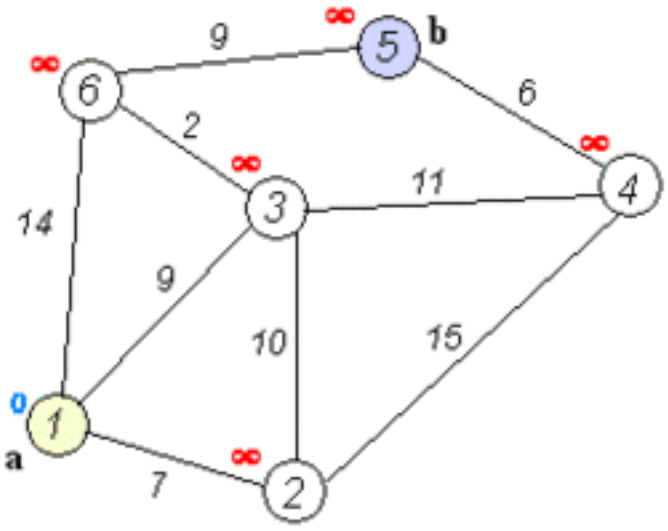
CAMINO MAS CORTO: ALGORITMO DE DIJKSTRA

Descripción
El algoritmo de dijkstra determina la ruta más corta desde un nodo origen hacia los demás nodos para ello es requerido como entrada un grafo cuyas aristas posean pesos. Algunas consideraciones:

- Si los pesos de mis aristas son de valor 1, entonces bastará con usar el algoritmo de BFS.
- Si los pesos de mis aristas son negativos no puedo usar el algoritmo de dijsktra, para pesos negativos tenemos otro algoritmo llamado Algoritmo de Bellmand-Ford.

Como trabaja
Primero marcamos todos los vértices como no utilizados. El algoritmo parte de un vértice origen que será ingresado, a partir de ese vértices evaluaremos sus adyacentes, como dijkstra usa una técnica greedy – *La técnica greedy utiliza el principio de que para que un camino sea óptimo, todos los caminos que contiene también deben ser óptimos*- entre todos los vértices adyacentes, buscamos el que esté más cerca de nuestro punto origen, lo tomamos como punto intermedio y vemos si podemos llegar más rápido a través de este vértice a los demás. Después escogemos al siguiente más cercano (con las distancias ya actualizadas) y repetimos el proceso. Esto lo hacemos hasta que el vértice no utilizado más cercano sea nuestro destino. Al proceso de actualizar las distancias tomando como punto intermedio al nuevo vértice se le conoce como **relajación (*relaxation*).**
Dijkstra es muy similar a BFS, si recordamos BFS usaba una Cola para el recorrido para el caso de Dijkstra usaremos una Cola de Prioridad o Heap, este Heap debe tener la propiedad de Min-Heap es decir cada vez que extraiga un elemento del Heap me debe devolver el de menor valor, en nuestro caso dicho valor será el peso acumulado en los nodos.
Tanto java como C++ cuentan con una cola de prioridad ambos implementan un Binary Heap aunque con un Fibonacci Heap la complejidad de dijkstra se reduce haciéndolo mas eficiente, pero en un concurso mas vale usar la librería que intentar programar una nueva estructura como un Fibonacci Heap, claro que particularmente uno puede investigar y programarlo para saber como funciona internamente.

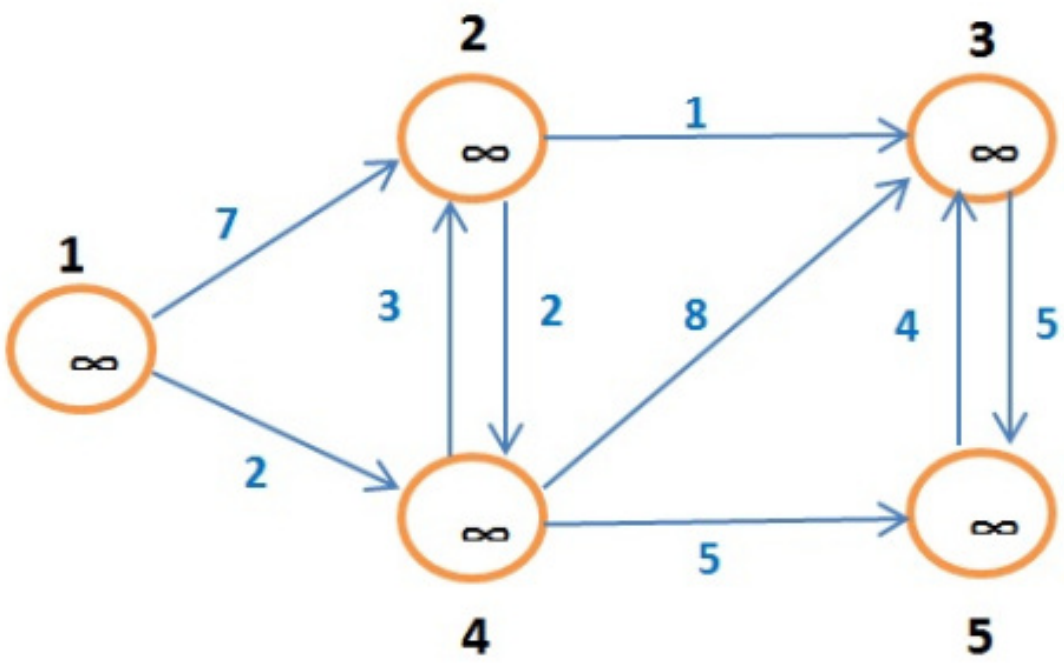


Algoritmo en pseudocódigo
Considerar distancia[i] como la distancia mas corta del vértice origen ingresado al vértice i.

```
método Dijkstra(Grafo,origen):
    creamos una cola de prioridad Q
    agregamos origen a la cola de prioridad Q
    mientras Q no este vacío:
        sacamos un elemento de la cola Q llamado u
        si u ya fue visitado continuo sacando elementos de Q
        marcamos como visitado u
        para cada vértice v adyacente a u en el Grafo:
            sea w el peso entre vértices ( u , v )
            si v no ah sido visitado:
               Relajacion( u , v , w )

método Relajacion( actual , adyacente , peso ):
    si distancia[ actual ] + peso < distancia[ adyacente ]
       distancia[ adyacente ] = distancia[ actual ] + peso
       agregamos adyacente a la cola de prioridad Q
```

Ejemplo y Código paso a paso
Tengamos el siguiente grafo, cuyos ID están en color negro encima de cada vértice, los pesos esta en color azul y la distancia inicial en cada vértice es infinito



Algunas consideraciones para entender el código que se explicara junto con el funcionamiento del algoritmo.

```
#define MAX 10005 //maximo numero de vértices
#define Node pair< int , int > //definimos el nodo como un par( first , second ) donde
```

```
#define INF 1<<30 //definimos un valor grande que represente la distancia infinita inic
```

Inicializamos los valores de nuestros arreglos

Vértices	1	2	3	4	5
Distancia[u]	∞	∞	∞	∞	∞
Visitado[u]	0	0	0	0	0
Previo[u]	-1	-1	-1	-1	-1

Donde:

- **Vértices:** ID de los vértices.
- **Distancia[u]:** Distancia mas corta desde vértice inicial a vértice con ID = u.
- **Visitado[u]:** 0 si el vértice con ID = u no fue visitado y 1 si ya fue visitado.
- **Previo[u]:** Almacenara el ID del vértice anterior al vértice con ID = u, me servirá para impresión del camino mas corto.

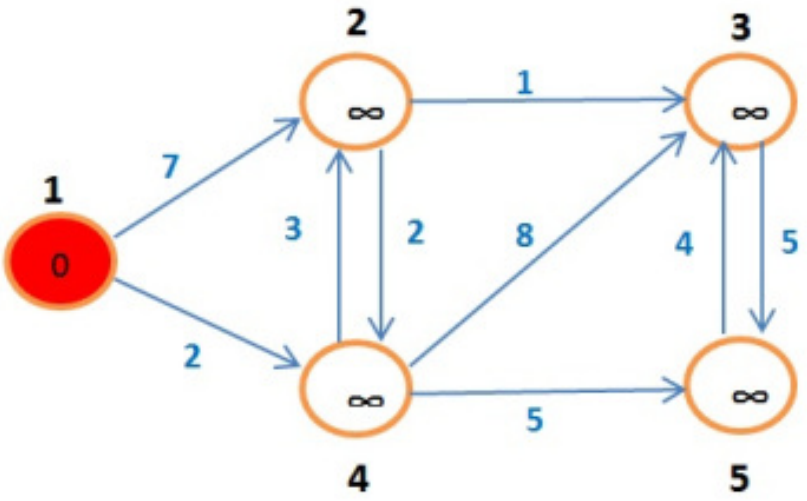
En cuanto al código los declaramos de la siguiente manera:

```
vector< Node > ady[ MAX ]; //lista de adyacencia
int distancia[ MAX ];      //distancia[ u ] distancia de vértice inicial
                             a vértice con ID = u
bool visitado[ MAX ];      //para vértices visitados
int previo[ MAX ];         //para la impresion de caminos
priority_queue< Node , vector<Node> , cmp > Q; //priority queue
propia del c++, usamos el comparador definido para que el
de menor valor este en el tope
int V;                     //numero de vertices
```

Y la función de inicialización será simplemente lo siguiente

```
//función de inicialización
void init(){
    for( int i = 0 ; i <= V ; ++i ){
        distancia[ i ] = INF;  //inicializamos todas las
                                distancias con valor infinito
        visitado[ i ] = false; //inicializamos todos los
              vértices como no visitado
        previo[ i ] = -1;      //inicializamos el previo del vértice i con -1
    }
}
```

De acuerdo al vértice inicial que elijamos cambiara la distancia inicial, por ejemplo la ruta más corta partiendo del vértice 1 a todos los demás vértices:



Cola	
Nodo	Peso
1	0

El vértice 1 es visitado, la distancia de vértice 1 -> vértice 1 es 0 por estar en el mismo lugar.

```
1  Q.push( Node( inicial , 0 ) ); //Insertamos el vértice inicial
   en la Cola de Prioridad
2  distancia[ inicial ] = 0;      //Este paso es importante,
   inicializamos la distancia del inicial como 0
```

Extraemos el tope de la cola de prioridad

```
1  while( !Q.empty() ){              //Mientras cola no este vacia
2          actual = Q.top().first;//Obtengo de la cola el nodo con menor peso,
   en un comienzo será el inicial
3          Q.pop();    //Sacamos el elemento de la cola
```

Si el tope ya fue visitado entonces no tengo necesidad de evaluarlo, por ello continuaría extrayendo elementos dela cola:

```
1  if( visitado[ actual ] ) continue; //Si el vértice actual ya fue
   visitado entonces sigo sacando elementos de la cola
```

En este caso al ser el tope el inicial no esta visitado por lo tanto marcamos como visitado.

```
1  visitado[ actual ] = true;          //Marco como visitado el vértice actual
```

Hasta este momento la tabla quedaría de la siguiente manera

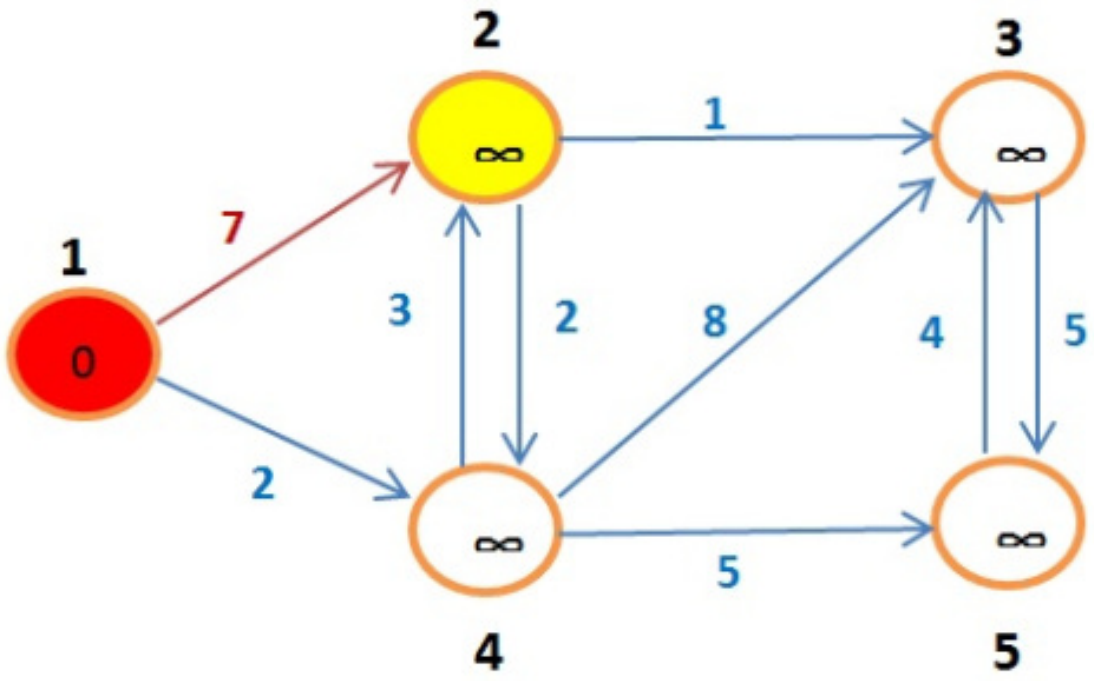
Vértices	1	2	3	4	5
Distancia[u]	0	∞	∞	∞	∞
Visitado[u]	1	0	0	0	0
Previo[u]	-1	-1	-1	-1	-1

Ahora vemos sus adyacentes que no hayan sido visitados. Tendríamos 2 y 4.

```
1  for( int i = 0 ; i < ady[ actual ].size() ; ++i ){ //reviso sus adyacentes
   del vertice actual
2      adyacente = ady[ actual ][ i ].first;   //id del vertice adyacente
3      peso = ady[ actual ][ i ].second;        //peso de la arista que une
```

```
4    actual con adyacente ( actual , adyacente )
        if( !visitado[ adyacente ] ){        //si el vertice adyacente no fue visitado
```

Evaluamos primero para vértice 2

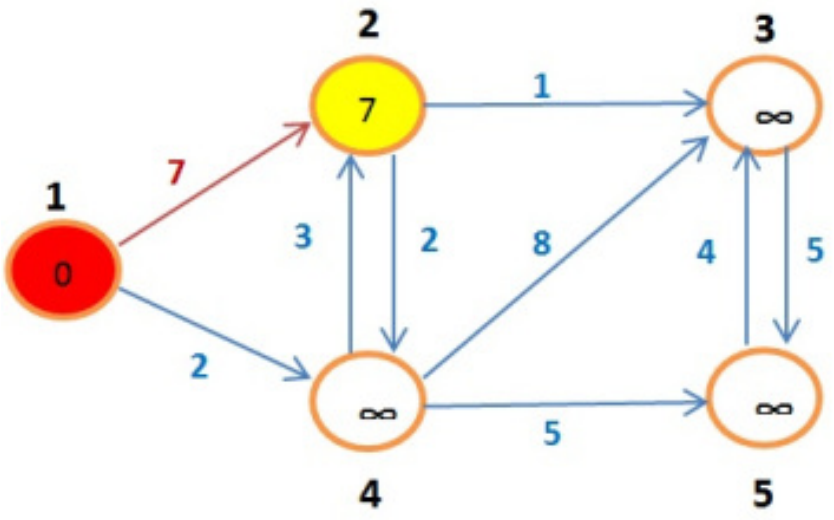


Vemos que la distancia actual desde el vértice inicial a 2 es ∞, verifiquemos el paso de relajación:

distancia[1] + 7 < distancia[2] -> 0 + 7 < ∞ -> 7 < ∞

El paso de relajación es posible realizarlo entonces actualizamos la distancia en el vértice 2 y agregando el vértice en la cola de prioridad con nueva distancia quedando:

Vértices	1	2	3	4	5
Distancia[u]	0	7	∞	∞	∞
Visitado[u]	1	0	0	0	0
Previo[u]	-1	1	-1	-1	-1



Cola	
Nodo	Peso
2	7

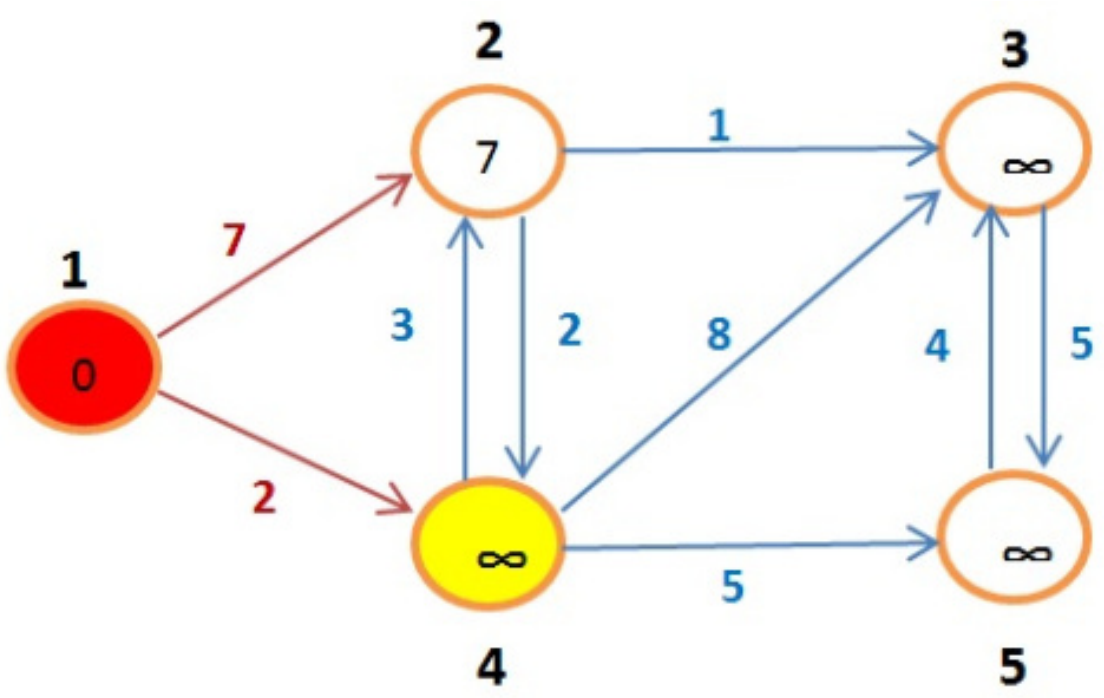
El paso de relajación se daría en la siguiente parte:

```
    for( int i = 0 ; i < ady[ actual ].size() ; ++i ){ //reviso sus adyacentes
1   del vertice actual
2       adyacente = ady[ actual ][ i ].first;   //id del vertice adyacente
3       peso = ady[ actual ][ i ].second;        //peso de la arista que une
4                                   actual con adyacente ( actual , adyacente )
5       if( !visitado[ adyacente ] ){        //si el vertice adyacente
6                               no fue visitado
            relajacion( actual , adyacente , peso ); //realizamos el paso de relajacion
7       }
    }
```

Donde la función de relajación seria

```
    //Paso de relajacion
1   void relajacion( int actual , int adyacente , int peso ){
2       //Si la distancia del origen al vertice actual + peso de su arista
3        //es menor a la distancia del origen al vertice adyacente
4       if( distancia[ actual ] + peso < distancia[ adyacente ] ){
5           distancia[ adyacente ] = distancia[ actual ] + peso;  //relajamos el
                    vertice actualizando la distancia
6           previo[ adyacente ] = actual;                          //a su vez
                            actualizamos el vertice previo
7           Q.push( Node( adyacente , distancia[ adyacente ] ) ); //agregamos
8                                   adyacente a la cola de prioridad
9       }
    }
```

Ahora evaluamos al siguiente adyacente que es el vértice 4:



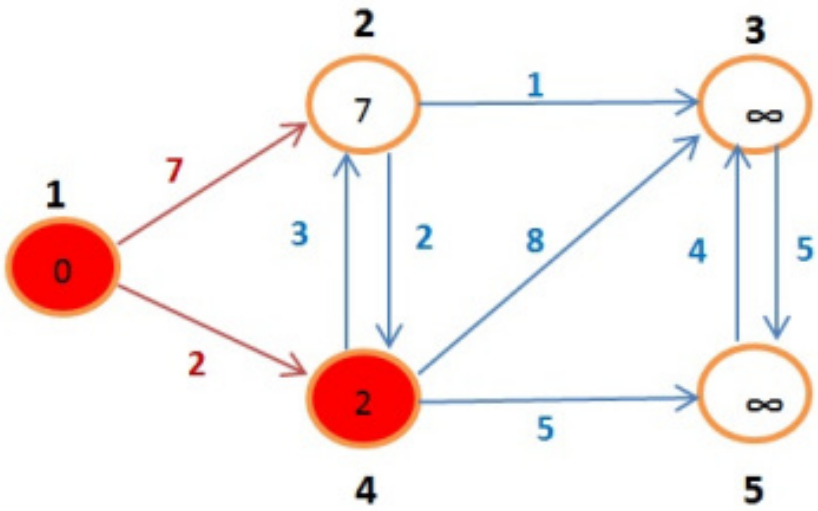
De manera similar al anterior vemos que la distancia actual desde el vértice inicial a 4 es ∞, verifiquemos el paso de relajación:

distancia[1] + 2 < distancia[4] -> 0 + 2 < ∞ -> 2 < ∞

El paso de relajación es posible realizarlo entonces actualizamos la distancia en el vértice 4 quedando:

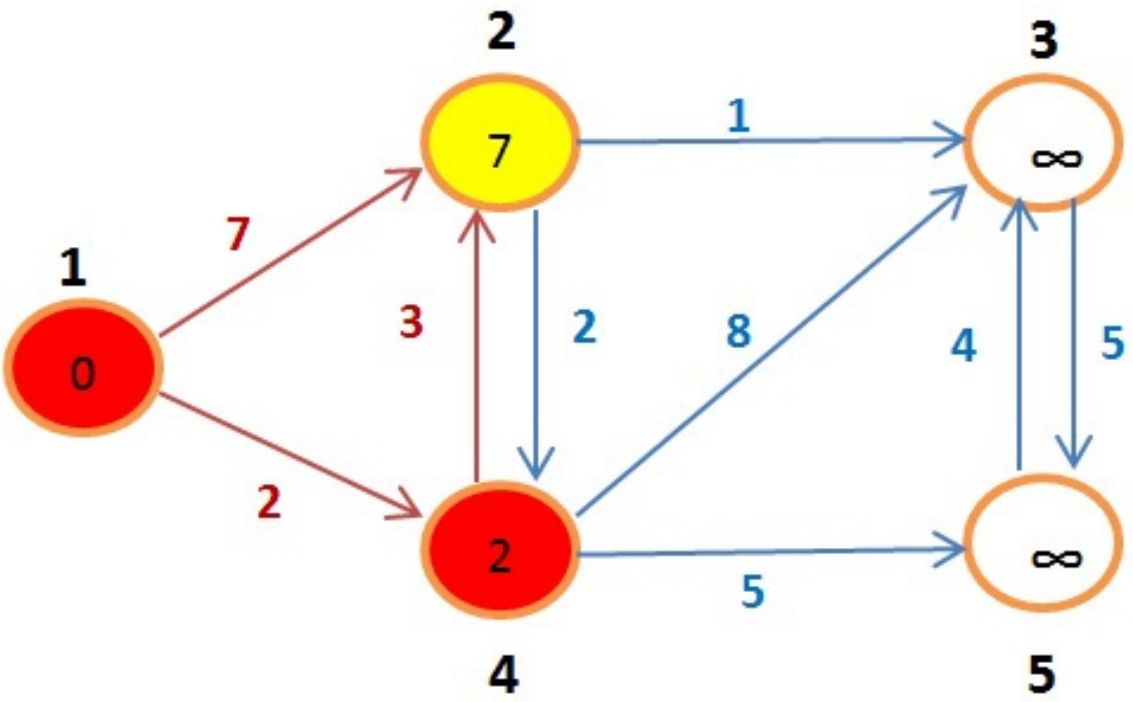
Vértices	1	2	3	4	5
Distancia[u]	0	7	∞	2	∞
Visitado[u]	1	0	0	0	0
Previo[u]	-1	1	-1	1	-1

En cuanto a la cola de prioridad como tenemos un vértice con menor peso este nuevo vértice ira en el tope de la cola:



Cola	
Nodo	Peso
4	2
2	7

Revisamos sus adyacentes no visitados que serian vértices 2, 3 y 5.
Empecemos por el vértice 2:



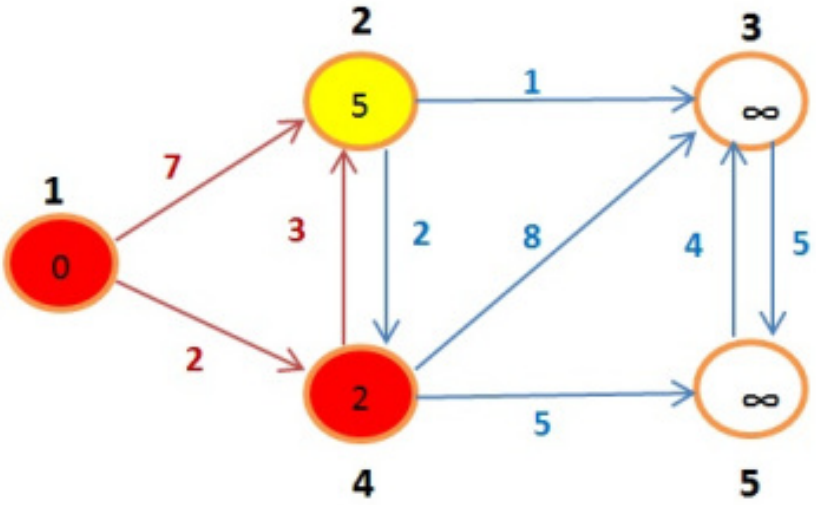
Ahora vemos que la distancia actual del vértice inicial al vértice 2 es 7, verifiquemos el paso de relajación:

distancia[4] + 3 < distancia[2] -> 2 + 3 < 7 -> 5 < 7

En esta oportunidad hemos encontrado una ruta mas corta partiendo desde el vértice inicial al vértice 2, actualizamos la distancia en el vértice 2 y actualizamos el vértice previo al actual quedando:

Vértices	1	2	3	4	5
Distancia[u]	0	5	∞	2	∞
Visitado[u]	1	0	0	1	0
Previo[u]	-1	4	-1	1	-1

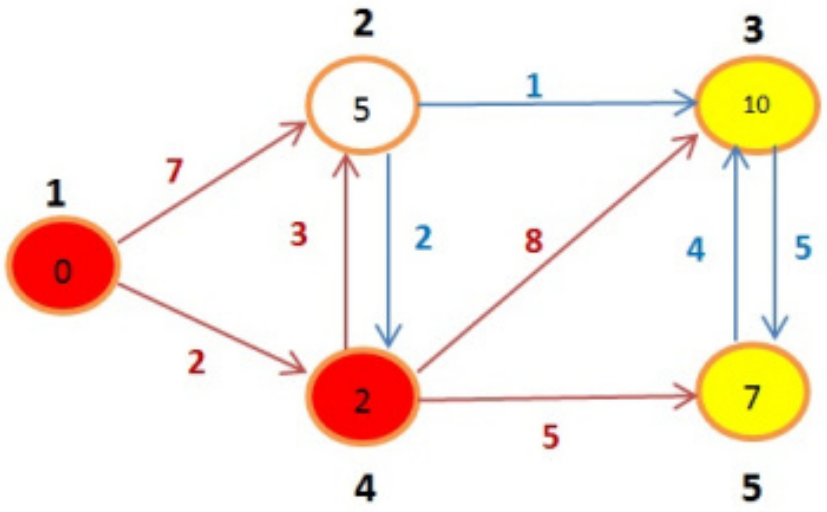
En cuanto a la cola de prioridad como tenemos un vértice con menor peso este nuevo vértice ira en el tope de la cola, podemos ver que tenemos 2 veces el mismo vértice pero como usamos una técnica greedy siempre usaremos el valor óptimo:



Cola	
Nodo	Peso
2	5
2	7

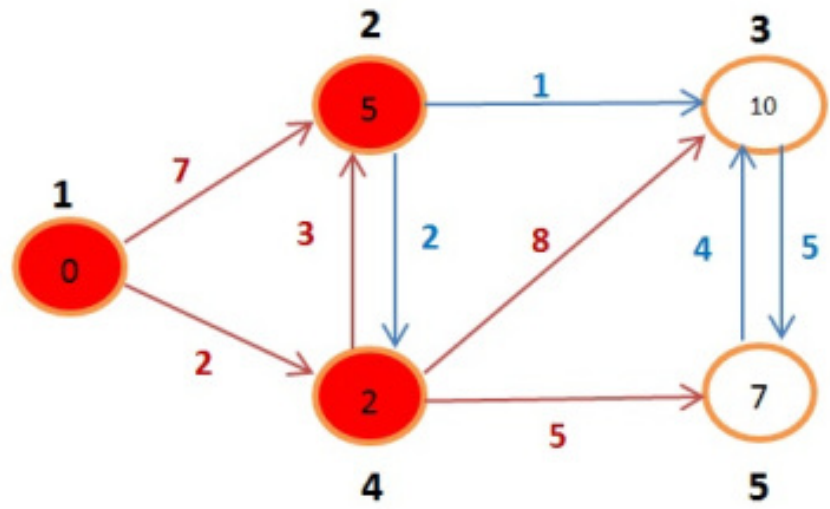
Continuamos con los Vértices 3 y 5 como tienen valor ∞ si será posible relajarlos por lo que sería similar a los pasos iniciales solo que en los pasos iniciales distancia[1] era 0 en este caso distancia[4] es 2, quedando:

Vértices	1	2	3	4	5
Distancia[u]	0	5	10	2	7
Visitado[u]	1	0	0	1	0
Previo[u]	-1	4	4	1	4



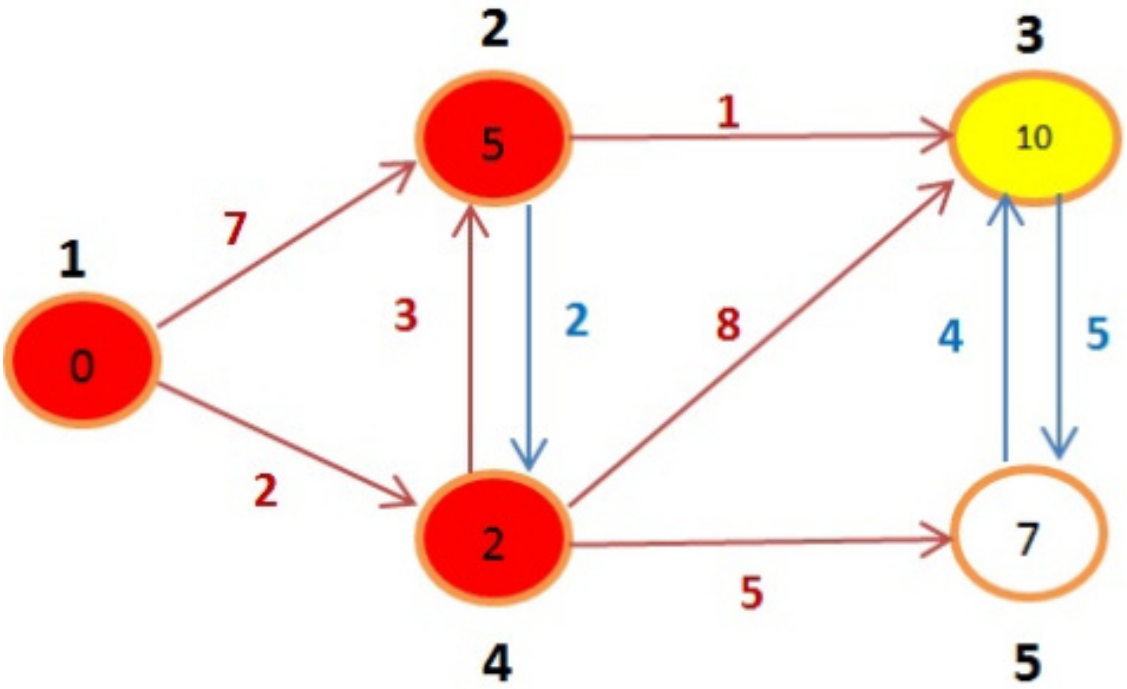
Cola	
Nodo	Peso
2	5
2	7
5	7
3	10

Hemos terminado de evaluar al vértice 4, continuamos con el tope de la cola que es vértice 2, el cual marcamos como visitado.



Cola	
Nodo	Peso
2	5
2	7
5	7
3	10

Los adyacentes no visitados del vértice 2 son los vértices 3 y 5. Comencemos con el vértice 3

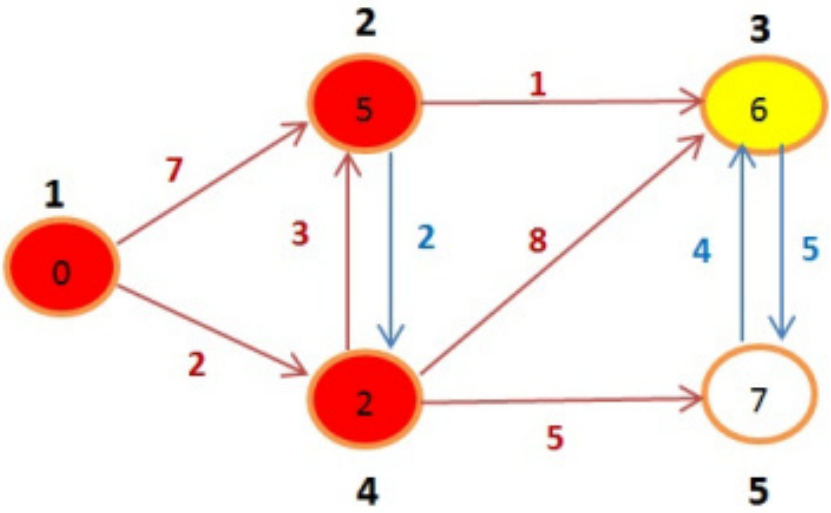


Ahora vemos que la distancia actual del vértice inicial al vértice 3 es 10, verifiquemos el paso de relajación:

distancia[2] + 1 < distancia[3] -> 5 + 1 < 10 -> 6 < 10

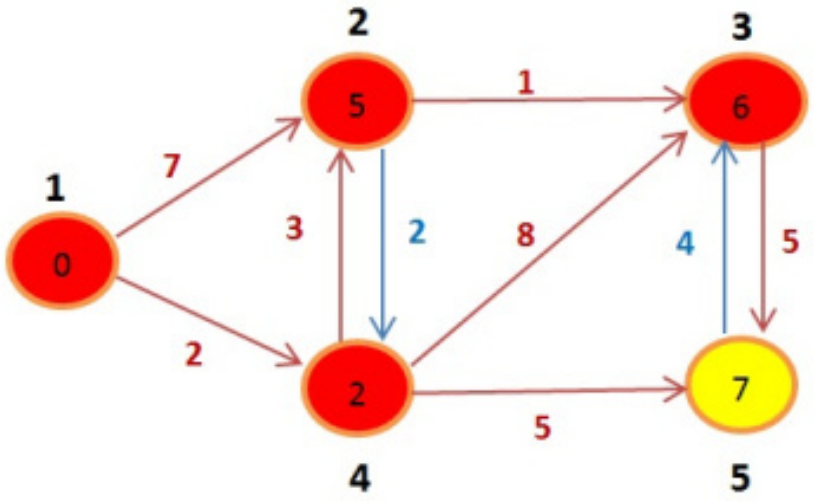
En esta oportunidad hemos encontrado una ruta mas corta partiendo desde el vértice inicial al vértice 3, dicha ruta sería 1 -> 4 -> 2 -> 3 cuyo peso es 6 que es mucho menor que la ruta 1 – > 4 -> 3 cuyo peso es 10, actualizamos la distancia en el vértice 3 quedando:

Vértices	1	2	3	4	5
Distancia[u]	0	5	6	2	7
Visitado[u]	1	1	0	1	0
Previo[u]	-1	4	2	1	4



Cola	
Nodo	Peso
3	6
2	7
5	7
3	10

El siguiente vértice de la cola de prioridad es el vértice 3 y su único adyacente no visitado es el vértice 5.



Cola	
Nodo	Peso
3	6
2	7
5	7
3	10

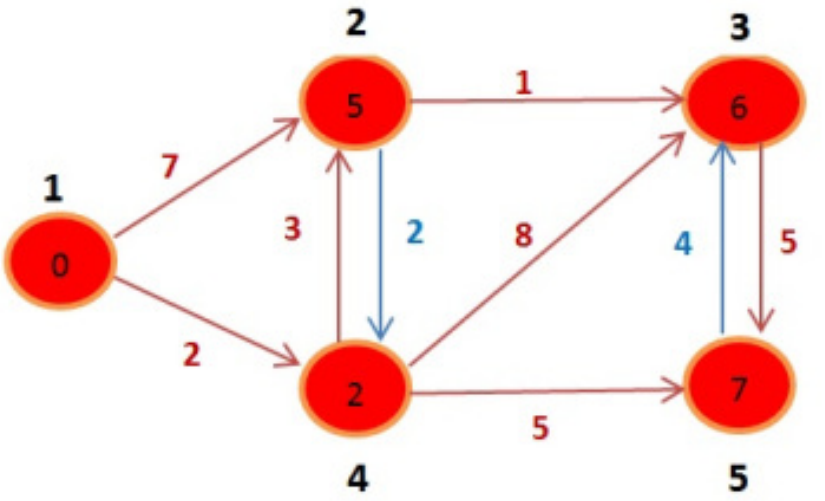
Vemos que la distancia actual del vértice inicial al vértice 5 es 7, verifiquemos el paso de relajación:

distancia[3] + 5 < distancia[5] -> 6 + 5 < 7 -> 11 < 7

En esta oportunidad se no cumple por lo que no relajamos el vértice 5, por lo que la tabla en cuanto a distancias no sufre modificaciones y no agregamos vértices a la cola:

Vértices	1	2	3	4	5
Distancia[u]	0	5	6	2	7
Visitado[u]	1	1	1	1	0
Previo[u]	-1	4	2	1	4

Ahora tocaría el vértice 2 pero como ya fue visitado seguimos extrayendo elementos de la cola, el siguiente vértice será el 5.



Nodo	Peso
2	7
5	7
3	10

Nodo	Peso
5	7
3	10

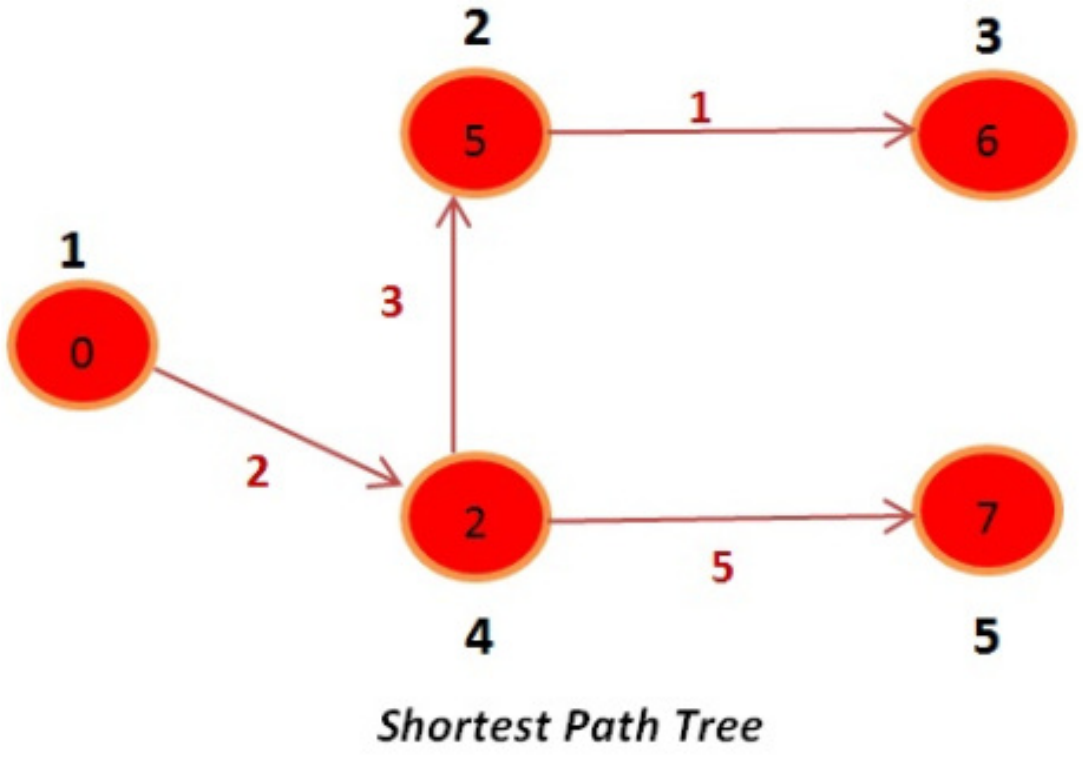
Al ser el ultimo vértice a evaluar no posee adyacentes sin visitar por lo tanto hemos terminado el algoritmo. En el grafico anterior observamos que 2 aristas no fueron usadas para la relajación, las demás si fueron usadas. La tabla final quedaría de la siguiente manera:

Vértices	1	2	3	4	5
Distancia[u]	0	5	6	2	7
Visitado[u]	1	1	1	1	1
Previo[u]	-1	4	2	1	4

De la tabla si deseo saber la distancia mas corta del vértice 1 al vértice 5, solo tengo que acceder al valor del arreglo en su índice respectivo (distancia[5]).

Shortest Path Tree

En el proceso anterior usábamos el arreglo previo[u] para almacenar el ID del vértice previo al vértice con ID = u, ello me sirve para formar el árbol de la ruta mas corta y además me sirve para imprimir caminos de la ruta mas corta.



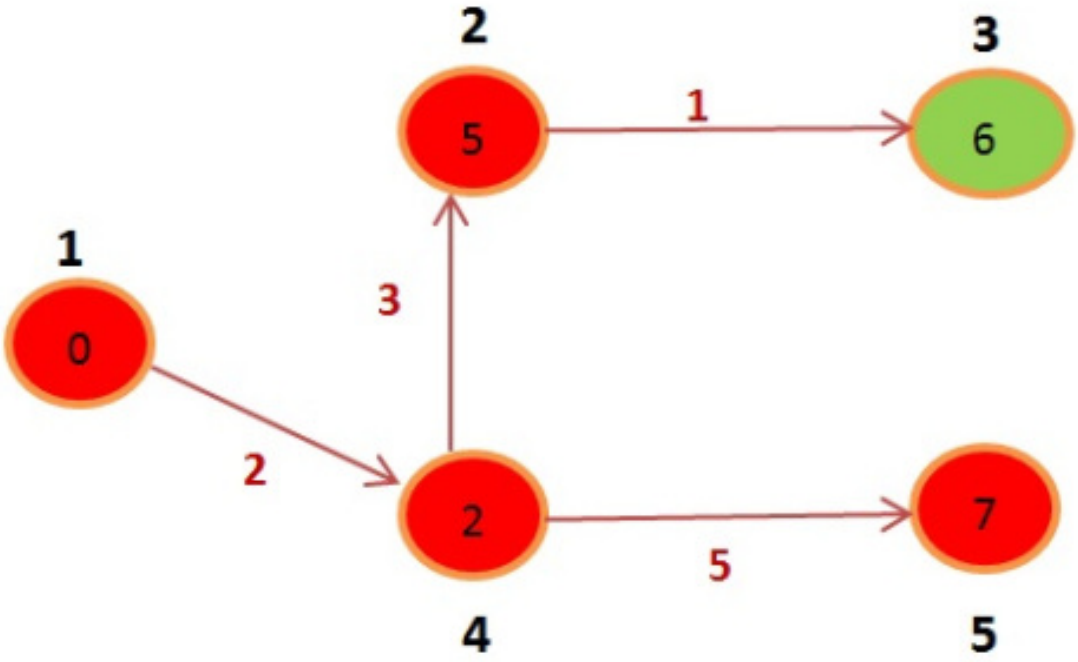
Shortest Path Tree

Impresión del camino encontrado.

Para imprimir el camino mas corto deseado usamos el arreglo previo[u], donde u tendrá el ID del vértice destino, o sea si quiero imprimir el camino mas corto desde vértice 1 -> vértice 3 partiré desde previo[3] hasta el previo[1]. De manera similar a lo que se explico en el algoritmo BFS, en este caso se realizara de manera recursiva:

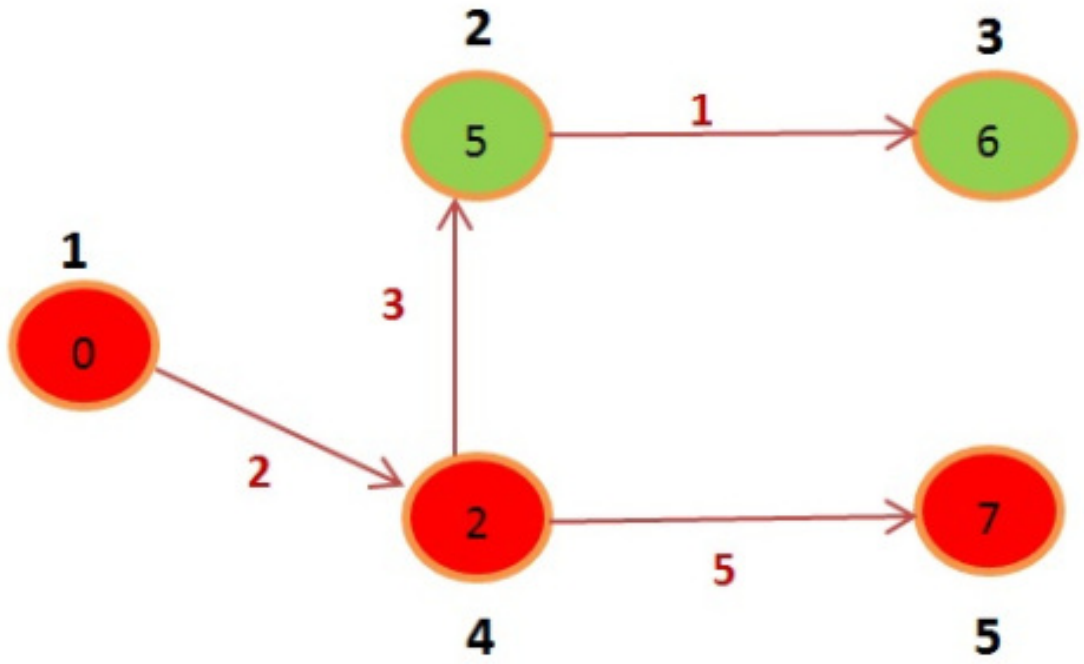
```
1  //Impresion del camino mas corto desde el vertice inicial y final ingresados
2  void print( int destino ){
3      if( previo[ destino ] != -1 )     //si aun poseo un vertice previo
4          print( previo[ destino ] );  //recursivamente sigo explorando
       printf("%d " , destino );        //terminada la recursion imprimo
5  los vertices recorridos
6  }
```

Veamos gráficamente el funcionamiento, desde el grafo comenzamos en 3



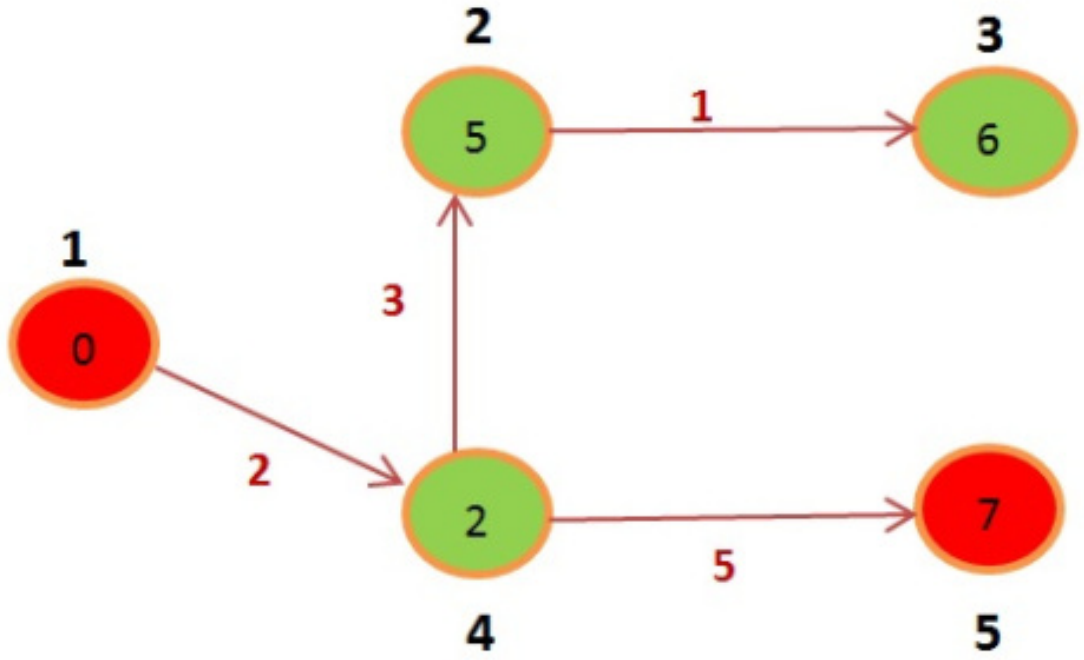
Vértices	1	2	3	4	5
Previo[u]	-1	4	2	1	4

El previo de 3 es el vértice 2, por lo tanto ahora evaluó 2:



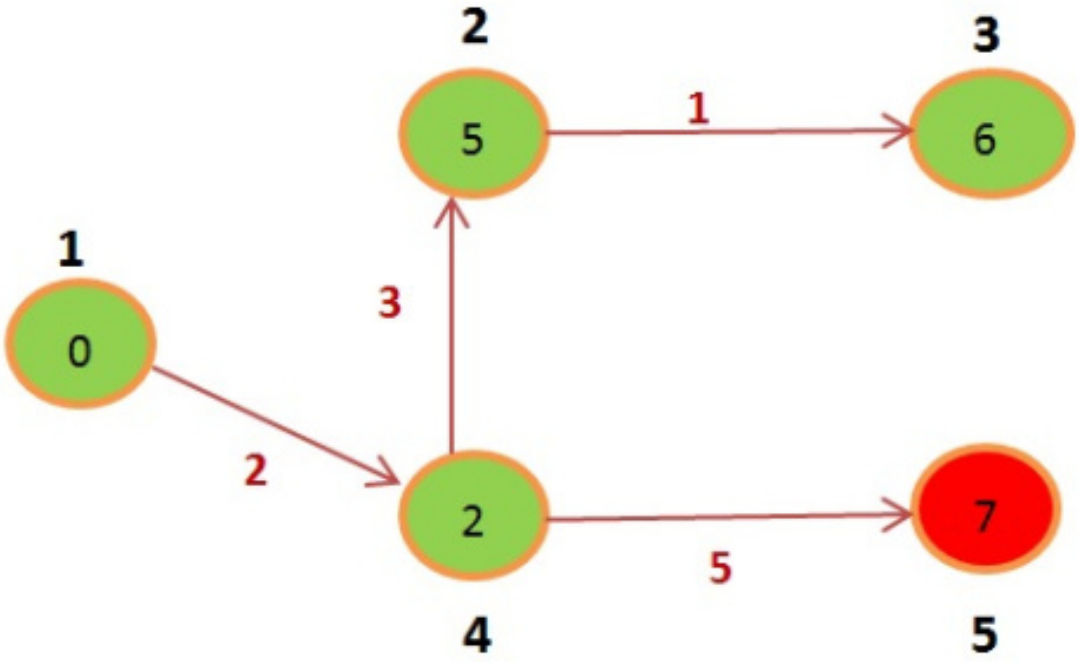
Vértices	1	2	3	4	5
Previo[u]	-1	4	2	1	4

Ahora el previo de 2 es el vértice 4:



Vértices	1	2	3	4	5
Previo[u]	-1	4	2	1	4

El previo de 4 es el vértice inicial 1



Vértices	1	2	3	4	5
Previo[u]	-1	4	2	1	4

Como el previo de 1 es -1 terminamos el recorrido, ahora en el retorno de las llamadas recursivas imprimo el camino: 1 4 2 3

Referencias

Arias, J. (2012). CAMINO MAS CORTO: ALGORITMO DE DIJKSTRA. 2020, mayo 13, de Algorithms and More Recuperado de https://jariasf.wordpress.com/2012/03/19/camino-mas-corto-algoritmo-de-dijkstra/