Algoritmos de búsqueda en grafos I

Dr. Eduardo A. Rodríguez Tello

Cinvestav-Tamaulipas

7 de febrero de 2018

1. Representación computacional de grafos

2. Algoritmos de búsqueda en grafos

Representación computacional de grafos

Matriz de adyacencia

- La forma más natural para representar grafos en una computadora es la *matriz de adyacencia*
- Sea $V = \{v_1 \dots v_{|V|}\}$ el conjunto de vértices del grafo G, y E el conjunto de sus aristas. La matriz de adyacencia será una matriz de tamaño $|V| \times |V|$, donde la entrada (i, j) será

$$a_{i,j} = \begin{cases} 1 & \text{si existe una arista de } v_i \text{ a } v_j \\ 0 & \text{en otro caso} \end{cases}$$

- En el caso de grafos no dirigidos, la matriz es simétrica.

Eficiencia

- Verificar la existencia de una arista se realiza en tiempo constante.
- Sin embargo, se requiere $O(|V|^2)$ espacio de almacenamiento.

Representación computacional de grafos

Lista de adyacencia

- Otra representación posible es la *lista de adyacencia*.
- Se trata de un arreglo de $|V|$ listas enlazadas, una para cada vértice. La lista correspondiente al vértice u contiene todos los vértices v para los cuales $(u, v) \in E$.
- Así, cada arista aparecerá una vez (considerando todas las listas) si el grafo es dirigido, o dos si el grafo es no dirigido.

Eficiencia

- El tamaño de esta estructura de datos es $O(|V| + |E|)$, lo que siempre es menor o igual que la matriz de adyacencia.
- Verificar la existencia de una arista ya no requiere tiempo constante; sin embargo, esta estructura es más eficiente para encontrar todos los vecinos de cada vértice.

Representación computacional de grafos

Matriz vs. lista de adyacencia

- Pero ¿cuál de las dos representaciones es más apta, o eficiente?
- Depende de la aplicación, pero sobretodo del número de aristas (esto es, si se trata de un grafo *denso* o *disperso*)
- Por ejemplo, si deseamos almacenar un grafo completo, o casi completo, implicaría búsquedas costosas en la lista de adyacencia.
- Por otro lado, imaginemos que deseamos almacenar el grafo de la *web*, donde una arista significa que hay un enlace de un sitio al otro.
- Este grafo es disperso, porque en promedio, cada sitio tiene enlaces a unos seis sitios más, de las millones de opciones disponibles.

Búsqueda en profundidad (DFS)

- El primer problema que trataremos de resolver es saber qué partes del grafo son alcanzables desde un cierto vértice.
- Si suponemos que tenemos el grafo almacenado en forma de una lista de adyacencia, la única información que nos brinda de manera directa, es el conjunto de vecinos de un vértice.
- Para saber si un vértice es alcanzable, necesitaremos cierta memoria adicional: primero, para saber si ya visitamos cierto vértice (para no viajar en círculos); y también para poder regresar por donde llegamos (como en un laberinto).
- La manera de saber si ya visitamos un vértice será mediante un arreglo booleano. La cuerda para regresar será con una pila (simulada, mediante recursividad).

Búsqueda en profundidad (DFS)

En el siguiente procedimiento las rutinas previsita y posvisita son operaciones realizadas cuando se llega por primera vez a un vértice, y cuando se abandona para no regresar, respectivamente. Las definiremos a detalle más adelante.

Procedure explorar(G, v)

Require: Un grafo $G = (V, E)$; un vértice $v \in V$

```
1: visitado[v] = true
2: previsita(v)
3: for cada arista (v, u) ∈ E do
4:     if ¬visitado[u] then
5:         explorar(G, u)
6:     end if
7: end for
8: posvisita(v)
9: return visitado[]      // visitado[u] = true si u es alcanzable desde v
```

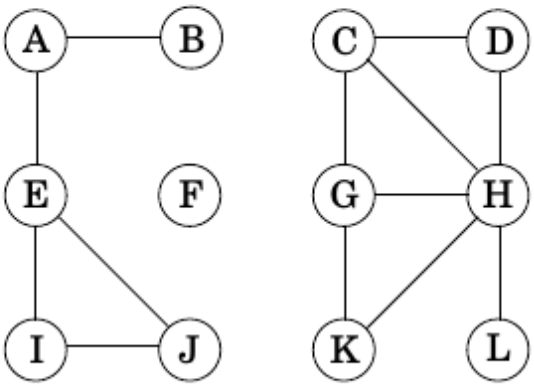
Búsqueda en profundidad (DFS)

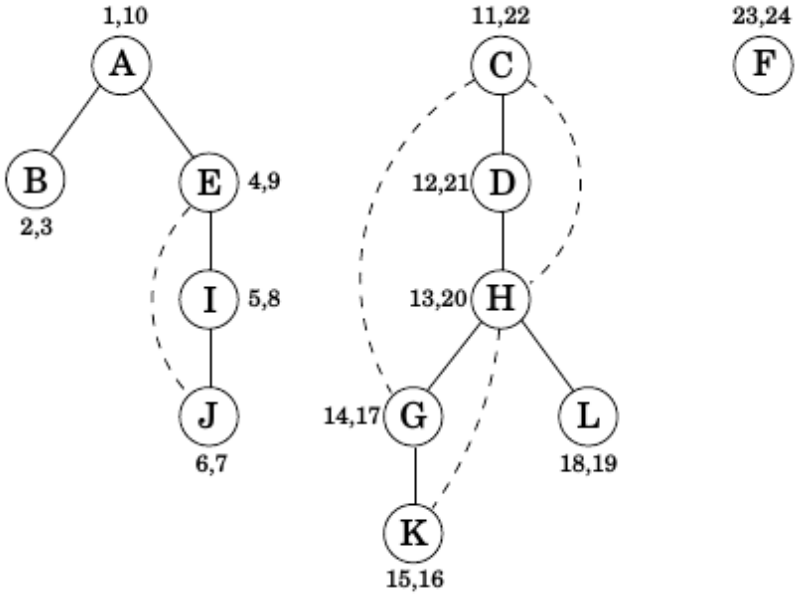
El procedimiento explorar sólo encuentra la parte de grafo que es alcanzable desde un punto de partida. Para examinar el resto del grafo, es necesario reiniciar el procedimiento desde los vértices que no se han examinado.

Procedure $\mathrm{dfs}(G)$

1: **for** cada vértice $v \in V$ **do**
2: $\quad visitado[v] = \textbf{false}$
3: **end for**
4: **for** cada vértice $v \in V$ **do**
5: $\quad$ **if** $\neg visitado[v]$ **then**
6: $\quad\quad \mathrm{explorar}(v)$
7: $\quad$ **end if**
8: **end for**

Búsqueda en profundidad (DFS)





- 3 llamadas a $\mathrm{explorar}(\cdot)$, con A, C y F (3 árboles)
- Arcos explorados con $\mathrm{explorar}(\cdot)$ son lineas sólidas, aristas del árbol (*tree edges*)
- Arcos que llevan a vértices ya visitados son líneas punteadas, aristas reversa (*back edges*)

Análisis del algoritmo de búsqueda en profundidad

- La búsqueda en profundidad consiste en explorar todos y cada uno de los vértices (gracias al arreglo $visitado[]$). Debemos entonces analizar el tiempo que toma la exploración.
- La exploración realiza las siguientes tareas:
  1. Marcar el vértice como visitado y efectuar la previsita y la posvisita. Esto se hace en tiempo constante.
  2. Un ciclo en el cual se examinan los vértices adyacentes, probando si llevan a un lugar nuevo.
- El tiempo total invertido en el paso 1 es $O(|V|)$, porque se hace para todos los vértices.
- En el paso 2, cada arista (v, u) se examina dos veces, lo que toma $O(|E|)$.

Análisis del algoritmo de búsqueda en profundidad

- Por lo tanto el tiempo de toda la búsqueda es, entonces, $O(|V| + |E|)$,
- ¡tan eficiente como simplemente recorrer la lista de adyacencia!

Componentes conexas de un grafo

- Como vimos, la búsqueda en profundidad crea un árbol por cada componente conexa del grafo. Así, valiéndonos del procedimiento previsita, podemos saber qué vértices pertenecen a cada componente.

Procedure previsita(v)

1: $numcc[v] = cc$

- Donde cc es global y debe ser incrementada en uno, cada vez que dfs llame al procedimiento explorar.

Registro de visitas

- Otro uso que podemos dar a los procedimientos $\mathrm{previsita}$ y $\mathrm{posvisita}$, es el registro de los momentos en el que un nuevo vértice es descubierto por la búsqueda, y en el momento en el que dicho vértice se abandona durante la misma.

Procedure $\mathrm{previsita}(v)$

$pre[v] = clock$
$clock = clock + 1$

Procedure $\mathrm{posvisita}(v)$

$post[v] = clock$
$clock = clock + 1$

Registro de visitas

Propiedad:

Para cualquier par de vértices u y v, los intervalos $[pre(u), post(u)]$ y $[pre(v), post(v)]$ son, ya sea disjuntos, o uno está contenido en el otro.

Registro de visitas

Propiedad:

Para cualquier par de vértices u y v, los intervalos $[pre(u), post(u)]$ y $[pre(v), post(v)]$ son, ya sea disjuntos, o uno está contenido en el otro.

- ¿Por qué?

Registro de visitas

Propiedad:

Para cualquier par de vértices u y v, los intervalos $[pre(u), post(u)]$ y $[pre(v), post(v)]$ son, ya sea disjuntos, o uno está contenido en el otro.

- ¿Por qué?
- Porque $[pre(u), post(u)]$ es el tiempo durante el cual un vértice u está en la pila. El comportamiento "el último en entrar, es el primero en salir" de la pila explica el resto

Ciclos en grafos dirigidos

- El procedimiento para la búsqueda de primero en profundidad funciona sin modificación en los grafos dirigidos, únicamente hay que tomar en cuenta el par ordenado que representa la arista.
- Afortunadamente, con este procedimiento tan eficiente, podremos extraer información del grafo dirigido que no es tan evidente como en los grafos no dirigidos.
- Empezaremos con la siguiente propiedad.

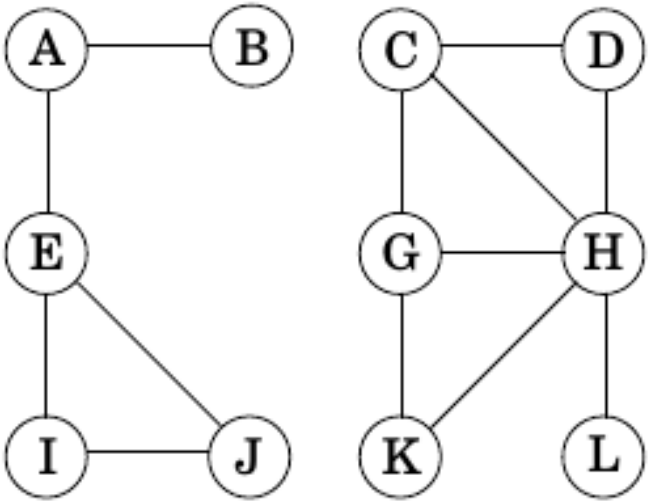
Ciclos en grafos dirigidos

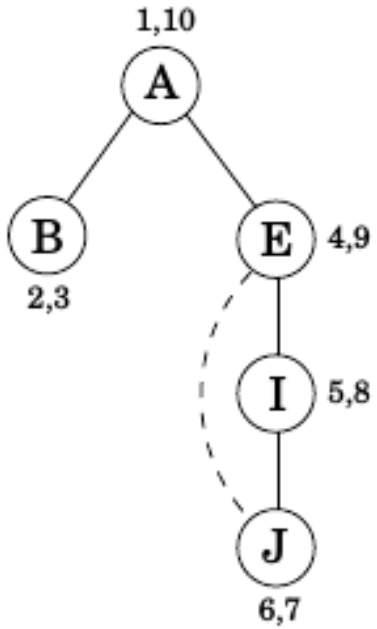
Propiedad:

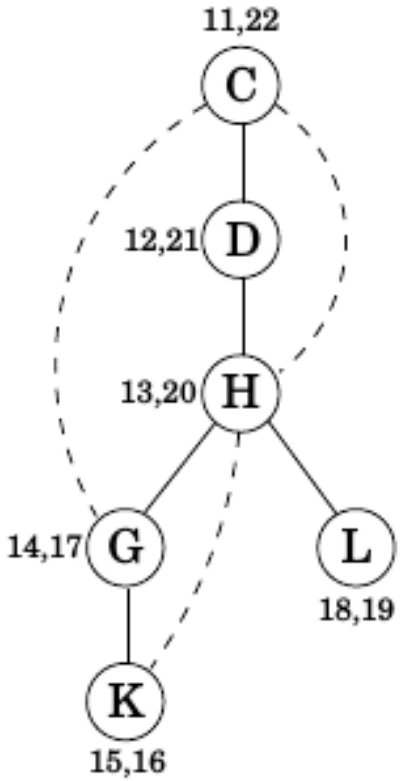
Un grafo dirigido tiene un ciclo si y sólo si su búsqueda en profundidad revela una arista reversa.

- Una arista reversa es la que va de un vértice a alguno de sus ancestros en el árbol generado por la búsqueda
- Tal arista es fácil de identificar: si existe una arista (u, v) en la que los vértices cumplan $pre[v] < pre[u] < post[u] < post[v]$

Ciclos en grafos dirigidos







- $pre[v] < pre[u] < post[u] < post[v]$

Grafos dirigidos acíclicos (GDAS)

- Mediante la propiedad anterior, podemos verificar si el grafo es *acíclico* en tiempo lineal. Estos grafos son muy importantes porque modelan relaciones de jerarquías o de temporalidad.
- Por ejemplo, si cada tarea se representa con un vértice, una arista de u a v se puede usar si u es una pre-condición de v. Así, el orden de tareas será realizable si no hay ciclos.
- Si el grafo es un GDA, entonces será posible *linealizar* (u *ordenar topológicamente*) de manera que nos permita realizar las tareas conservando el orden necesario.
- La tarea de linealizar es simple, hay que ordenar las tareas en orden decreciente de sus números $post$.

Fuentes y pozos

- Un vértice es *fuente* si no tiene aristas de entrada. Inversamente, un vértice es *pozo* si no tiene aristas de salida.

Propiedad:

Todo GDA tiene al menos una fuente y un pozo.

- De esta manera, al linealizar un GDA, el primer vértice debe ser una fuente, y el último vértice debe ser un pozo.

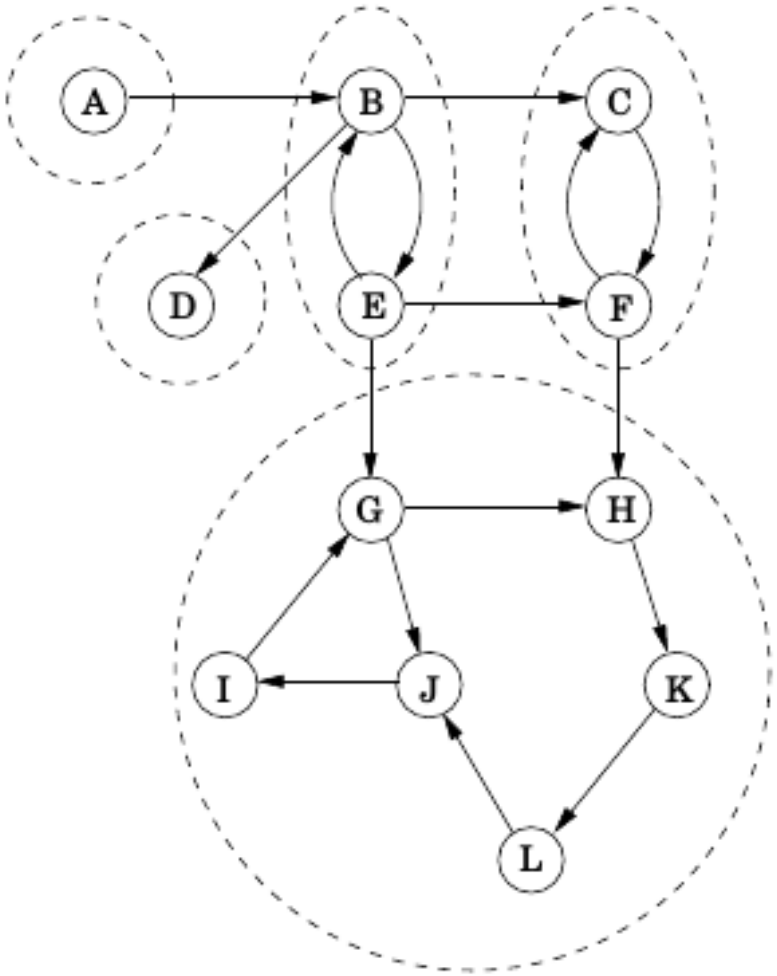
Conectividad en grafos dirigidos

Definición

Dos vértices u y v de un grafo dirigido son conexos si existe un camino de u a v *y* un camino de v a u.

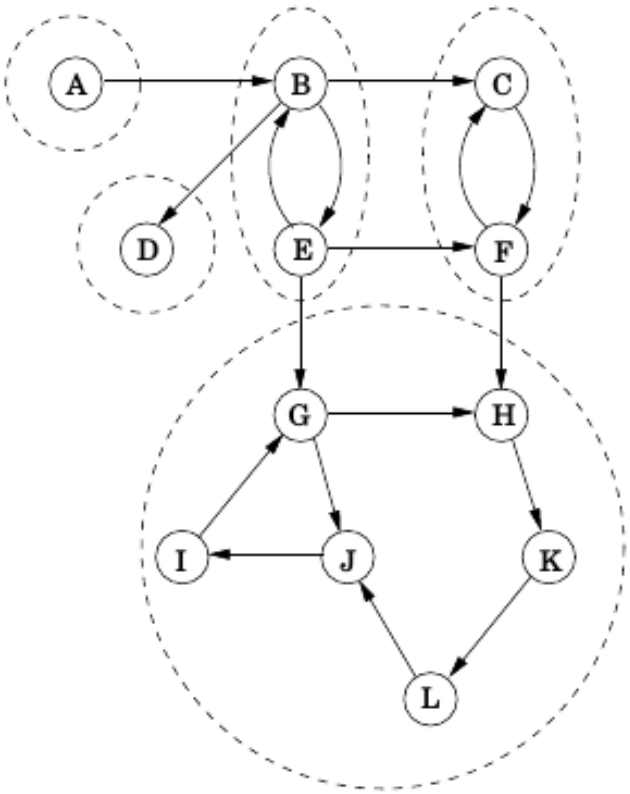
- Utilizando esta definición, podemos dividir V en conjuntos disjuntos de vértices alcanzables entre ellos bidireccionalmente; a tales conjuntos los llamaremos *componentes fuertemente conexos*.

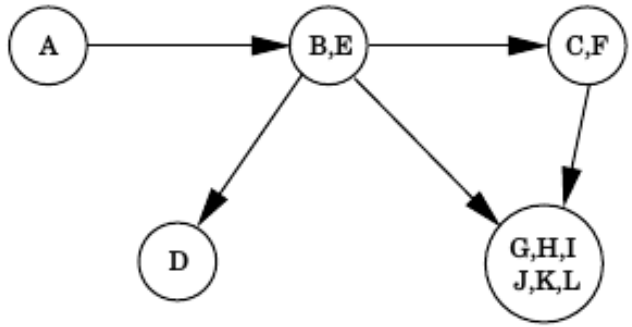
Conectividad en grafos dirigidos



Conectividad en grafos dirigidos

- Ahora, si reducimos cada componente fuertemente conexo a un único meta-vértice, y dibujamos una arista de un meta-vértice a otro si hay una arista (en la misma dirección) entre sus respectivos componentes, obtendremos un *meta-grafo*, que además será un GDA.





Conectividad en grafos dirigidos

- Como podemos imaginar, identificar los componentes fuertemente conexos puede ser muy útil, pero necesitamos un algoritmo eficiente que lleve esto a cabo
- A continuación lo estudiaremos

Identificación de componentes fuertemente conexos

- El algoritmo DFS, junto con algunas propiedades, nos ayudarán a encontrar una algoritmo eficiente.

Propiedad:

Si comenzamos el procedimiento explorar en un vértice u, entonces su ejecución terminará precisamente cuando todos los vértices alcanzables desde u hayan sido visitados.

- Dada esta primera propiedad, para que el procedimiento explorar obtenga exactamente un componente conexo, tal componente debe ser pozo (un componente es pozo si es representado por un vértice pozo en el meta-grafo).
- El problema ahora es identificar un vértice que forme parte de un componente pozo.

Identificación de componentes fuertemente conexos

Propiedad:

El vértice que es asignado con el mayor número $post$ en una búsqueda de primero en profundidad debe ser parte de un componente fuente.

- Esta propiedad nos ayuda a encontrar un vértice en componente fuente,
- ¡pero necesitamos un vértice en un componente pozo!
- Para esto definiremos el grafo *revés* G^R, que es el mismo grafo G pero con todas sus aristas en dirección opuesta. Tal grafo tiene los mismos componentes fuertemente conexos que el original.

Identificación de componentes fuertemente conexos

- Así, al hacer una búsqueda de primero en profundidad en G^R, el vértice con el mayor número $post$ estará en un componente fuente en G^R, o sea, un componente pozo en G.
- Esto resuelve nuestro problema, y es posible aislar un componente pozo. ¿Cómo proceder con los demás?

Propiedad:

Si C y C' son componentes fuertemente conexos, y hay una arista de un vértice en C a un vértice en C', entonces el mayor número $post$ en C es más grande que el mayor número $post$ en C'.

- Usando esta propiedad, si eliminamos el componente fuertemente conexo identificado previamente, el siguiente vértice con el número $post$ más alto pertenecerá a un componente pozo del grafo restante.

Identificación de componentes fuertemente conexos...

- El algoritmo, finalmente, sólo requiere tiempo lineal, y queda de la siguiente manera:

Procedure identificarComponentesFuertementeConexos($G = (V, E)$)

1: Ejecutar dfs en G^R.
2: Ejecutar dfs para identificar los componentes conexos no dirigidos en G, procesando los vértices en orden decreciente de los números $post$ identificados en el paso 1.