AED: Grafos 1

gabriel.spranger@utec.edu.pe
macarena.oyague@utec.edu.pe

Mayo 2022

Índice

1. Algoritmos de Búsqueda

Cada algoritmo de búsqueda tiene ciertas propiedades que revelan información importante sobre el grafo. Elegir el algoritmo de búsqueda que necesitamos depende del problema que tratemos de resolver. Por ello es bueno saber las ventajas de los algoritmos de búsqueda más conocidos para que cuando veamos el problema que tratemos de resolver, sabremos qué algoritmo se adecúa más a la situación.

1.1. Breadth-First Search

BFS es un algoritmo que empieza en un nodo origen s, visita todos los nodos alcanzables desde s a distancia 1, luego a distancia 2, etc, hasta que se hayan visitado todos los nodos alcanzables desde s. Notar que por distancia nos referimos a número de aristas.

Al terminar, BFS produce un árbol que tiene el camino de menor número de aristas desde el nodo origen s hasta cualquier nodo v alcanzable desde s, dado que primero visita los nodos a distancia 1 de s, luego lo que están a distancia 2 y así sucesivamente.

Listing 1: Pseudocódigo de BFS para grafos dirigidos o no dirigidos.

```
BFS(s):
1.  s.distance = 0
2.  q = queue()
3.  q.push(s)
4.  while q.notEmpty():
5.      u = q.pop()
6.      for v in adj[u]:
7.          if v.visited == false:
8.              v.visited = true
9.              v.distance = u.distance + 1
10.             v.predecessor = u
11.             q.push(v)
```

- El atributo `distance` que se le pone a cada vértice v indica el número de aristas en el camino más corto desde s hasta v.
- El atributo `visited` sirve para que cada nodo sea visitado como máximo una sola vez.
- El atributo `predecessor` sirve para que si queremos obtener el camino mínimo entre un vértice v hasta s, basta con llamar recursivamente `v = v.predecessor` mientras $v \neq s$, pero cuidado porque si $v == null$ antes de que $v == s$, significa que no hay camino entre s y v. Esto se podría dar porque s está en un componente del grafo y v está en otro (ambos componentes son disjuntos).

1.1.1. Complejidad

Todas las operaciones de BFS corren en $O(1)$ excepto por los bucles de las líneas 4 y 6. El bucle de la línea 4 corre exactamente $O(V)$ veces, dado que cada vértice es encolado como máximo una vez y desencolado posteriormente. El bucle de la línea 6 corre en su totalidad para cada vértice, es decir, se recorren todas las aristas de cada vértice del grafo, esto hace que el bucle de la línea 6 corra un total de $O(E)$ veces (ya que se recorren todas las aristas de cada vértice, por lo tanto se recorren todas las aristas del grafo):

$$\sum_{v \in V} |adj[v]| = O(E)$$

Por lo tanto, el tiempo de ejecución de BFS es $O(V) + O(E) = O(V + E)$. Por esto se puede decir que BFS corre en tiempo lineal en relación al número de vértices y aristas del grafo.

Si quieren ver una prueba matemática que demuestra la correctitud de BFS, pueden revisar a partir de la página 598 del Cormen.

1.1.2. Propiedades

- **Distancias:** Parte de un vértice s y va visitando los vértices a distancia 1 de s, luego los que están a distancia 2 de s, luego los que están a distancia 3 y así sucesivamente hasta que no hayan más nodos que visitar desde s (pueden seguir habiendo nodos en el grafo si hay otro componente).
- **Shortest-Path Tree:** Genera un árbol (gracias al atributo `predecessor`) que permite saber el camino mínimo de aristas desde el vértice s hasta cualquier nodo v alcanzable desde s.

1.2. Depth-First Search

DFS empieza en un vértice cualquiera y explora las aristas del vértice más recientemente descubierto hasta que ese vértice no tiene más aristas que visitar, en dicho caso, DFS "regresa" en la recursión hasta que encuentra un nodo que sigue teniendo aristas que visitar y el proceso sigue hasta que todos los nodos del grafo han sido visitados. Cabe resaltar que a diferencia de BFS, DFS sí visita todos los nodos del grafo, sea el grafo disconexo o no.

DFS puede producir múltiples árboles (un bosque) a diferencia de BFS que solo produce uno, ya que DFS puede empezar desde distintos vértices. Si se llegaron a visitar todos los vértices desde un origen, pero siguen quedando vértices sin visitar en el grafo, entonces DFS empieza la búsqueda nuevamente desde uno de esos vértices sin visitar.

Listing 2: Pseudocódigo de DFS para grafos dirigidos o no dirigidos.

```
DFS():
1.   for v in vertices:
2.     if v.color == white:
3.       DFS–VISIT(v)

DFS–VISIT(v):
1.     time += 1
2.     v.d = time
3.     v.color = gray
4.     for u in adj[v]:
5.         if u.color == white:
6.             u.predecessor = v
7.             DFS–VISIT(u)
8.     v.color = black
9.     time += 1
10.    v.f = time
```

- El atributo `color` revela bastante información sobre al grafo como veremos a continuación. Por ahora, la aplicación más importante es que sirve como un *flag* para marcar un nodo como no visitado (*white*), visitado pero aun faltan aristas suyas por visitar (*gray*) y visitado completamente (*black*).
- El atributo `predecessor` sirve un propósito similar al de BFS: construir un árbol a partir del recorrido de DFS. Sin embargo, como el grafo puede estar particionado en componentes, podría no generarse un solo árbol sino un bosque. Cada raíz del bosque es el nodo donde se llamó a DFS-VISIT en la línea 3 de DFS.
- Los atributos `d` y `f` guardan el "tiempo" o la "iteración" en la cual se descubrió y visitó completamente al vértice, respectivamente.

1.2.1. Complejidad

El `for` de la línea 1 de DFS corre en $O(V)$ sin importar la complejidad de cada llamada de DFS-VISIT. En cuanto a DFS-VISIT, se llama una sola vez por cada vértice, ya que para que se llame a DFS-VISIT sobre un vértice v, v debe ser *white* (se asume que todos los nodos empiezan de color *white* y el `for` de la línea 1 de DFS se asegura que se llame al menos una vez por cada nodo del grafo) y apenas se llama a DFS-VISIT se pinta a v de color *gray* (y por lo tanto nunca se hará otra llamada a DFS-VISIT con v). Dado que ya sabemos que DFS-VISIT se llama una sola vez por cada vértice v y dado que en cada llamada se itera sobre todas las aristas de v, en total, DFS-VISIT itera sobre todas las aristas del grafo, es decir, tiene un tiempo de ejecución total de $O(E)$ (todo el trabajo extra que hace DFS-VISIT es constante). Por lo tanto, DFS corre en $O(V) + O(E) = O(V + E)$.

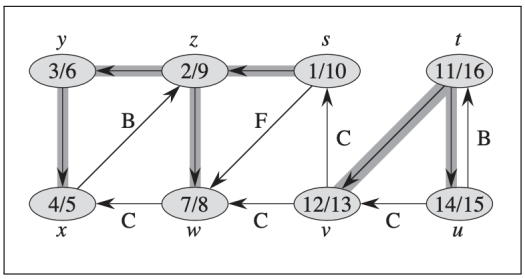

Figura 1: Un grafo con los tiempos de inicio (`d`) y final (`f`) en formato (`d/f`).

1.2.2. Propiedades

- **Estructura de paréntesis:** Si al correr DFS hacemos que se imprima un paréntesis izquierdo "(" cuando pintamos un vértice *gray* y un paréntesis derecho ")" cuando pintamos de *black*, entonces veremos que la impresión final es una expresión correcta de paréntesis.

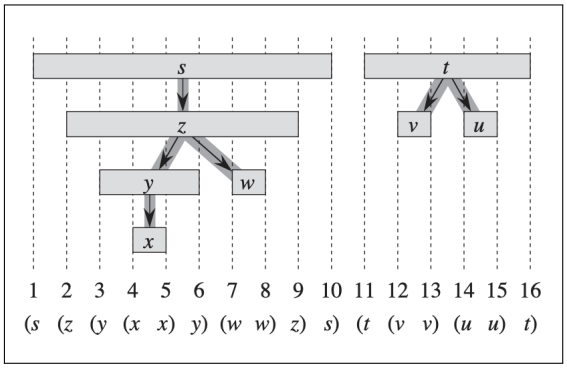

Figura 2: Estructura correcta de paréntesis generada al correr DFS sobre este grafo. Notar que en este caso se imprime el paréntesis junto con el nombre del vértice al cual le corresponde.

- **Componentes:** Para grafos no dirigidos, DFS puede ser usado para descubrir los componentes conexos de un grafo. Dado que una llamada a DFS-VISIT desde v visita todos los nodos alcanzables desde v, basta con contar el número de veces que se llama a DFS-VISIT en la línea 3 de DFS para obtener el número de componentes conexos del grafo. Para grafos dirigidos, se usa a DFS como una subrutina para hallar los componentes fuertemente conexos de un grafo mediante algoritmos como el de Kosaraju y el de Tarjan. Un componente fuertemente conexo de un grafo es un subgrafo dirigido donde hay un camino entre cualquier par de vértices dentro de ese subgrafo.

- **Clasificación de aristas:** La arista más simple que DFS genera es el **tree edge**, el cual es generado cada vez que se fija el `predecessor` de un nodo en la línea 6 de DFS-VISIT. Estos **tree edges** corresponden a las aristas sombreadas de la siguiente imagen:

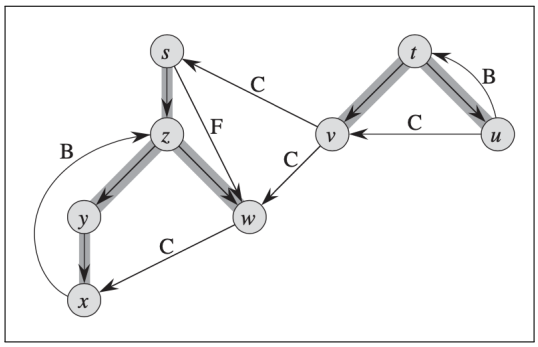

Figura 3: Bosque generado al correr DFS en este grafo.

La otra arista que DFS genera son los **back edges**. Estos son los que tienen la letra **B**. Lo interesante de estas aristas es que indican si hay ciclos en el grafo. Podemos identificar estas aristas en la línea 4 de DFS-VISIT. Si el nodo u que estamos revisando es de color *gray*, significa que en el camino de nodos visitados que estamos trazando con DFS-VISIT hay una arista que lleva a un nodo de ese camino que ya hemos visitado, y por lo tanto es un ciclo.

El otro tipo de arista es el **forward edge**, representados por la letra **F**. Estas aristas representan caminos alternativos desde un nodo s hasta un nodo w que DFS-VISIT no tomó en cuenta dado el orden de evaluación de las aristas de s (en el ejemplo, se empezó por z antes que w). Se identifican si al iterar sobre los vecinos de un nodo u, encontramos un nodo v *black* cuyo tiempo de descubrimiento `d` sea mayor al de u. Esto es porque si el tiempo de descubrimiento de v es mayor al de u y además u y v están conectados por una arista (el **forward edge**), significa que v es un descendiente de u (los **forward edges** van de ancestro a descendiente en el bosque de DFS).

Finalmente, el último tipo de arista es el **cross edge** identificado con la letra **C**. Son las aristas que no son **tree**, **back** ni **forward**.

Los **forward** y **cross** solo aplican para grafos dirigidos.

- **Topological Sort:** Podríamos ver un grafo dirigido como un grafo de "precedencias", es decir, si existe una arista (u, v), entonces lo que sea que represente u tiene que venir antes que v. En otras palabras, podemos ver al grafo dirigido como uno que representa "un orden" de algo.

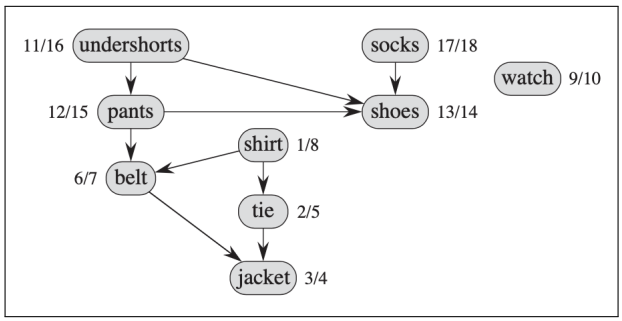

Figura 4: Un DAG (*Directed Acyclic Graph*) que representa dependencias entre las prendas de vestir que uno se pone. Por ejemplo, *pants* obviamente tiene que estar antes que *belt*, mientras que no importa el orden de *watch* y tampoco si uno se pone primero *socks* o *undershorts*.

Claramente, este orden no está definido para grafos dirigidos cíclicos, dado que se estaría contradiciendo que un vértice tiene que venir antes de otro vértice.

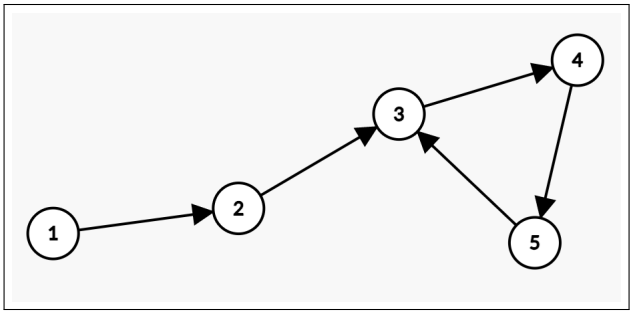

Figura 5: Un grafo dirigido con ciclos. Si lo interpretamos como un grafo que representa un orden, vemos que hay una contradicción. Estamos diciendo que 3 va antes de 4 y que 4 va antes de 5, pero también que 5 va antes que 3, lo cual es una contradicción.

Dado esto, un **topological sort** de un DAG es una lista de vértices de tal manera que si existe la arista (u, v), entonces u está antes que v en la lista. Por lo tanto, si dibujamos el DAG a partir de la lista generada por un **topological sort**, veremos que todas las aristas van de izquierda a derecha. El algortimo es el siguiente:

Listing 3: Pseudocódigo de topological sort.

```
TopoSort(G):
1. Llamar a DFS sobre G, pero cada vez
   que se marque un nodo como negro,
```

```
        hacerle pushFront a una listaEnlazada
    2.  Devolver la listaEnlazada
```

1.3. Dijkstra

Dijkstra es un algoritmo que construye un **shortest path tree** que dado un nodo origen s, permite hallar los caminos con el mínimo peso posible hasta cualquier nodo $v \in V - \{s\}$. Funciona para grafos con pesos positivos (un algoritmo que funciona para cualquier peso es Bellman-Ford). La idea detrás de Dijkstra es que va construyendo un subconjunto de vértices S que inicialmente empieza solo con el nodo origen s. En cada iteración de Dijkstra, se elige un vértice v que está más cerca a S, se actualizan las distancias de los los u que están conectados con v y se sigue iterando hasta que $S = V$. Como siempre se elige el vértice **a menor distancia** de S, se dice que Dijkstra es un **algoritmo greedy** o voraz.

Listing 4: Pseudocódigo de Dijkstra.

```
Dijkstra(G, s):
1.    for v in G.vertices:
2.        v.d = INT_MAX
3.        v.predecessor = null
4.    s.d = 0
5.    visitados = list()
6.    pendientes = min_heap(vertices)
7.    while pendientes.notEmpty():
8.      u = pendientes.extract_min()
9.      visitados.push(u)
10.     for v in adj[u]:
11.       if v.d > u.d + w(u, v):
12.         v.d = u.d + w(u, v)
```

- El atributo `d` de un vértice $v \in V - \{s\}$ será siempre mayor o igual al peso del camino mínimo desde el vértice origen s hasta v.
- El atributo `predecessor` sirve para que cuando se quiera consultar al **shortest path tree** el camino mínimo entre el vértice origen s y un nodo cualquiera v, basta con empezar en v y seguir el atributo `predecessor` hasta que lleguemos a s.
- En las líneas 1-4 hacemos la inicialización necesaria: el peso total del camino mínimo entre s y todos los demás vértices del grafo es ∞ dado que es inicialmente desconocido; los predecesores todavía no se tienen que fijar; como s es el nodo origen, el peso total del camino mínimo desde s hasta s es 0.

- `pendientes` es un *min heap* que inicialmente tiene a todos los vértices del grafo y está indexado por el atributo `d`. Sirve para hallar el nodo que esté más cerca al conjunto de nodos `visitados` (S).
- `visitados` es una lista que tiene los nodos cuyo atributo `d` ya ha sido fijado y es el más mínimo posible, por lo tanto nunca va a cambiar.
- Notar que siempre se cumple que: $V = \text{visitados} \cup \text{pendientes}$.
- La línea 10-12 realiza un proceso que se llama **relajación**; lo que significa es que se verifica si se pueden mejorar las distancias `d` de los vecinos de u.

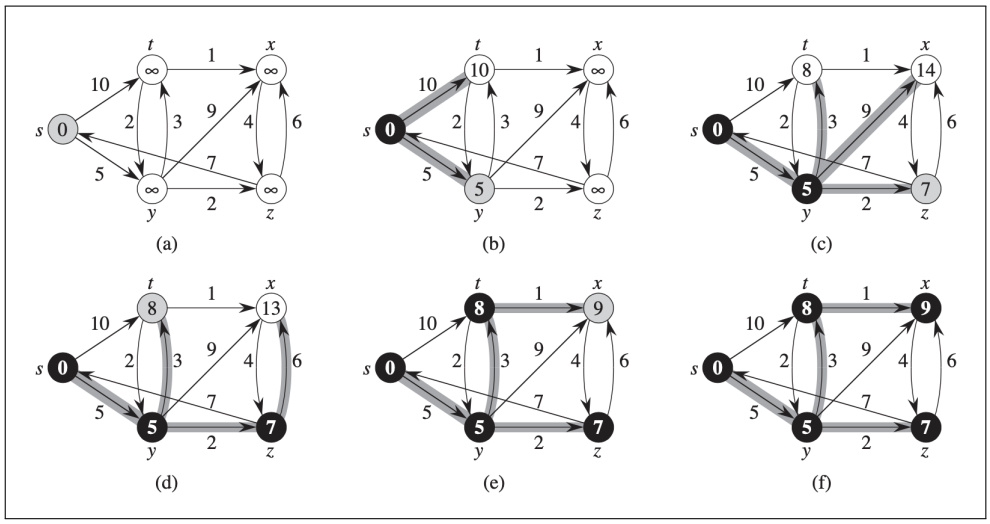

Figura 6: Ilustración del algoritmo Dijkstra. **(a)** muestra el grafo antes de correr Dijkstra tomando como vértice origen a s. **(b)-(f)** muestran el estado del grafo luego de cada iteración del `while` de la línea 7. **Notar** que el número que cada vértice tiene es el valor del atributo `d`. **Notar** que las aristas y los vértices sombreados representan el **shortest path tree** resultante.

1.3.1. Complejidad

- El `for` de la línea 1 toma $O(V)$.
- La línea 6 es un BUILD-MAX-HEAP implícito, por lo tanto corre en $O(V)$.
- El `while` de la línea 7 corre en $O(V)$ dado que primero el *min heap* tiene todos los vértices del grafo y en cada iteración del `while` se hace un solo EXTRACT-MIN. Por lo tanto, el `while` correrá en $O(V)$.
- La línea 8 corre en $O(\lg n)$, ya que es un EXTRACT-MIN.

- El `for` de la línea 10 corre para cada vértice y como se iteran sobre las aristas **de cada** vértice, el `for` de la línea 10 corre **en total** $O(E)$.
- La línea 12 es un DECREASE-KEY implícito, por lo tanto corre en $O(\lg n)$.

Por lo tanto, tenemos que la complajidad $T(V, E)$ de Dijkstra es:

$$O(V) + O(V) + O(V \lg V) + O(E \lg V)$$

$$= O((V + E) \lg V)$$

Esa es la complejidad asumiendo que usamos un *min heap*. Sin embargo, si usamos un *Fibonacci Heap*, cuyo DECREASE-KEY corre en $O(1)$ **amortizado**, tendremos que Dijkstra corre en:

$$O(V \lg V + E)$$

1.3.2. Propiedades

- Produce un **shortest path tree** enraizado en el nodo origen s que permite hacer consultar sobre el camino de peso mínimo entre cualquier vértice v y s.

2. Referencias

- Todas las secciones fueron hechas basadas en el material del libro *Introduction to Algorithms (Cormen)*, Capítulo 22: https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf