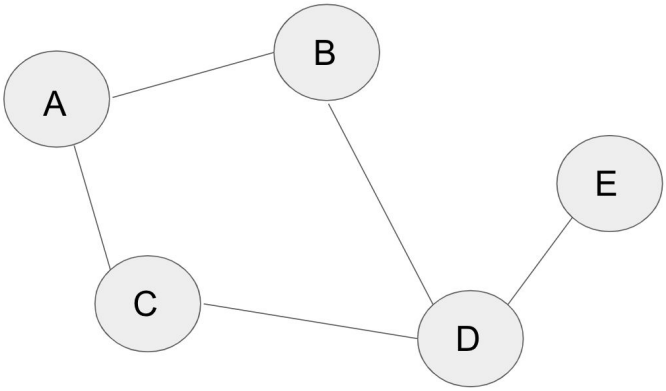
Grafos

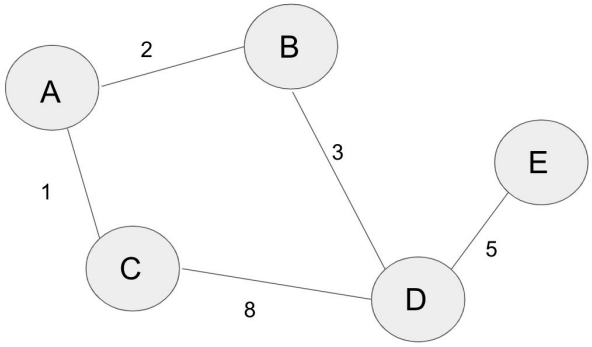
¿Qué es un grafo?

Conjunto de nodos (o vértices) y ejes (o aristas). Cada eje conecta 2 nodos.

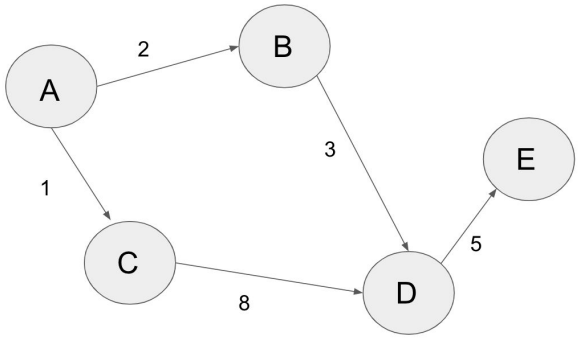


Características de grafos

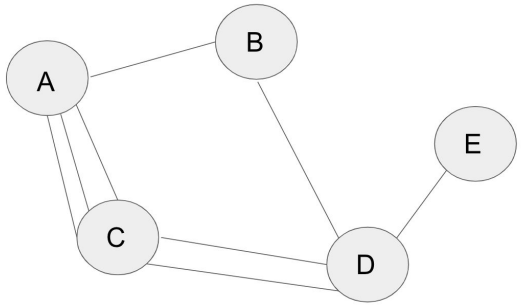
- Cada eje puede tener un valor asociado, lo que lo convierte en un grafo **con pesos.**



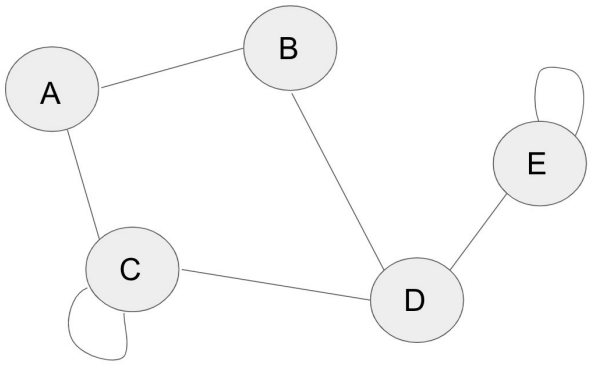
- Si los ejes pueden recorrerse en un único sentido, se representan con una flecha en vez de una línea y el grafo se llama **digrafo** o **grafo dirigido.**



- Si entre dos nodos puede haber más de un eje, el grafo se llama **multigrafo.**



- Cuando un eje conecta a un nodo con sí mismo, se llama un **lazo.**



Casos de uso

- Redes sociales (Facebook, Twitter, etc.)
    - Facebook: Una amistad va para ambos lados, si cada nodo representa a un usuario y se conectan los nodos que tienen amistad con un eje, se podría representar con un grafo simple sin pesos no dirigido.
    - Twitter (o instagram): Se puede seguir a una persona pero no ser seguido. Si cada nodo es un usuario y un eje representa que un usuario sigue a otro, se podría representar con un grafo simple sin pesos **dirigido.**
- Transporte (aviones, trenes, rutas, etc.):
    - Un sistema de aviones entre ciudades podría representarse tomando cada aeropuerto como un nodo y cada ruta con un eje. Se podría tomar el peso de los ejes como el precio del trayecto. Un mismo trayecto entre dos ciudades puede ser recorrido por varias aerolíneas distintas. Para representar este sistema puede usarse un **multigrafo dirigido con pesos.**
- Dependencias de proyectos de software (maven, gradle, npm, etc)
    - Los sistemas de dependencias deben instalar versiones específicas de cada librería. Cada librería a su vez puede depender de muchas otras, y varias librerías pueden tener una dependencia en común (y a veces, distintas librerías requieren versiones distintas de la misma dependencia). Estos sistemas usan

grafos especializados para lograr descargar la versión correcta de cada librería necesaria.

Formas de representación

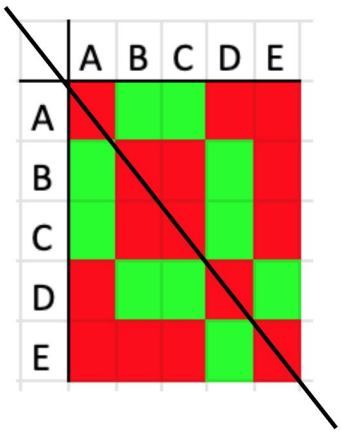
¿Cómo hacemos para representar un grafo en código? Hay varias opciones:

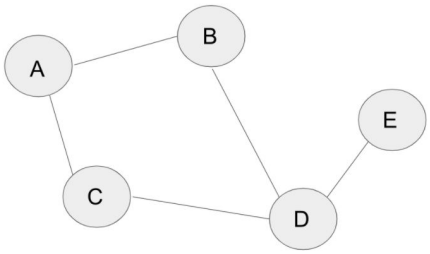
- Matriz de adyacencia
- Matriz de incidencia
- Lista de adyacencia
- Conjunto de nodos y ejes

Matriz de adyacencia

Se le asigna un índice a cada nodo que le corresponde a una fila y una columna de una matriz cuadrada. Si el nodo *i* y el nodo *j* tienen un eje en común, se representa en la posición *i, j* de la matriz.

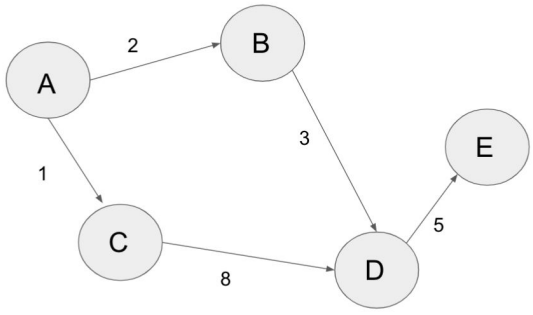
- Si el grafo no es dirigido, la matriz es simétrica:





- Si los grafos tienen peso, se marcan en la matriz:

	A	B	C	D	E
A	0	2	1	0	0
B	0	0	0	3	0
C	0	0	0	8	0
D	0	0	0	0	5
E	0	0	0	0	0



- Si el grafo tiene muchos ejes por cada nodo, se dice que el grafo es **denso** y la matriz aprovecha bien su espacio.

- Si el grafo tiene pocos ejes por cada nodo (**sparse** en inglés), la mayoría de las celdas de la matriz quedan vacías, por lo que se desperdicia mucho espacio.
- Agregar o leer ejes es muy simple.
- Agregar nodos no es eficiente, ya que hay que recrear la matriz para hacer espacio (existen posibles optimizaciones).

Matriz de incidencia

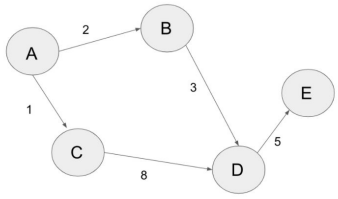
Cada fila representa un nodo y cada columna un eje. En la posición [*i,j*] se indica si el eje j conecta al nodo *i*

- Cada columna contiene solo dos valores, por lo que no es muy eficiente en espacio.

Lista de adyacencia

Los nodos se guardan en una lista, y cada uno contiene a su vez otra lista con sus nodos adyacentes:

A	B	C	D	E
B - 2	D - 3	D - 8	E - 5	
C - 1				



Conjunto de nodos y ejes

Basado en la idea de una lista de adyacencia, se pueden crear clases *Node* y *Edge* que representen un nodo y un eje. *Node* tendría una lista de *Edge* y su identificador y *Edge* tendría el nodo al que apunta y su costo en caso de ser un grafo con pesos. Los nodos se pueden guardar en un mapa donde la key sea el identificador de cada nodo (En los ejemplos que vimos sería una letra).

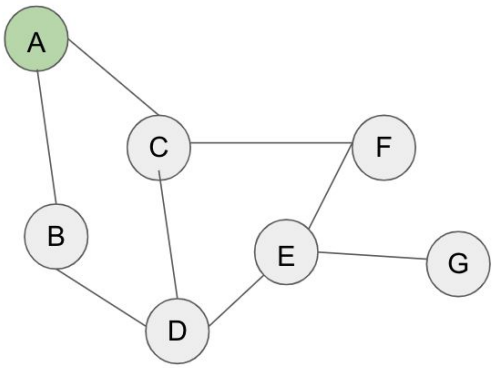
Formas de recorrer un grafo

Formas hay muchas, pero algunas de ellas son:

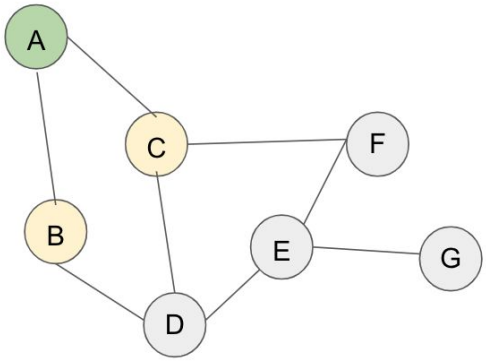
- BFS
- DFS
- Dijkstra

BFS

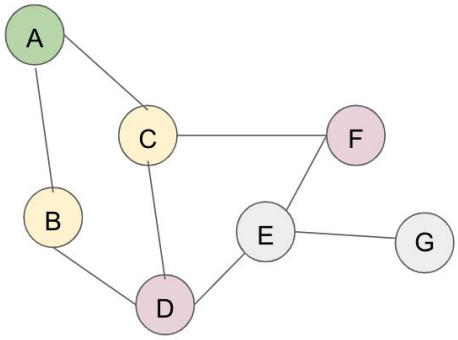
Se recorre el grafo por niveles. Se empieza marcando el nodo inicial y luego en cada paso se recorren y marcan los nodos no marcados que son adyacentes a un grafo marcado. Por ejemplo, arrancando en A:



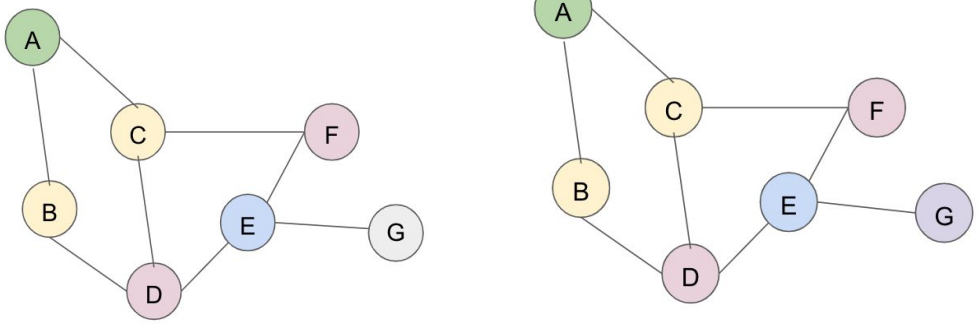
El siguiente paso sería marcar B y C, ya que son los nodos que cumplen la condición mencionada:



En el siguiente paso los nodos que cumplen la condición son D y F:



Y en 2 pasos más se recorre primero E y después G:



->

En código, esto se puede representar con una cola. La cola mantiene los nodos que quedan por visitar y por cada paso se toma un nodo y se añaden a la cola todos sus vecinos que no están marcados:

```
void printBfs(String startingLabel) {
  unmarkAllNodes();

  Queue<Node> nodesToVisit = new LinkedList<>();
  nodesToVisit.add(nodes.get(startingLabel));

  while (!nodesToVisit.isEmpty()) {
    Node current = nodesToVisit.remove();
    if (!current.marked) {
      current.mark();
      System.out.println(current.label);
      for (Node edgeNode : current.edges) {
        if (!edgeNode.marked) {
          nodesToVisit.add(edgeNode);
        }
      }
    }
  }
}
```

Haciendo un seguimiento del grafo anterior por este código, la cola empieza únicamente con **A.**

- En el primer paso se elimina A de la cola y como A no está marcado se lo marca y se añaden B Y C a la cola. Quedaría el estado de la cola **B-C.**
- En el siguiente paso se elimina B de la cola y como no está marcado se mira a sus vecinos, A y D. Como A ya está marcado, se ignora, pero se agrega D a la cola, que queda **C-D.**
- En el siguiente paso se elimina C de la cola, sus vecinos son A, D y F. A se ignora por estar marcado, pero se agregan D y F a la cola. Nótese que si bien D ya estaba en la cola todavía no fue procesado, así que se agrega igual. El estado queda **D-D-F.**
- En el siguiente paso se elimina D de la cola y se lo marca, que tiene 3 vecinos, B, C y E. B y C ya están marcados, pero se agrega E a la cola, que queda **D-F-E.**
- En el siguiente paso se elimina D de la cola de vuelta, pero como ya está marcado no se hace nada, por lo que la cola queda **F-E.**

- El resto de los pasos quedan como ejercicio para el lector.

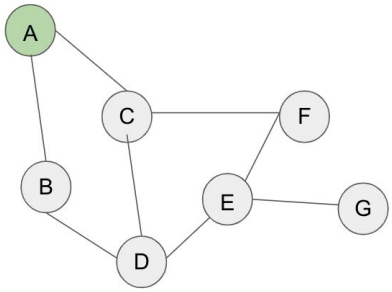
La complejidad de este algoritmo es **O(N + E)** siendo N la cantidad de nodos y E la cantidad de ejes. Esto es porque cada nodo se recorre y marca una única vez y cada eje se recorre 2 veces (1 por cada nodo que conecta).

DFS

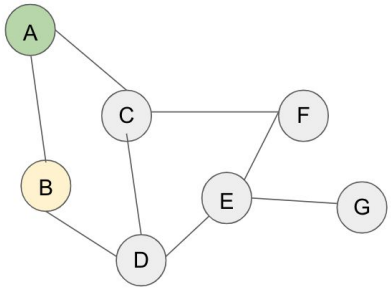
Se recorre el grafo por profundidad. Se siguen los siguientes pasos, empezando por el nodo inicial:

- Dado un nodo, se marca como visitado y elige uno de los vecinos no visitados al azar.
- Nos paramos en ese nodo vecino y hacemos lo mismo que en el paso anterior.
- Después de terminar de visitar un vecino en profundidad, visitamos el resto de ellos de igual forma.

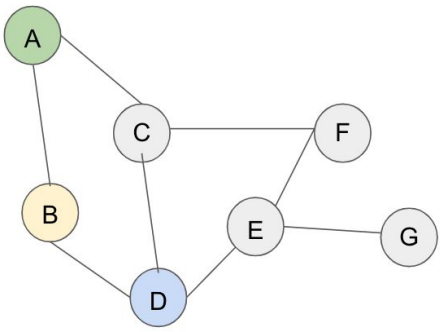
Por ejemplo, arrancando en A:



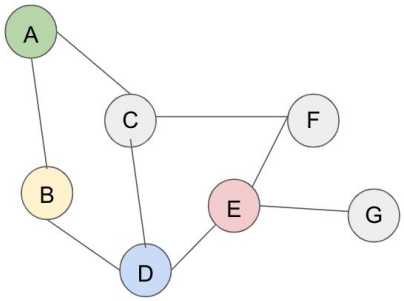
Se podría seguir por B o por C, primero se elige B:



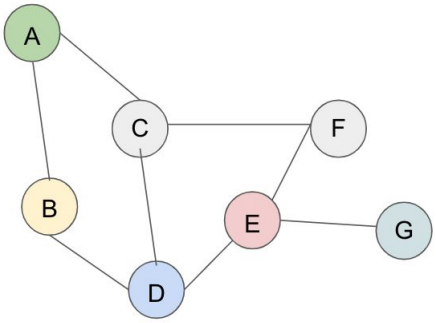
Luego, como A está marcado como visitado, la única opción es seguir por D:



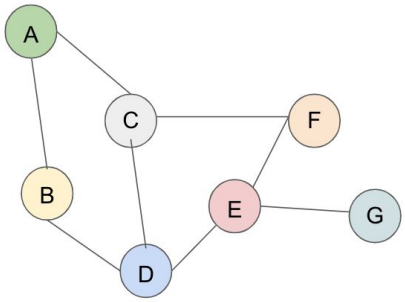
Aca de vuelta hay 2 opciones, C o E. Se elige de forma aleatoria, por ejemplo la E:



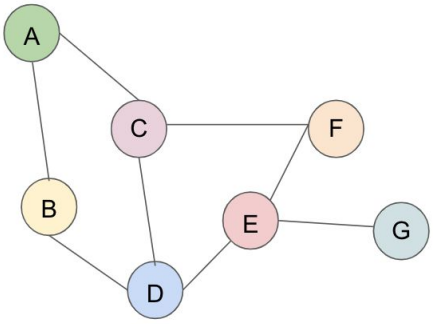
En el siguiente paso se podría elegir F o G, en nuestro ejemplo la G:



Ahora que la G no tiene ningún otro vecino, volvemos a E y se elige el vecino restante no marcado, la F:



Finalmente, desde F solo se puede visitar C y terminamos de marcar todo el grafo:



En código, esto se puede representar con una pila. La forma más fácil de representar la pila es con una función recursiva, pero se puede hacer también de forma iterativa haciendo uso de la clase *Stack.* En cada paso de la función recursiva se recibe un nodo, se le aplica una función (en este caso imprimir el valor del nodo) y se llama la función recursivamente con todos los vecinos no marcados.

```
private void printDfs(Node node) {
  if (node.marked) {
    return;
  }
  node.mark();
  System.out.println(node.label);

  for (Node edgeNode : node.edges) {
    if (!edgeNode.marked) {
      printDfs(edgeNode);
    }
  }
}
```

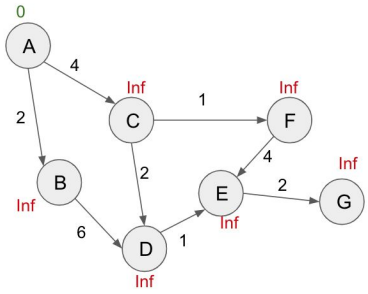
La complejidad de este algoritmo es **O(N + E)** siendo N la cantidad de nodos y E la cantidad de ejes. Esto es porque cada nodo se recorre y marca una única vez y cada eje se recorre 2 veces (1 por cada nodo que conecta).

Dijkstra

El objetivo de Dijkstra es encontrar el camino más corto entre nodos de un grafo cuyos ejes tengan pesos no negativos. Si se tienen ejes con pesos negativos, la alternativa es el algoritmo de Bellman-Ford, la desventaja de este último es que tiene un peor orden de complejidad.

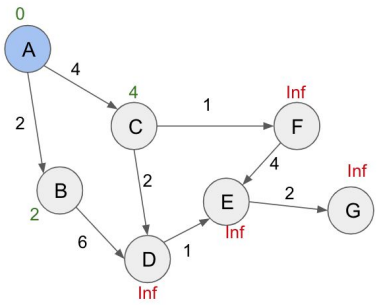
Los pasos del algoritmo de Dijkstra son:

- Marcar todos los nodos con infinito excepto el nodo inicial (en este caso la A es el nodo inicial):

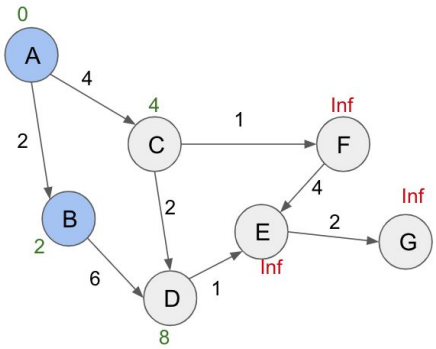


  Este número representa el menor costo conocido de llegar hasta ese nodo hasta este momento.
- En cada paso, tomamos el nodo con menor costo que aún no haya sido visitado y actualizamos el costo de sus vecinos de ser necesario con el costo del nodo + el costo del eje que los conecta.

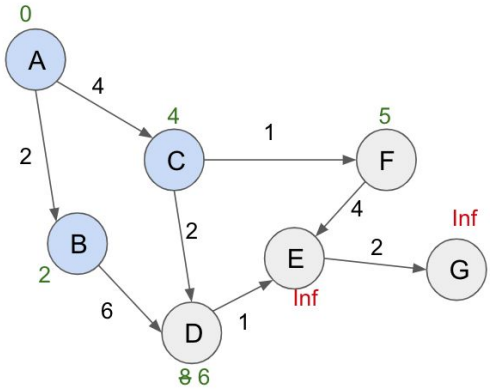
Hacemos un seguimiento del algoritmo en el grafo propuesto, empezamos en la A y actualizamos el valor de sus vecinos (B y C), ya que encontramos un camino más corto hacia ellos:



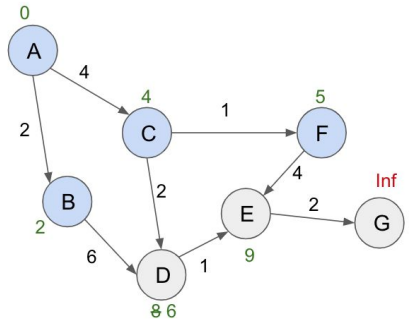
En el siguiente paso, el nodo con menor costo es B (costo 2) por lo que nos paramos en ese nodo y actualizamos D con 2 + 6 = **8**



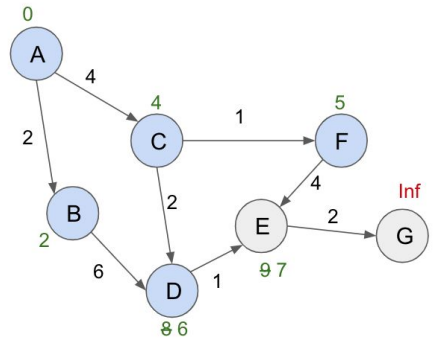
En el siguiente paso, el nodo con menor costo es C con 4. Desde allí, encontramos un mejor camino hacia D, antes el mejor era 8 pero ahora encontramos un camino con costo 4 + 2 = **6**, por lo que actualizamos su valor (también actualizamos F):



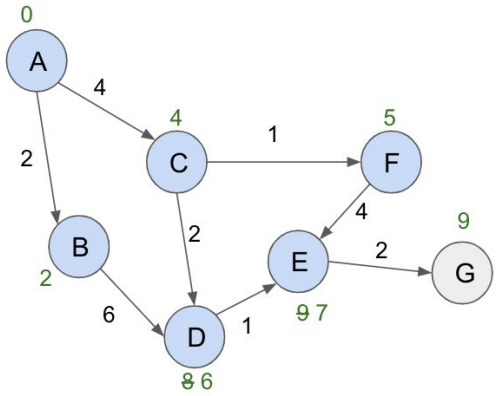
En este paso, el nodo con menor costo es F, por lo que lo marcamos y actualizamos E:



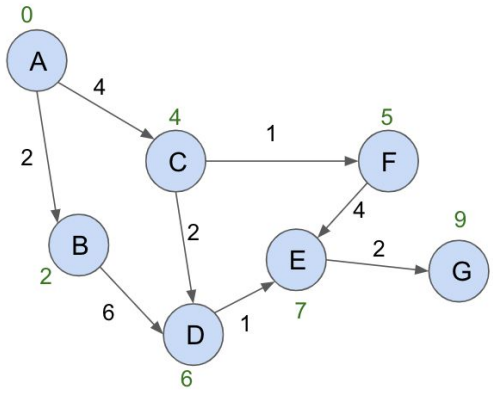
Ahora nos paramos en D y pasa lo mismo que antes, encontramos un mejor camino hasta E:



Finalmente, llegamos a G y terminamos de recorrer todo el grafo:



->



El grafo final nos muestra cual es el camino más corto partiendo desde A hacia todo el resto. Por ejemplo:

Dijkstra(A, G) = 9
Dijkstra(A, D) = 6
Dijkstra(A, F) = 5

Este algoritmo funciona porque cuando nos paramos en un nodo sabemos que no es posible encontrar un camino más corto hacia el, ya que es el nodo con menor costo conocido. Si hubiera otro nodo con costo menor nos hubiéramos parado en ese otro.

Para implementar este algoritmo, puede resultarnos útil hacer uso de una **PriorityQueue.** Esta es una cola que tiene de diferencia que siempre devuelve el nodo con menor valor (en vez de el primero que haya sido insertado). Por ejemplo:

- Insertamos **4, 6, 2, 8**.
- Una cola normal los devolvería en ese mismo orden.
- Una Priority Queue devuelve **2, 4, 6, 8**.

Los métodos add y remove tienen complejidad O(log(N))
Para los detalles de la API se puede ver la documentación oficial de la clase:
https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

Para implementar este algoritmo, a la implementación que tenemos de grafo con pesos debemos agregarle un campo **cost** a la clase **Node** que debe ser numérico (int, long, double, etc) y una clase auxiliar **PqNode** que usaremos para usar de nodo en la PriorityQueue:

```java
class PqNode implements Comparable<PqNode> {
  Node node;
  double cost;

  public PqNode(Node node, double cost) {
    this.node = node;
    this.cost = cost;
  }

  @Override
  public int compareTo(PqNode other) { return Double.compare(cost, other.cost); }
}
```

El campo **cost** de la clase Node representa el menor costo posible para llegar hasta ese nodo con los ejes recorridos hasta ese momento y el campo **cost** de PqNode representa el camino de un recorrido en particular. Podría haber más de un PqNode con el mismo nodo, pero se visitará primero el que tenga menor **cost**. Por eso es importante marcar un nodo al visitarlo y no volver a analizarlo si ya ha sido visitado:

```java
public void printDijkstra(String startingLabel) {
  // Poner en false el campo marked de todos los nodos.
  unmarkAllNodes();
  // Poner el costo inicial de todos los nodos en infinito.
  nodes.values().forEach(node -> node.cost = Double.MAX_VALUE);

  // Iniciar la PriorityQueue con el primer nodo con costo 0.
  Node startingNode = nodes.get(startingLabel);
  startingNode.cost = 0;
  PriorityQueue<PqNode> queue = new PriorityQueue<>();
  queue.add(new PqNode(startingNode,  cost: 0));

  while (!queue.isEmpty()) {
    PqNode pqNode = queue.remove();
    // Nos aseguramos de no analizar un nodo dos veces verificando
    // que no esté marcado y marcándolo inmediatamente.
    if (pqNode.node.marked) continue;
    pqNode.node.mark();
    System.out.println(pqNode.node.label + ": " + pqNode.cost);

    for (Edge edge : pqNode.node.edges) {
      // Para cada vecino, lo agregamos a la PQ solo si el camino
      // que encontramos es el menor encontrado hasta el momento.
      double targetNodeCost = pqNode.cost + edge.weight;
      if (targetNodeCost < edge.targetNode.cost) {
        edge.targetNode.cost = targetNodeCost;
        queue.add(new PqNode(edge.targetNode, targetNodeCost));
      }
    }
  }
}
```

La complejidad de este algoritmo es **O((N + E) * log(N))** siendo N la cantidad de nodos y E la cantidad de ejes. La E es porque cada eje se recorre 2 veces (1 por cada nodo que conecta) y N surge de que cada nodo se recorre una única vez. Los log(N) sumados a ambas letras salen de que cada vez que se analiza un nodo hay que hacer un **remove** de la PriorityQueue que tiene costo log(N) y los **add** de la PQ se terminan haciendo E veces. Existen formas de

modificar el algoritmo para que la complejidad en el peor caso termine siendo **O(N * log(N) + E)** lo cual es una mejora para grafos densos especialmente.