Universidad Nacional Autónoma de México

Facultad de Ciencias

Tarea 08

Presentan

- Nelson Osmar Garcia Villa - 322190357
- Valeria Camacho Hernández - 322007273
- Mauricio Casillas Álvarez - 322196342

Asignatura

Gráficas y Juegos 2025-2

Profesor

César Hernández Cruz

Ayudante

Iñaki Cornejo de la Mora

Fecha

Viernes 02 de mayo del 2025

Tarea 08

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;
   Respuesta: Procedemos por doble implicación.

   Supongamos por contradicción que e no es un puente. Entonces al eliminar e, la gráfica sigue siendo conexa. Sea entonces $G - \{e\} = G'$.
   Luego, como G' es conexa y tiene todos los vértices de G, entonces tiene al menos un árbol generador T' que no contiene a e. Esto contradice la suposición de que e está en todos los árboles generadores de G. Por lo tanto, e debe ser un puente.
   $\therefore$ e tiene que ser puente para que pertenezca a todos los árboles generadores de G.
   [$\Leftarrow$ Si e es un puente de G, entonces e está en cada árbol generador.
   Supongamos por contradicción que existe un árbol T tal que $T \subseteq G$ y que dicho árbol no contenga a e.
   Como T es un árbol generador de G, quiere decir que T es una subgráfica, la cual debe ser conexa al igual que debe contener todos los vértices de G.
   La hipótesis nos dice que e es un puente, pero si al borrar la arista los dos extremos de la arista e no están conectados, entonces ya no se puede llegar por ningún camino a uno de los extremos de e con otro extremo, por definición.
   Implica que si $T \subseteq G - \{e\}$, entonces T no puede ser conexo, lo que rompe la definición de árbol.
   $\therefore$ Todo árbol generador de G debe tener la arista e.

   (b) e no está en árbol generador alguno de G si y sólo si e es un lazo.
   Respuesta:
   $\Rightarrow$] Si e no está en ningún árbol generador, entonces e es un lazo.
   Supongamos que existe una arista arbitraria e que une a u y v, siendo estos dos vértices diferentes.
   Por condición del ejercicio, G es conexa, implica que exista un árbol generador $T \in G$ que tiene a todos los vértices de G (incluyendo a u y v).
   Entonces como e conecta dos vértices distintos, al agregar la arista a T
   Pero si al agregar e siempre se forma un ciclo sin poder eliminarse, significa que e no conecta dos componentes diferentes, simplemente conecta un vértice consigo mismo.
   [$\Leftarrow$ Si e es un lazo, entonces e no está en ningún árbol generador.
   Un lazo conecta un vértice consigo mismo y forma un ciclo de longitud 1. Como los árboles no tienen ciclos, un lazo no puede estar en un árbol generador. Por eso, si cualquier e es un lazo, nunca puede pertenecer a ningún árbol generador.
   $\therefore$ e no aparece en ningún árbol generador.

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.

   Respuesta: Procedemos a demostrar dos cosas:

   Por demostrar: $B_C(G)$ es conexa.

   Supongamos que G es conexa. Entonces cada bloque de G es conexo y sin vértices de corte. Por definición, los vértices de corte son los que unen bloques entre sí. En $B_C(G)$, cada bloque está conectado a los vértices de corte que lo tocan.

   Entonces, si recorremos de un bloque a otro en G pasando por vértices de corte, eso mismo se refleja en $B_C(G)$.

   $\therefore B_C(G)$ es conexo.

   Supongamos que en $B_C(G)$ hay un ciclo, significaría que existe una secuencia de bloques y vértices de corte conectados $(b_1, c_1, b_2, c_2, ..., b_k, c_k, b_1)$.

   Implicaría que en G hay un camino que conecta bloques de manera cíclica.

   Pero los bloques en G son subgráficas conexas sin vértices de corte.

   Así que formar un ciclo en $B_C(G)$ significaría que en G puedes unir bloques de dos maneras diferentes sin pasar por un vértice de corte nuevo, lo cual rompe la definición de bloque.

   $\therefore B_C(G)$ no tiene ciclos.

   $\therefore B_C(G)$ 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 $O(| V | + | E |)$.

   Respuesta: Proponemos el siguiente algoritmo.

   Input: Una gráfica G como lista de adyacencias y un vértice r.

   Output: Tenemos dos posibles opciones:

   1) Si la gráfica es bipartita, entonces devuelve la bipartición (A, B).

   2) Si la gráfica no es bipartita, entonces devuelve un ciclo impar como lista de vértices y en orden.

   1. color $\leftarrow \emptyset$
   2. padre $\leftarrow \emptyset$
   3. nivel $\leftarrow \emptyset$
   4. cola $\leftarrow [r]$
   5. color[r] $\leftarrow 0$
   6. nivel[r] $\leftarrow 0$
   7. while la cola no esté vacía do
   8. sacar un nodo v de la cola
   9. para cada vecino w de v do
   10. if w no está en color then
   11. color$[w] \leftarrow 1 - color[v]$; Pintamos del color opuesto
   12. padre$[w] \leftarrow v$
   13. nivel$[w] \leftarrow$ nivel$[v] + 1$
   14. añadir w a la cola
   15. else if color$[w]$ = color$[v]$ then ; Tenemos un conflicto, entonces hacemos un ciclo impar
   16. reconstruir camino de v hasta la raíz con padre[ ]
   17. reconstruir camino de w hasta la raíz con padre[ ]
   18. encontrar el ancestro común
   19. unir los dos caminos y agregar arista (v, w)
   20. devolver el ciclo impar
   21. crear conjuntos $A \leftarrow \emptyset$, $B \leftarrow \emptyset$
   22. for each v en color do
   23. if color$[v] = 0$ then

24. agregar v a A

24. else

24. agregar w a B

25. devolver (A, B)

Cuando tenemos color[v] = color[w] significa que hay una arista entre dos nodos del mismo color. Como BFS va por niveles, lo que hacemos es usar los padre[v] para reconstruir un camino desde v y w hasta su ancestro común (como en árboles), y unir esos caminos para obtener el ciclo. Este ciclo siempre tendrá longitud impar, porque los nodos tenían la misma "distancia" al origen (o diferencia par), y están conectados.

Además,cuando hacemos el BFS, cada vértice de V se procesa vez pues se visita cada nodo a lo más una vez, así que tenemos $O(\mid V \mid)$. Luego, cuando se exploran las aristas incidentes a cada uno es igual una vez, así que tenemos $O(\mid E \mid)$. Entonces, BFS es $O(\mid V \mid + \mid E \mid)$. ★

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

   Respuesta: Proponemos el siguiente algoritmo.

   Input: Una gráfica $G = (V, E)$.

   Output: El ciclo impar más corto en G.

```
1. ciclo ← ∅; l ← 0
2. for cada vértice r ∈ V do;
3.         for cada vértice v ∈ G do;
4.                 visitado v ← falso
5.                 padre (v) ← null //Guardamos el padre para reconstruir caminos
6.                 nivel (v) ← −1 // Nivel, distancia desde r
7.         nivel (r) ← 0
8.         color (r) ← 0 //Usamos 0 y 1 según nivel
9.         C ← {r}
10.         while C diferente de ∅ do
11.                 sacar x ∈ C
12.                 for cada y ∈ vecinos x do;
13.                         if no visitado(y) then;
14.                                 nivel (y) ← nivel (x) + 1
15.                                 padre (y) ← x
16.                                 color (y) ← (color (x) +1) mod 2
17.                                 visitado (y) ← verdadero
18.                                 agregar a C
19.             else
20.                         if y diferente de padre(x) y color (y) = color (x) then
21.                                 ciclotemp ← construir ciclo desde x a y
22.                                 if longitud (ciclotemp) < l y es impar then;
23.                                         ciclo ← ciclotemp
24.                                                 l ← longitud (ciclotemp)
25.         end
26. end
27. return ciclo
```

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

Respuesta: Proponemos el siguiente algoritmo para encontrar un bosque generador.

Input: Gráfica arbitraria (no necesariamente conexa).
Output: Un bosque generador (unión de árboles generadores, uno por cada componente conexa de G).

1. Hacemos la inicialización:
   - Marcar todos los vértices $v \in V$ como no *visitados*.
   - Inicializar un conjunto vacío F para almacenar las aristas del bosque generador.
2. Hacemos el recorrido por componentes conexas, para cada vértice $v \in V$ no visitado:
   - Aplicar un recorrido, BFS o DFS, desde v.
   - Durante el recorrido, marcar los vértices alcanzables como *visitados* y registrar las aristas que los conectan.
   - Estas aristas forman un árbol generador para la componente conexa actual.
   - Agregar las aristas de árbol a F.
3. Hacemos su conclusión:
   - El algoritmo termina cuando todos los vértices han sido visitados.
   - El resultado es un bosque generador, donde cada árbol corresponde a una componente conexa de G.

Ahora, como cada vértice y arista se procesa una única vez, el tiempo de ejecución es $O(\mid V \mid + \mid E \mid)$.

Puntos Extra

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

   Ejercicio 3. Queremos demostrar dos cosas: que sea correcto y su complejidad. Procedemos como sigue.

   1) Por demostrar: El algoritmo es correcto. Procedemos a ver los siguientes casos.

   - Caso 1. G es una gráfica bipartita.

     Utilizamos una búsqueda por niveles (BFS), donde a cada vértice se le asigna un color (0 o 1, que equivale a decir que los pone en niveles: pares o impares) y a sus vecinos se le asigna el color opuesto. En el algoritmo, si la gráfica es bipartita, nunca habrá una arista que conecta dos vértices del mismo color, así que la línea 15 nunca se ejecuta. Por lo tanto, al finalizar, tenemos a los conjuntos A (color 0) y B (color 1), los cuales forman una bipartición válida porque toda arista conecte un vértice A con uno de B.

   - Caso 2. G no es una gráfica bipartita.

     Si existe una arista (v, w) con color$[v]$ =color$[w]$, el algoritmo entra en la línea 15. Esto indica que u y v están en el mismo nivel o en niveles iguales (por BFS). Entonces el camino desde la raíz hasta cada uno tiene la misma longitud, y si los uno con la arista que los conecta directamente, se forma un ciclo, el cual tiene longitud $2k + 1 =$ impar.

     Es decir,, se reconstruyen los caminos desde v y w hasta su ancestro común en el árbol BFS, y como los nodos tienen la misma distancia a la raíz, sus caminos al ancestro que tienen en común son de igual longitud. Entonces, se combinan los caminos de v a u y de w a u junto con la aristas (v, w), lo cual nos da un ciclo de longitud impar.

   $\therefore$ El algoritmo siempre detecta correctamente si la gráfica es bipartita o no, y en el caso en que no bipartita, devuelve correctamente un ciclo impar.

   2) Por demostrar: Su complejidad es $O(\mid V \mid + \mid E \mid)$.

   La inicialización de las estrucuturas de datos es $O(1)$. Luego, cuando hacemos el BFS, cada vértice de V se procesa vez pues se visita cada nodo a lo más una vez (así tenemos $O(\mid V \mid)$) y cuando se exploran las aristas incidentes a cada uno es igual una vez (así tenemos $O(\mid E \mid)$). Entonces tenemos que BFS puede recorrer $O(\mid V \mid + \mid E \mid)$, pero cuando ocurre el peor caso (el conflicto de tener dos vértices del mismo color), se reconstruye el camino al ancestro común entre ambos, lo cual puede ocurrir $O(\mid V \mid)$. Entonces, tenemos $O(\mid V \mid \cdot \mid V \mid + \mid E \mid)$. Sin embargo, el caso cuando ocurre el conflicto son pocas veces, entonces tenemos que cuando hacemos BFS, su complejidad es $O(\mid V \mid + \mid E \mid)$, como inicialmente explicamos.

   $\therefore$ Se cumple.

Por lo tanto, el algoritmo es correcto porque asigna colores válidos y devuelve (A, B) si no hay conflictos (hace la bipartición), detecta la primera arista conflictiva y construye un ciclo impar (si no hay bipartición), y mantiene la complejidad deseada. ★

Ejercicio 4. Queremos demostrar que es correcto y su complejidad. Procedemos como sigue.

1) Por demostrar: Obtiene los ciclos impares.

Durante el BFS, si un vecino del vértice actual u ya fue visitado y tiene el mismo color (mismo nivel), entonces existe un ciclo impar. Esto sucede porque el camino desde la raíz hasta u y hasta v tiene la misma longitud (por estar en el mismo nivel). Entonces, la arista (u, v) forma un ciclo cuya longitud es $2k + 1$ (impar), donde k es el nivel de u y v. Esto sucede por la reconstrucción del ciclo.

$\therefore$ Se cumple.

2) Por demostrar: Encuentra el ciclo impar más corto.

Cada vez que se hace BFS (osea por cada vértice), el primer ciclo impar que se detecta es el más corto que tiene como raíz a r, pues BFS garantiza que sea la distancia menor, entonces se guarda el más corto entre todos los y así se asegura que el ciclo impar más corto en toda la gráfica será devuelto.

$\therefore$ Se cumple.

3) Por demostrar: Es una gráfica bipartita (si no hay un conflicto de colores).

Si nunca se encuentra un conflicto de colores durante todos los BFS, eso significa que no hay ciclos impares, entonces la gráfica es bipartita. Así, los colores asignados (0/1) forman la bipartición.

$\therefore$ Se cumple.

4) Por demostrar: Su complejidad es $O(\mid V \mid \cdot \mid E \mid)$.

El algoritmo busca el ciclo impar más corto, no todos los ciclos impares. Entonces una vez que encuentra un ciclo impar, lo guarda como el más corto, pero si más adelante encuentra otro ciclo impar más corto, lo actualiza. Luego, si en algún momento encuentra un ciclo impar de longitud 3 (el mínimo posible), puede parar porque ya no habrá uno más corto.

Entonces, para cada vértice de V, el algoritmo hace BFS por cada vértice de la gráfica y como el número total de vértices es $|V|$, se hacen $|V|$ veces el recorrido BFS. Además, se recorren todos los vértices que son alcanzados por r, encuentra un ciclo impar, y guarda el ciclo más corto cuando hay varios, es decir, se visitan todos los vértices desde el origen y se exploran todas las aristas conectadas, por lo que BFS recorre a $O(|V| + |E|)$ en el peor caso (el conflicto). Aspi, tenemos que su complejidad es $O(\mid V \mid \cdot(|V| + |E|))$.

$\therefore$ No sabemos si cumple como pide.

Por lo tanto, el algoritmo es si detecta ciclos impares si hay dos vértices con el mismo color al aplicarle BFS, reconstruye correctamente el ciclo más corto usando los ancestros comunes que tenga y su complejidad es al cuadrado.

Ejercicio 5. Queremos demostrar tres cosas: correctud, complitud y su complejidad. Procedemos como sigue.

1) Por demostrar: Genera un bosque generador válido (un árbol por componente conexa).

- Cada componente conexa genera exactamente un árbol.

  Al iniciar el algoritmo de búsqueda desde un vértice no visitado v, se exploran todos los vértices alcanzables desde v (su componente conexa). Por definición, el algoritmo de búsqueda visita todos los vértices conexos a v. Luego, las aristas seleccionadas son aquellas que conectan un vértice visitado con uno no visitado, formando un árbol (porque no se repiten vértices ni se forman ciclos).

  Ahora, si se formara un ciclo en el árbol de una componente, existirían dos caminos distintos entre un par de vértices, pero BFS/DFS solo incluye aristas que descubren vértices por primera vez, evitando redundancias.

- Las aristas seleccionadas forman un árbol generador.

  Tenemos conexión porque para cualquier par de vértices u, w en la misma componente, existe un camino en el árbol generador pues el algoritmo de búsqueda encuentra un camino entre u y w en G. Así, las aristas de árbol preservan este camino (sin aristas redundantes).

  Ahora, tenemos maximalidad porque el árbol generador tiene $k1$ aristas para una componente con k vértices (propiedad de árboles), y si faltara una arista, habría vértices desconectados, contradiciendo la conexidad de la componente.

$\therefore$ Se cumple.

2) Por demostrar: Cubre todos los vértices y aristas necesarias.

- Todas las componentes conexas son procesadas.

  En su inicialización, todos los vértices están marcados como no visitados. En la iteración bucle principal itera sobre cada vértice $v \in V$ y si v no está visitado, se procesa su componente conexa completa. Luego, como G puede ser disconexa, cada nueva llamada al algoritmo de búsqueda desde un vértice no visitado cubre una componente conexa distinta.

- No se pierden aristas muy importantes. Las aristas que no son parte del bosque generador son las aristas de retroceso (en DFS) o cruzadas (en BFS). Estas conectan vértices ya visitados, lo que indicaría un ciclo. Estas se excluyen para mantener la estructura de árbol.

$\therefore$ Se cumple.

3) Por demostrar: El tiempo es $O(\mid V \mid + \mid E \mid)$.

La inicialización es $O(\mid V \mid)$ para marcar vértices como no visitados. Luego, en el recorrido de BFS o DFS, cada vértice se procesa una vez (al ser visitado) y además, cada arista se revisa dos veces (una por cada extremo en gráficas no dirigidos). Además, las operaciones por vértice/arista son $O(1)$. Por lo tanto, $O(\mid V \mid + \mid E \mid)$.

$\therefore$ Se cumple.

Por lo tanto, el algoritmo es correcto porque genera un árbol generador por componente conexa (sin ciclos y conexo), procesa todos los vértices y aristas exactamente una vez, y su complejidad es $O(\mid V \mid + \mid E \mid)$. ★

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

Respuesta: Proponemos el siguiente algortimo que mantiene la cola explícitamente (Q) pero en lugar de usar un bucle while, hacemos llamadas recursivas para procesar los elementos de la cola.

Input: Una gráfica conexa G con un vértice distinguido r.

Output: Funciones de parentesco p, nivel ℓ y tiempo de exploración t.

1. $Q \leftarrow [\,]$; Inicializamos la cola vacía donde iremos agregando los nodos por explorar.

2. $i \leftarrow 0$; Iniciamos un contador para el tiempo de cada visita.

3. $i \leftarrow i + 1$; Incrementamos el contador para el tiempo antes de usarlo por primera vez para que $t(r)$ comience en 1.

4. colorear a r de negro ; Marcamos el vértice inicial r como visitado (negro = ya procesado o visitado).

5. añadir r al final de Q ; Ponemos el vértice inicial r en la cola para que sea el primer nodo que la función procese.

6. $t(r) \leftarrow i; p(r) \leftarrow \emptyset; \ell(r) \leftarrow 0$; Para el vértice r, su tiempo de descubrimiento es i, su padre es nulo ($\emptyset$), y su nivel es 0 (pues es la raíz).

7. bfs (Q, G, p, ℓ, color, t, i) ; Recursión donde suponemos han sido inicializadas previamente con el vértice raíz r.

Función bfs recursiva: Esta función recorre una gráfica conexa empezando desde un vértice raíz r para devolver la función parentesco p para saber quién es el padre de cada nodo), el nivel ℓ de cada nodo que es la distancia desde r, y el el tiempo t de descubrimiento, es decir, el orden en que fue visitado cada nodo.

1. if $Q = [\,]$ then ; Este es el caso base: cuando ya no hay más vértices por visitar (la cola está vacía), el algoritmo termina y regresa las estructuras construidas.

2. return (p, ℓ, t) ; Regresa la función de parentesco p, nivel ℓ y tiempo de exploración t.

3. else ; Este es el caso cuando Q no es vacía.

4. $x \leftarrow$ primer elemento de Q ; Elegimos algún vértice de x en Q para mantener la lógica FIFO. Este vértice x debe ser el primero que entró (el más antiguo).

5. $Q \leftarrow Q - \{x\}$; Quitamos al vértice x de la cola.

6. for each vecino y de x en G do ; Exploramos los vecinos de x.

7. if color de $[y] \neq$ negro then ; Revisamos si el vecino y aún no ha sido visitado. Recordemos que el color negro es que ya fue procesado.

8. $i \leftarrow i + 1$; Aumentamos el tiempo en que fue descubierto.

9. color de $[y] \leftarrow$ negro ; Lo marcamos como visitado.

10. $p[y] \leftarrow x$; Ponemos que el padre de y (el vecino de x) es justamente x pues llegamos a y desde x.

11. $t[y] \leftarrow i$; Guardamos el tiempo en que fue descubierto.

12. $\ell[y] \leftarrow \ell[x] + 1$; El nivel de y es el nivel del padre $x + 1$.

13. $Q \leftarrow Q \cup \{y\}$; Añadimos a y a la cola para que también sea procesado más adelante.

14. bfs $(Q, G, p, \ell,$ color, $t, i)$; Esta es la llamada recursiva. Así, el algoritmo continúa con la nueva cola Q, que ahora incluye a los vecinos que fueron recién descubiertos. Con esto quisimos que fuera el equivalente al while $Q \neq [\ ]$ de la versión iterativa, pero ahora que lo estamos implementado con recursión, lo que hacemos es que cada llamada procesa un vértice, actualiza la cola, y vuelve a llamarse si aún hay vértices por explorar. ★

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

   Respuesta: Proponemos el siguiente algoritmo. Queremos usa una función que se llaman a sí mismas, de modo que la información de cada llamada se guarde automáticamente en la pila interna del sistema. Es decir, usamos una pila implícita (no explícita), ya que no la estamos escribiendo como una estructura en el pseudocódigo.

   Input: Gráfica G con un vértice distingido r.

   Output: Funciones de parentesco p, tiempo de entrada f y tiempo de salida ℓ.

   1. $i \leftarrow 0$; Contador de tiempo.
   2. $f \leftarrow \emptyset$; Tiempos de descubrimiento.
   3. $\ell \leftarrow \emptyset$; Tiempos de finalización.
   4. $p \leftarrow \emptyset$; Función de parentesco.
   5. visitado $\leftarrow \emptyset$; Nodos visitados.
   6. dfs(r) ; Recursión al nodo raíz.
   7. return (p, f, ℓ) ; Retorno de resultados.

   Función dfs recursiva: Esta función recorre recursivamente la gráfica desde el nodo v, visitando todos los vértices alcanzables desde v antes de retroceder. Se llama a sí misma para seguir el recorrido y por lo tanto procesar los demás vértices. Cuando dfs(v) se llama, marca el nodo v como visitado y anota el tiempo de entrada, luego recorre sus vecinos, y para cada vecino no visitado, lo marca como hijo ($p(w) = v$) y llama a dfs para w. Después, una vez que ya no hay vecinos nuevos que visitar, marca el tiempo de salida y termina.

   1. $i \leftarrow i + 1$; Incrementamos el contador.
   2. $f[v] \leftarrow i$; Marcamos el tiempo de entrada al nodo v.
   3. visitado $\leftarrow$ visitado$\cup\{v\}$; Marcamos a v como visitado.
   4. for each $w \in$ vecinos de v en G do
   5. if $w \notin$ visitado then
   6. $p[w] \leftarrow v$; Establecemos al padre de w.
   7. dfs(w) ; Llamada recursiva.
   8. $i \leftarrow i + 1$; Incremento final del contador.
   9. ℓ[v]$\leftarrow i$; Marcamos el tiempo de salida del nodo v. ★

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.

   Respuesta: Proponemos el siguiente algoritmo que se detiene cuando encuentra un vértice objetivo t, y devuelve la trayectoria mínima desde r hasta t. Es decir, encuentra el camino más corto desde el vértice r hasta el vértice t en una gráfica, y lo devuelve como una lista de vértices, y si no hay camino entre ellos, devuelve false.

   Input: Una gráfica conexa G con un vértice distinguido r y objetivo t.

   Output: Camino mínimo desde r hasta t, o `false` si no existe.

   1. $Q \leftarrow [\,]$; Inicializamos la cola vacía donde iremos agregando los nodos por explorar.

   2. $i \leftarrow 0$; Iniciamos un contador para el tiempo de cada visita.

   3. $i \leftarrow i+1$; Incrementamos el contador antes de usarlo por primera vez para que $t(r)$ comience en 1.

   4. colorear a r de negro ; Marcamos el vértice inicial r como visitado.

   5. añadir r al final de Q ; Ponemos el vértice inicial r en la cola.

   6. $t(r) \leftarrow i; p(r) \leftarrow \emptyset; \ell(r) \leftarrow 0$; Inicializamos sus atributos.

   7. while $Q \neq [\,]$ do

   8. $\quad x \leftarrow$ primer elemento de Q ; Sacamos la cabeza de la cola.

   9. $\quad Q \leftarrow Q - \{x\}$; Eliminamos a x de la cola para ya no trabajar con el.

   10. $\quad$ if $x = t$ then ; Verificamos si el nodo que estamos procesando es el objetivo t. Si es así, ya encontrado el nodo que queríamos y ahora podemos construir el camino más corto desde el nodo inicial r hasta t.

   11. $\quad\quad$ camino $\leftarrow [\,]$; Creamos una lista vacía que contendrá el camino desde r hasta t.

   12. $\quad\quad$ actual $\leftarrow t$; Empezamos a construir el camino desde el nodo objetivo t, retrocediendo hacia r usando los padres con p.

   13. $\quad\quad$ while actual $\neq r$ do ; Mientras no lleguemos al nodo inicial r, seguimos retrocediendo.

   14. $\quad\quad\quad$ insertar actual al inicio de camino ; Como estamos retrocediendo desde t hasta r, y queremos que el camino esté en orden correcto (de r a t), agregamos el nodo actual al principio del camino.

15. actual $\leftarrow$ p[actual] ; Saltamos al padre del nodo actual. Esto nos permite retroceder un paso en el camino.

16. insertar r al inicio de camino ; Ahora, ya que el ciclo se detuvo en el nodo inicial r, lo agregamos manualmente al inicio para que el camino esté completo.

17. return camino ; Devolvemos el camino completo desde r hasta t, como una lista de nodos.

18. for each vecino y de x en G do; Si no hemos encontrado a t, entonces seguimos procesando. Esta línea inicia el bucle donde exploramos a los vecinos de x, es decir, los nodos conectados directamente a x.

19. if color[y] $\neq$ negro then

20. $i \leftarrow i + 1$

21. color[y] $\leftarrow$ negro

22. añadir y al final de Q

23. $t(y) \leftarrow i;\ p(y) \leftarrow x;\ \ell(y) \leftarrow \ell(x) + 1$

24. return false ; Si se vació Q sin encontrar a t. ★