vértice del grafo. Por lo tanto, el ciclo es $O(|V|)$. Luego, por regla de la suma, la complejidad temporal del algoritmo es $O(|V| + |E|)$.

3. Problema 3

Sea $G =< V, E >$ un grafo dirigido y ponderado, $s \in V(G)$, $t \in V(G)$, $P =< e_i, w_i >$ donde $e_i \in E(G)$, $w_i \in N$, y $0 \leq k \leq |P|$. Se desea calcular el camino de costo mínimo de s a t en G pudiendo cambiar a lo sumo k aristas del grafo G por aristas del conjunto P. La complejidad temporal de su algoritmo debe ser $O(k|E|log|V|)$.

3.1. Propuesta de Solución

Sean $G_1, G_2, \ldots, G_{(k+1)}$ grafos idénticos a G tal que $x = x_i$ donde $x \in G$ y $x_i \in G_i$ y G' el grafo formado por la unión de los anteriores. Cada arco de P será colocado entre todo par consecutivo de estos grafos, de manera que, si el arco $< a, b > \in P$ entonces $< a_i, b_{(i+1)} > \in G'$, para todo $1 \leq i \leq k$. Luego se calculan las distancias de los caminos de costo mínimo de s_1 hacia t_i para todo $1 \leq i \leq k + 1$ y, el menor de estos caminos, es la respuesta al problema.

3.2. Correctitud

Un camino que vaya desde el vértice s_1 hacia el vértice t_i , pasa exactamente por i arcos de P debido a que no existen arcos que vayan desde el grafo $G_{(i+1)}$ hacia el grafo G_i.
Sea j el índice del camino de costo mínimo de s_1 a t_i, para todo $1 \leq i \leq k+1$. Sea P dicho camino, que va desde s_1 a t_j, no puede contener un par de vértices v_i, v_{i+m}. Supongamos que $P = s_1, ..., v_i, ..., v_{i+m}, w, ..., t_j$ donde $1 \leq i+m \leq j$, entonces el camino podría ser reducido a $P' = s_1, ..., v_i, w, ..., t_l$ donde $i \leq l \leq j$. La arista $< v_i, w > \in G'$ por la forma en que se construyó el grafo y el costo de $P' \leq P$, ya que todas las aristas tienen peso no negativo.
Por lo tanto, al aplicar Dijkstra se contruye el camino de costo mínimo desde s hasta t tomando a lo sumo k aristas de P.

3.3. Pseudocódigo

```
1: n ← |V|
2: G' ← vacío de tamaño n(k + 1)                    ▷ Construcción del grafo
3: for i = 0 to k do
4:     for u, v ∈ E(G) do
5:         G'[i * n + u].append(i * n + v)
6:     end for
7: end for
8: for < u, v > in P do:
9:     for i to k do:
10:        G'.append(< u_i, v_{i+1} >)
11:    end for
12: end for
13: d ← Dijkstra(G', s)                             ▷ Encontrar la respuesta
14: return min_{0≤i≤k} d[i * n + t]
```

3.4. Análisis de la complejidad temporal

Construir G' es $O(|V'| + |E'|)$. Aplicar Dijkstra es $O(|E'|log|V'|)$.
Cada vértice del grafo original se repite en cada uno de los $(k+1)$ grafos de G', por lo tanto $|V'| = (k+1)|V|$. Por cada uno de estos $(k+1)$ grafos se repiten también las aristas del grafo G. Además, entre todo par de grafos consecutivos se encuentran las aristas de P que los enlaza por lo que $|E'| = (k+1)|E| + k|P|$. Como $|P| \leq |E|$, entonces $|E'| \leq (2k+1)|E|$
Luego construir G' es $O((k+1)|V| + (2k+1)|E|) = O(k(|V| + |E|))$ y aplicar Dijkstra es $O((2k+1)|E|log(k+1)|V|) = O(k|E|log|V|)$.
Por la regla de la suma la complejidad temporal del algoritmo es $O(k|E|log|V|)$.

4. Problema 4

Sea $< T = V >$ un árbol, con raź $r \in V$, que se comporta como un circuito lógico. Cada hoja (excluyendo la raíz) representa una entrada del circuito y el

resto de los vértices son elementos lógicos, incluyendo la raíz que es la única salida del circuito. Las entradas reciben un bit de información y la salida retorna un bit. Hay 4 tipos de elementos lógicos AND (2 entradas), OR (2 entradas), XOR (2 entradas) y NOT (1 entrada). Los elementos lógicos toman los valores de sus descendientes directos (las entradas) y devuelven el resultado de su operación. Se conoce la estructura de T , el tipo de elemento lógico en sus vértices, la configuración inicial de los bits en las entradas y que hay exactamente una entrada rota. Para arreglar dicha entrada hay que cambiar el valor en ella. Como se desconoce la ubicación de la entrada rota se requiere saber para cada entrada I cuál sería la salida si cambiara el valor de I. Diseñe un algoritmo que resuelva esta problemática con complejidad temporal $O(|V|)$.

4.1. Propuesta de Solución

Para hallar la solución se realizaran dos DFS, un primero para hallar la salida del circuito para las entradas dadas y un segundo en el cual se hallará, para cada nodo del árbol, las siguientes propiedades booleanas:

- yo cambio a mi padre
- mi padre cambia a la raíz

Teniendo estas propiedades se tiene información suficiente para saber por cada entrada, si esta estuviera rota, si esta cambia la salida.

4.2. Correctitud

Este problema puede ser modelado por un árbol donde las hojas son las entradas y el resto de los vértices son los elementos lógicos. Además la raíz contiene la salida del circuito. Esta modelación crea un árbol binario ya que todo nodo tiene a lo sumo dos hijos, en el circuito no existen ciclos y todos elementos se conectan.
En un primer recorrido de DFS se calcula la salida del circuito. Para ello se hallan las salidas de cada nodo a partir de los valores iniciales dados.
Esto se realiza cuando un nodo v va a pasar a ser negro y ya todos sus descendientes son negros, es decir, ya están calculados. En ese momento v le puede preguntar a sus hijo sus valores y según la operación lógica que tenga el nodo, saber su salida. En el caso de las hojas, estas ya saben su valor porque son las entradas.

Demostración por Inducción:

- **Caso base:** Las hojas son las entradas, por lo tanto conocen su valor, es decir, están bien calculadas
- **Hipótesis:** Todo nodo en el $n-$ésimo nivel o superior está bien calculado
- **Tesis:** Si el $n-$ésimo nivel está bien calculado, entonces el $n-1$ nivel se calulará bien

$\mathbf{n \Rightarrow n-1}$
Un nodo v en el nivel $n-1$ conoce su operación y el valor de sus hijos del nivel $n+1$, por lo tanto están bien calculados. Entonces v puede ser calculado y el

resultado es correcto.

Un segundo recorrido por el árbol con un DFS calcula el valor de las propiedades *yo-cambio-a-mi-padre* y *yo-cambio-la-raíz*. Esto se hará en el momento en el que un nodo v sea descubierto.
Sea w el padre de v. En el momento que v sea descubierto, ya w sabe si, cuando él cambia de valor, cambia la salida de la raíz (es decir, la salida del circuito). También w sabrá si cuando la salida de su hijo v cambia, si su salida varía. (Se considera que el nodo raíz siempre cambia la salida del circuito). Cuando el DFS llegue a las hojas, ya estas sabrán si cambian la raíz de modificar ellas sus valores. Es decir, se sabrá para cada hoja i, si esta está rota, el resultado del circuito.

Demostración por Inducción Fuerte:

- **Caso base:** La raíz cambia el valor de salida. Además sus hijos ya están calculados por lo tanto sabe para cada hijo, si este cambia, si la salida de la raíz cambia.
- **Hipótesis:** Cuando el $n-$ésimo nodo es descubierto se calculan bien sus propiedades
- **Tesis:** Cuando el nodo $n+1$ es descubierto por el DFS se calculan bien sus propiedades

Cuando el DFS descuebre el nodo $n+1$ su padre p, donde $1 \leq p \leq n$, ya está bien calculado por hipótesis. $n+1$ conoce si cambia al padre y si su padre cambia a la raíz. Si $n+1$ es capaz de cambiar la salida de p y p cambia la raíz, entonces $n+1$ también cambia la raíz. De lo contrario $n+1$ no afecta la salida del circuito.

4.3. Pseudocódigo

Algorithm 3 Solution

```
1: G.output = DFS-Visit-OutPut(G, u)
2: G.root.changes-root = True
3: G = DFS-Visit-Set-Prop(G, v, True)
4: result ← list
5: for each leave in G do
6:     if leave.changes-root then
7:         result.append(not G.output)
8:     else result.append(G.output)
9:     end if
10: end for
11: return result
```

Algorithm 4 DFS-Visit-OutPut(G, u)

```
1: for each u ∈ Adj[u] do:
2:     if u not visited then
3:         DFS-Visit-OutPut(G, u)
4:         u.output = OutPut(u.right-son, u.left-son)      ▷ Halla el valor de la salida de u
5:     end if
6: end for
7: return G.root.output
```

Algorithm 5 DFS-Visit-Set-Prop(G, v, i-change-parent)

```
1: if not root and i-change-parent and v.parent.changes-root then
2:     v.changes-root = True
3: end if
4: for each u ∈ Adj[v] do:
5:     if u not visited then
6:         child-changes-me = v.Child-Changes-Me(u)
7:         DFS-Visit-Set-Prop(G, u, child-changes-me)
8:     end if
9: end for
10: return result
```

4.4. Análisis de la complejidad temporal

La solución ejecuta dos DFS, los cuales tienen una complejidad temporal de $O(|V| + |E|)$ (solo añaden una propiedad cada uno, lo cual tiene una complejidad de O(1) y no afecta la correctitud o la complejidad del algoritmo). Sin embargo, en este caso se hace DFS sobre un árbol y todo árbol cumple que tiene exactamente $V - 1$ aristas, siendo V la cantidad de vértices en el árbol. Por lo tanto la complejidad temporal sería $O(|V| + (|V| - 1)) = O(2 * |V| - 1) = O(|V|)$