UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Tarea 03

PRESENTA

Gael Bensussen Gonzalez, José Ángel Olmedo Guevara, Jiménez Rivera Emiliano Kaleb

PROFESOR

César Hernandez Cruz

ASIGNATURA

Gráficas y Juegos

7 de mayo de 2025

Tarea 8

1. Sean G una gráfica conexa y $e \in E$. Demuestre que

   a) e está en cada árbol generador de G si y sólo si e es un puente de G;

   b) e no está en árbol generador alguno de G si y sólo si e es un lazo.

2. Sea G una gráfica conexa con conjunto de bloques B y conjunto de vértices de corte C. La *gráfica de bloques y cortes* de G, denotada por $B_C(G)$, esta definida por $V_{B_C(G)} = B \cup C$ y si $u, v \in V_{B_C(G)}$, entonces $uv \in E_{B_C(G)}$ si y sólo si $u \in B$, $v \in C$ y v es un vértice de u. Demuestre que $B_C(G)$ es un árbol.

3. Modifique el algoritmo BFS para que regrese una bipartición de la gráfica (si la gráfica es bipartita) o un ciclo impar (si la gráfica no es bipartita). Su algoritmo debe seguir usando tiempo $O(|V| + |E|)$.

4. Describa un algoritmo basado en BFS para encontrar el ciclo impar más corto en una gráfica. Su algoritmo puede usar tiempo $O(|V||E|)$.

5. Describa un algoritmo de tiempo $O(|V| + |E|)$ para encontrar un bosque generador en una gráfica arbitraria (no necesariamente conexa).

Puntos Extra

1. Demuestre que los algoritmos propuestos en los ejercicios 3, 4 y 5 son correctos (un punto por cada demostracción correcta).

2. Modifique BFS para que sea recursivo en lugar de iterativo.

3. Modifique DFS para que sea recursivo en lugar de iterativo y no utilice una pila (ni otras colecciones).

4. Modifique al algoritmo BFS para que:

   a) Reciba una gráfica no necesariamente conexa con dos vértices distinguidos r y t.

   b) El algoritmo empiece en r, y termine cuando encuentre al vértice t, en cuyo caso lo regresa, junto con una trayectoria de longitud mínima de r a t, o cuando decida que el vértice t no puede ser alcanzado desde r, en cuyo caso regresa el valor `false`.

   c) El primer paso dentro del ciclo `while` sea **eliminar** la cabeza de la cola.

Soluciones:

1a)

Sean G una gráfica conexa y $e \in E(G)$.

1. e está en cada árbol generador syss e es un puente de G.

   Demostración 1 $\Rightarrow$ *Supongamos que e está en cada árbol generador. Supongamos por el contrario que e no es puente de G, i.e., entonces $G-e$ sigue siendo conexa y por teorema existe un árbol generador T para $G-e$. Pero si $\forall z \in T$ está en G, lo que quiere decir que existe un árbol generador sin la arista e, lo cual contradice la hipótesis. Por lo tanto, e es puente.*

   $\Leftarrow$ *Supongamos que e es un puente. Sabemos que los árboles generadores conservan los vértices de G. Además, $G-e$ es inconexa. Si suponemos que existe algún árbol generador en $G-e$, notamos que es imposible, pues quitando a e se generan dos componentes conexas que ninguna gráfica generadora de $G-e$ podrá conectar, y como llegamos a una contradicción de suponer que existía algún árbol generador sin e, concluimos que e está en todo árbol generador.*

1b)

2. e no está en ningún árbol generador de G syss e es un lazo.

   Demostración 2 $\Rightarrow$ *Supongamos que e no está en ningún árbol generador. Por (a), entonces e no es un puente, entonces e es parte de un ciclo en G. Si suponemos por contradicción que C es de longitud 3 o más, entonces bien podríamos hacer un árbol generador excluyendo cualquier otra arista del ciclo diferente de e, por lo que existiría un árbol generador con e, lo cual es una contradicción. Por lo tanto, C es de longitud dos o menos, y la única opción es de longitud uno. Por lo tanto, e es un lazo.*

   $\Leftarrow$ *Supongamos que e es un lazo. Como es un ciclo de longitud 1 y en un árbol generador no hay ciclos, concluimos que ningún árbol generador de G contiene a e.*

2

Veremos que la gráfica es conexa y acíclica. Cómo por teorema la gráfica se puede descomponer en bloques, entre cualesquiera dos de ellos comparten a lo más un vértice y además sabemos que G conexa siempre se podrá llegar de un bloque a otro bloque a otro bloque o de un vértice de corte a otro vértice de corte pues siempre existen caminos entre cualesquiera dos vértices (ya sea de corte o que pertenezca a algun bloque). Ahora veamos que es acíclica. Supongamos que $C = (B_1, v_1, B_2, v_2, ..., B_k v_k, B_1)$ donde B_1 son bloques y v_1 vertices de corte $\forall i \in \{1, .., k\}$. Supongamos, SPG que $k = 3$ (el más chico) ent $C = (B_1, v_1, B_2.v_2, B_3, v_3, B_1)$. Así entonces B_1, B_2, B_3 son bloques i-e maximales con la propiedad de no ser separables pero notemoss que si borramos v_1 del ciclo i.e $B_1 - v_1$ y $B_2 - v_1$ siguen estando conectados todos los vértices de cada bloque y también los vértices de corte y B_1, B_2, B_3 no son maximales sino que $B_1 \cup B_2 \cup B_3$ es no separable; todo esto es una contradicción pues $v_1, v_2 y v_3$ son vértices de corte y B_1, B_2, B_3 eran maximales con la propiedad de ser no separables. COnluimos que $B_C(G)$ es acíclica y por lo tanto es un arbol.

3.

Algorithm 1 Devolver bipartición o ciclo impar en $G = (V, E)$

```
1: for all v ∈ V do
2:     color[v] ← NULL                          ▷ NULL indica "sin asignar"
3:     padre[v] ← NULL
4: end for
5: for all r ∈ V such that color[r] = NULL do
6:     color[r] ← 0
7:     Enqueue(Q, r)
8:     while Q ≠ ∅ do
9:         u ← Dequeue(Q)
10:        for all (u, w) ∈ E do
11:            if color[w] = NULL then
12:                color[w] ← 1 − color[u]
13:                padre[w] ← u
14:                Enqueue(Q, w)
15:            else if color[w] = color[u] then
16:                return ReconstruirCicloImpar(u, w)
17:            end if
18:        end for
19:    end while
20: end for
21: L ← { v ∈ V : color[v] = 0}
22: R ← { v ∈ V : color[v] = 1}
23: return (L, R)
```

Algorithm 2 ReconstruirCicloImpar(u, w)

$P_u \leftarrow [\,]$, $P_w \leftarrow [\,]$
$x \leftarrow u$, $y \leftarrow w$
while $x \neq y$ **do**
    if profundidad$(x) \geq$ profundidad(y) **then**
        Append(P_u, x)
        $x \leftarrow padre[x]$
    else
        Append(P_w, y)
        $y \leftarrow padre[y]$
    end if
end while
$z \leftarrow x$ ▷ Ancestro común
Append(P_u, z)
Reverse(P_w) ▷ invierte para recorrer de z a w
Concatenar $C \leftarrow P_u \,\|\, P_w$
Append(C, u) ▷ cerramos el ciclo con la arista (w, u)
return C

4

Algorithm 3 Ciclo Impar Más Corto en $G = (V, E)$

```
minLen ← ∞
for all v ∈ V do
    color[v] ← NULL
    padre[v] ← NULL
    dist[v] ← ∞
end for
for all r ∈ V tal que color[r] = NULL do
    color[r] ← 0
    dist[r] ← 0
    padre[r] ← NULL
    Enqueue(Q, r)
    while Q ≠ ∅ do
        u ← Dequeue(Q)
        for all (u, w) ∈ E do
            if color[w] = NULL then
                color[w] ← 1 − color[u]
                dist[w] ← dist[u] + 1
                padre[w] ← u
                Enqueue(Q, w)
            else if w ≠ padre[u] and color[w] = color[u] then
                len ← dist[u] + dist[w] + 1
                if len < minLen then
                    minLen ← len
                    u0 ← u, w0 ← w
                end if
            end if
        end for
    end while
end for
if minLen < ∞ then
    C_best ← ReconstruirCicloImpar(u0, w0, padre)
    return C_best
else
    return "No hay ciclo impar"
end if
```

Algorithm 4 ReconstruirCicloImpar($u, w, padre$)

$P_u \leftarrow [\,], \quad P_w \leftarrow [\,]$
$x \leftarrow u, \quad y \leftarrow w$
while $x \neq y$ **do**
    if $dist[x] \geq dist[y]$ **then**
        $P_u \leftarrow [x] \cup P_u$
        $x \leftarrow padre[x]$
    else
        $P_w \leftarrow [y] \cup P_w$
        $y \leftarrow padre[y]$
    end if
end while
$P_u \leftarrow [x] \cup P_u$ ▷ añadir LCA
Reverse(P_w)
$C \leftarrow P_u \cup P_w$ ▷ unir ramas y arista (u, w)
return C

5.

Algorithm 5 Algoritmo de Bosque Generador
I: gráfica G
O: F=(V,E) bosque generador

$F \leftarrow \emptyset$
for cada vértice $v \in V$ no visitado **do**
    $Q \leftarrow [\,]$
    Colorear v de negro
    Enqueue(Q, v)
    while $Q \neq [\,]$ **do**
        Sacar x del tope de Q
        if x tiene vecino z no coloreado **then**
            $F \leftarrow F \cup \{x, z\}$
            Colorear z de negro
            Enqueue(Q, z)
        else
            Eliminar x de Q
        end if
    end while
end for
return F

Puntos extras

1)

¿Por qué el algoritmo de la gráfica bipartita encuentra el ciclo impar más corto?

Es impar porque al observar que cualquier vértice a una distancia par de la raíz tiene la misma coloración que la raíz y aquellos a distancia imoar el color contrario.
Como en el algoritmo se encuentran dos vértices del mismo color si su ancesto compún es la raíz, solo están las opciones par + par + 1 = impar o impar + impar + 1 = par + 1 = imoar donde 1 es la arsira que los une. Pero si r no es su ancestro compun entonces supongamos que es w así la distancia de w a ambos extremos $x, y \in T$ es $l(x) - l(w)$ y $l(y) - l(w)$. Así la longitud del ciclo será: $l(x) - l(w) + l(y) - l(w) + 1 = l(x) + l(y) - 2l(w) + 1$
sI $l(x)$ y $l(y)$ son pares:
$2k - 2q - 2l(w) + 1 = 2(k + q - l(w)) + 1$ i.e impar.
Si $l(x)$ y $l(y)$ impares: $(2k+1) + (2q+1) - 2l(w) + 1 = 2(k+q-l(w)) + 2 + 1 = 2([k+1] - l(w)] + 1$ i.e impar.

Demostración de Corrección del Algoritmo 3: Bipartición o Ciclo Impar

Especificación

El Algoritmo 3 debe devolver una bipartición (L, R) de la gráfica $G = (V, E)$ si es bipartita, o un ciclo impar si no lo es, en tiempo $O(|V| + |E|)$.

Invariantes de Bucle

Consideremos el Algoritmo 1 (página 5 de tarea08.pdf). Para cada componente conexa procesada por BFS, después de cada iteración del bucle `while` (líneas 8–16):

1. Todos los vértices en la cola Q y los ya procesados están coloreados (0 o 1). Esto asegura que cada vértice visitado tiene un color asignado, lo que es necesario para verificar bipartición.

2. Si un vértice w está coloreado, entonces color$[w] = 1 -$ color[padre$[w]$] (si padre$[w] \neq$ NULL). Esto garantiza que los colores alternen en el árbol BFS, una propiedad clave para la bipartición.

3. Si no se ha encontrado un ciclo impar, los vértices coloreados forman una bipartición válida de la subgráfica inducida por los vértices visitados, es decir, no hay aristas entre vértices del mismo color.

4. Si se encuentra un conflicto (color$[u] =$ color$[w]$ para u, w adyacentes, línea 12), existe un ciclo impar, lo que indica que la gráfica no es bipartita.

Demostración de Corrección

- **Inicialización**: Antes de procesar cada componente conexa, en las líneas 1–3, se inicializa color$[v] =$ NULL, padre$[v] =$ NULL para todo $v \in V$. En la línea 5, para cada r no coloreado, se asigna color$[r] = 0$, y en la línea 6 se encola r en Q. En este punto, $Q = \{r\}$, color$[r] = 0$, y todos los demás vértices no tienen color. Los invariantes 1 y 2 se cumplen trivialmente (solo r está coloreado, y no tiene padre). El invariante 3 se cumple porque la subgráfica inducida solo tiene un vértice (r). El invariante 4 aún no aplica, ya que no se ha encontrado un conflicto.

- **Mantenimiento**: Consideremos el bucle `while` (líneas 8–16) dentro del bucle `for` (líneas 4–18).
  - **Línea 9**: Se desencola un vértice u de Q. Por el invariante 1, u ya está coloreado (fue encolado en la línea 6 o 13 con un color asignado).
  - **Líneas 10–14**: Para cada vecino w de u (bucle `for`):
    - **Líneas 11–13**: Si color$[w]$ = NULL, entonces w no ha sido visitado. Se asigna color$[w] = 1 - $ color$[u]$ (línea 11), lo que asegura el invariante 2, ya que w recibe un color opuesto al de su padre u (establecido en padre$[w] = u$, línea 12). Luego, w se encola (línea 13), manteniendo el invariante 1, ya que w ahora está coloreado y en Q.
    - **Líneas 12–13**: Si color$[w] \neq$ NULL y color$[w]$ = color$[u]$ (línea 12), se ha encontrado un conflicto: u y w son adyacentes y tienen el mismo color. Esto viola la propiedad de bipartición, indicando un ciclo impar (invariante 4). El algoritmo llama a `ReconstruirCicloImpar` (Algoritmo 2, página 6) para construir el ciclo:
      - ◇ En las líneas 1–11 de `ReconstruirCicloImpar`, se trazan los caminos desde u y w hasta un ancestro común z usando padre. Por el invariante 2, los colores alternan a lo largo de estos caminos.
      - ◇ En las líneas 12–14, se construye el ciclo $C = u \to P_u \to z \to P_w \to w \to u$. Dado que color$[u]$ = color$[w]$, y los colores alternan, la longitud del ciclo $|P_u| + |P_w| + 1$ es impar (dos caminos de igual paridad más una arista).
  - Si no se encuentra conflicto en esta iteración, el invariante 3 se mantiene: no hay aristas entre vértices del mismo color en la subgráfica visitada, ya que cada nuevo vértice w recibe un color opuesto al de su padre.
- **Terminación**: El bucle `while` (líneas 8–16) termina cuando $Q = \emptyset$, lo que ocurre cuando todos los vértices de la componente conexa han sido visitados (BFS explora todos los vértices alcanzables desde r). El bucle exterior (líneas 4–18) procesa cada componente conexa. Al finalizar, si no se encontró un ciclo impar, las líneas 19–20 construyen $L = \{v : \text{color}[v] = 0\}$ y $R = \{v : \text{color}[v] = 1\}$, y la línea 21 devuelve (L, R). Por el invariante 3, (L, R) es una bipartición válida, ya que no hay aristas entre vértices del mismo color.
- **Corrección de la Salida**:
  - Si G es bipartita, no se encuentran conflictos de color (línea 12 nunca se cumple), y (L, R) es una bipartición válida, ya que las aristas solo conectan vértices de L con R (por el invariante 3).
  - Si G no es bipartita, contiene un ciclo impar. En algún momento, la condición de la línea 12 se cumple, detectando un conflicto de color, y `ReconstruirCicloImpar` devuelve un ciclo impar válido (invariante 4).
- **Complejidad**: BFS procesa cada vértice y arista una vez por componente, dando $O(|V|+|E|)$. La reconstrucción del ciclo impar (Algoritmo 2) es lineal en la longitud del camino, $O(|V|)$, lo que no afecta la complejidad total.

Por lo tanto, el Algoritmo 3 es correcto, pues devuelve una bipartición si G es bipartita o un ciclo impar si no lo es, en tiempo $O(|V| + |E|)$.

Demostración de Corrección del Algoritmo 4: Ciclo Impar Más Corto

Especificación

El Algoritmo 4 debe devolver el ciclo impar de longitud mínima en $G = (V, E)$, o indicar que no existe, en tiempo $O(|V||E|)$.

Invariantes de Bucle

Consideremos el Algoritmo 3 (página 7 de tarea08.pdf). Para cada BFS iniciado en r, después de cada iteración del bucle `while` (líneas 11–23):

1. dist[v] es la distancia mínima desde r a v en la componente conexa de r, asegurando que las distancias sean correctas para calcular longitudes de ciclos.
2. color[v] = 0 si dist[v] es par, color[v] = 1 si es impar, manteniendo una relación entre distancias y colores para detectar ciclos impares.
3. Si ocurre un conflicto (color[u] = color[w] para u, w adyacentes, $w \neq$ padre[u], línea 17), entonces len = dist[u] + dist[w] + 1 es la longitud de un ciclo impar.
4. minLen es la longitud mínima de los ciclos impares encontrados hasta el momento, garantizando que se rastree el ciclo más corto.

Demostración de Corrección

- **Inicialización**: En las líneas 1–5, se inicializan minLen = ∞, color[v] = NULL, dist[v] = ∞, padre[v] = NULL para todo $v \in V$. Para cada r no coloreado (líneas 6–23), en las líneas 7–9 se asigna color[r] = 0, dist[r] = 0, padre[r] = NULL, y en la línea 10 se encola r. En este punto, dist[r] = 0 es la distancia mínima (invariante 1), color[r] = 0 satisface el invariante 2 (dist[r] es par), y los invariantes 3 y 4 se cumplen trivialmente (no hay conflictos aún, y minLen = ∞).
- **Mantenimiento**: Consideremos el bucle `while` (líneas 11–23) dentro del bucle `for` (líneas 6–25).
  - **Línea 12**: Se desencola un vértice u. Por el invariante 1, dist[u] es la distancia mínima desde r, y por el invariante 2, color[u] corresponde a la paridad de dist[u].
  - **Líneas 13–21**: Para cada vecino w de u:
    - **Líneas 14–17**: Si color[w] = NULL, w no ha sido visitado. En la línea 14, se asigna color[w] = 1−color[u], en la línea 15, dist[w] = dist[u]+1, en la línea 16, padre[w] = u, y en la línea 17, se encola w. Esto mantiene el invariante 1, ya que dist[w] es la distancia mínima a través de u (BFS explora en orden de distancia). El invariante 2 se cumple porque si dist[u] es par (color[u] = 0), entonces dist[w] = dist[u] + 1 es impar y color[w] = 1, y viceversa.
    - **Líneas 18–21**: Si $w \neq$ padre[u] y color[w] = color[u] (línea 18), hay un conflicto. Por el invariante 2, dist[u] y dist[w] tienen la misma paridad. En la línea 19, se calcula len = dist[u] + dist[w] + 1, que es impar (suma de dos números de igual paridad más 1), cumpliendo el invariante 3. En las líneas 20–21, si len $<$ minLen, se actualiza minLen, u_0, y w_0, manteniendo el invariante 4.
  - Si no hay conflicto, los invariantes 1 y 2 aseguran que las distancias y colores siguen siendo correctos para la siguiente iteración.

- **Terminación**: El bucle `while` (líneas 11–23) termina cuando $Q = \emptyset$, explorando toda la componente conexa de r. El bucle exterior (líneas 6–25) repite este proceso para cada r, asegurando que todas las componentes sean procesadas. Al finalizar, en las líneas 26–29, si minLen $< \infty$, se reconstruye el ciclo impar más corto usando `ReconstruirCicloImpar` (línea 27); de lo contrario, se devuelve "No hay ciclo impar" (línea 29). Por el invariante 4, minLen contiene la longitud mínima de todos los ciclos impares encontrados.
- **Corrección de la Salida**:
  - Si G tiene un ciclo impar, al menos un BFS detecta un conflicto (línea 18). Como se ejecuta BFS desde cada vértice (línea 6), todos los posibles ciclos impares se consideran. Por el invariante 4, minLen refleja la longitud del ciclo impar más corto, y `ReconstruirCicloImpar` (Algoritmo 4, página 8) lo reconstruye correctamente.
  - Si G no tiene ciclos impares, no se activa la condición de la línea 18, minLen permanece infinito, y el algoritmo devuelve "No hay ciclo impar".
- **Complejidad**: Cada BFS toma $O(|V| + |E|)$, y se ejecuta $|V|$ veces (línea 6), dando $O(|V| \cdot (|V| + |E|)) = O(|V||E|)$, como se especifica.

Por lo tanto, el Algoritmo 4 es correcto, pues encuentra el ciclo impar de longitud mínima (si existe) en tiempo $O(|V||E|)$.

Demostración de Corrección del Algoritmo 5: Bosque Generador

Especificación

El Algoritmo 5 debe devolver un bosque generador $F = (V, E_F)$ de una gráfica arbitraria $G = (V, E)$ (no necesariamente conexa) en tiempo $O(|V| + |E|)$.

Invariantes de Bucle

Consideremos el Algoritmo 9 (página 10 de tarea08.pdf). Para cada componente conexa procesada, después de cada iteración del bucle `while` (líneas 6–12):

1. F es un bosque (sin ciclos) que contiene un subconjunto de los vértices visitados, asegurando que el resultado parcial sea válido.
2. Todos los vértices en Q y los ya procesados están coloreados, garantizando que cada vértice se procese una sola vez.
3. Las aristas en F conectan vértices de la misma componente conexa y forman árboles parciales, manteniendo la estructura de árbol dentro de cada componente.

Demostración de Corrección

- **Inicialización**: En la línea 1, se inicializa $F = \emptyset$. En las líneas 2–12, para cada vértice $v \in V$ no visitado, se procesa su componente conexa. En este punto, ningún vértice está coloreado, F está vacío (es un bosque, invariante 1), y $Q = \emptyset$ (invariante 2 se cumple trivialmente). El invariante 3 también se cumple, ya que F no tiene aristas.
- **Mantenimiento**: Consideremos el bucle `while` (líneas 6–12) dentro del bucle `for` (líneas 2–13).

  - **Líneas 3–5**: Para cada vértice v no visitado, se inicializa $Q = \emptyset$, se colorea v de negro (línea 4), y se encola v (línea 5). Esto asegura el invariante 2 para v.
  - **Línea 7**: Se saca x del tope de Q. Por el invariante 2, x ya está coloreado (fue encolado en la línea 5 o 10).
  - **Líneas 8–10**: Si x tiene un vecino z no coloreado (línea 8), se añade (x, z) a F (línea 9), se colorea z de negro (línea 10), y se encola z (línea 10). Dado que z no está coloreado, (x, z) no crea un ciclo en F (invariante 1), ya que z es un vértice nuevo en el árbol que se está construyendo. El invariante 2 se mantiene porque z ahora está coloreado y en Q. El invariante 3 se cumple, ya que (x, z) conecta x y z dentro de la misma componente, y F sigue siendo un árbol parcial (las aristas forman un camino desde v a z a través de los padres).
  - **Línea 11**: Si x no tiene vecinos no coloreados, se elimina x de Q, manteniendo el invariante 2 (los vértices en Q siguen coloreados).

- **Terminación**: El bucle `while` (líneas 6–12) termina cuando $Q = \emptyset$, lo que ocurre cuando todos los vértices de la componente conexa de v han sido procesados. El bucle exterior (líneas 2–13) asegura que todas las componentes sean procesadas. En la línea 14, se devuelve F, que contiene las aristas acumuladas.

- **Corrección de la Salida**:

  - **Conexión**: Por el invariante 3, F contiene un árbol para cada componente conexa, ya que el bucle `while` explora todos los vértices alcanzables desde v (líneas 6–12).
  - **Sin Ciclos**: El invariante 1 asegura que F es acíclico, ya que las aristas solo se añaden para vértices no coloreados (línea 9).
  - **Generador**: El bucle exterior (líneas 2–13) garantiza que todos los vértices de G sean incluidos en F, ya que cada vértice no visitado inicia una nueva componente.
  - Por lo tanto, F es un bosque generador de G.

- **Complejidad**: Cada vértice y arista se procesa una vez (líneas 6–12), dando $O(|V| + |E|)$, como se especifica.

Por lo tanto, el Algoritmo 5 es correcto, pues produce un bosque generador de G en tiempo $O(|V| + |E|)$.

2)

Algorithm 6 BFS Recursivo
I: gráfica G, Q una cola de vértices, i $\in N$
O: funciones p (parentesco), l (nivel) y t (tiempo)

```
1: if Q = [ ] then
2:     return (p, l, t)
3: end if
4: x ← Dequeue(Q)
5: for all vecino z de x no coloreado do
6:     i ← i + 1
7:     Colorear a z de negro
8:     p(z) ← x
9:     t(z) ← i
10:    l(z) ← l(x) + 1
11:    Enqueue(Q, z)
12: end for
13: return BFSRecursivo(Q, i)
```

Algorithm 7 Función Auxiliar BFS
I: gráfica G y r vértice distinguido
O: funciones p, l, t

```
1: i ← 1; Q ← [ ]
2: l(r) ← 0; t(r) ← i; p(r) ← ∅
3: Enqueue(Q, r)
4: Colorear a r de negro
5: return BFSRecursivo(Q, i)
```

3)

Algorithm 8 DFS Recursivo
I: Gráfica G, vértice v, i $\in N$
O: contador i

```
1: f(v) ← i
2: i ← i + 1
3: for all vecino z sin colorear de v do
4:     i ← i + 1
5:     Colorear z de negro
6:     p(z) ← v
7:     f(z) ← i
8:     i ← DFSRecursivo(G, z, i)
9: end for
10: i ← i + 1
11: l(v) ← i
12: return i
```

Algorithm 9 Función Auxiliar DFS
I: gráfica G y r vértice distinguido
O: funciones p, l, t

```
1: function FuncionAuxiliarDFS(G, r)
2:     p(r) ← ∅
3:     Colorear a r de negro
4:     DFSRecursivo(G, r, 0)
5:     return (p, f, l)          ▷ Considerando que ya las modificó el algoritmo principal
6: end function
```

4)

Algorithm 10 BFS Modificado para encontrar camino mínimo entre dos vértices
I: G = (V, E) una gráfica, r y t vértices distinguidos
O: Trayectoria de longitud mínima de r a t, o false si no hay camino

```
1: Q ← []
2: for cada vértice v ∈ V do
3:     p(v) ← ∅
4:     ℓ(v) ← ∞
5: end for
6: colorear a r de negro
7: añadir r al final de Q
8: ℓ(r) ← 0
9: while Q ≠ [] do
10:    eliminar la cabeza u de Q                    ▷ modificación (c)
11:    if u = t then
12:        return ReconstruirCamino(p, r, t)        ▷ encontramos t
13:    end if
14:    for cada vecino v de u no coloreado do
15:        colorear a v de negro
16:        p(v) ← u
17:        ℓ(v) ← ℓ(u) + 1
18:        añadir v al final de Q
19:    end for
20: end while
21: return false                                    ▷ t no puede ser alcanzado desde r
22: procedure ReconstruirCamino(p, r, t)
23:    camino ← [t]
24:    actual ← t
25:    while actual ≠ r do
26:        actual ← p(actual)
27:        insertar actual al inicio de camino
28:    end while
29:    return camino
30: end procedure
```