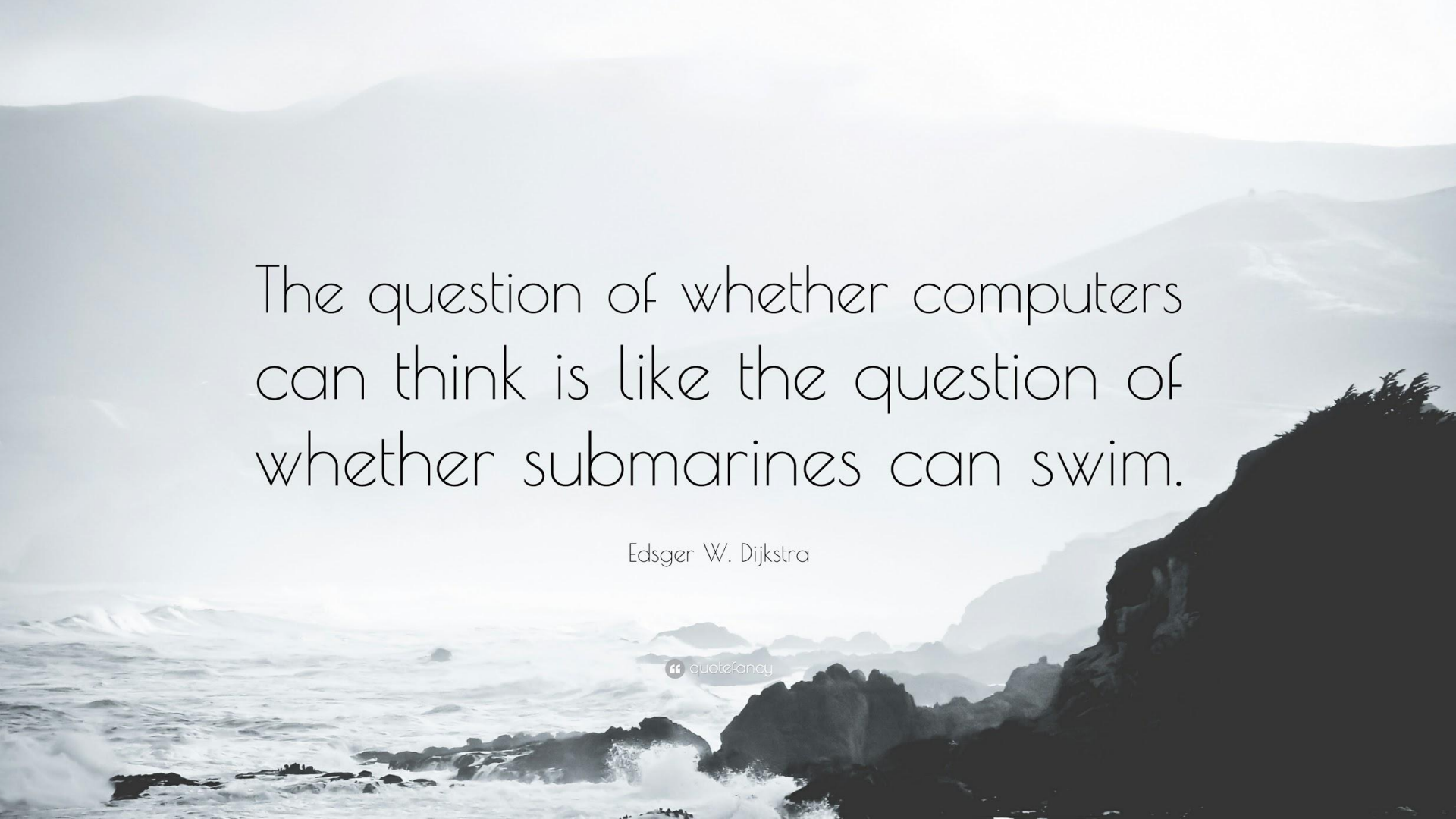


Grafos - parte 2

Dijkstra



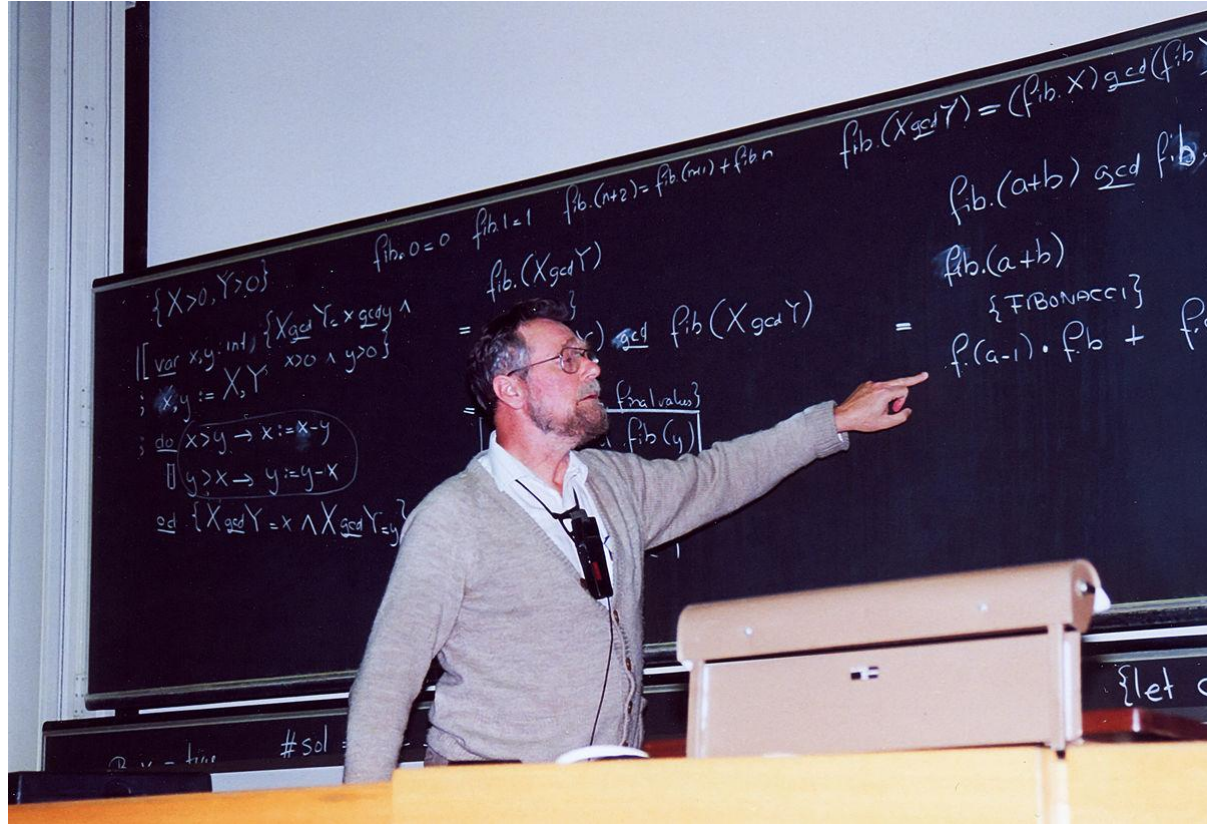
The question of whether computers
can think is like the question of
whether submarines can swim.

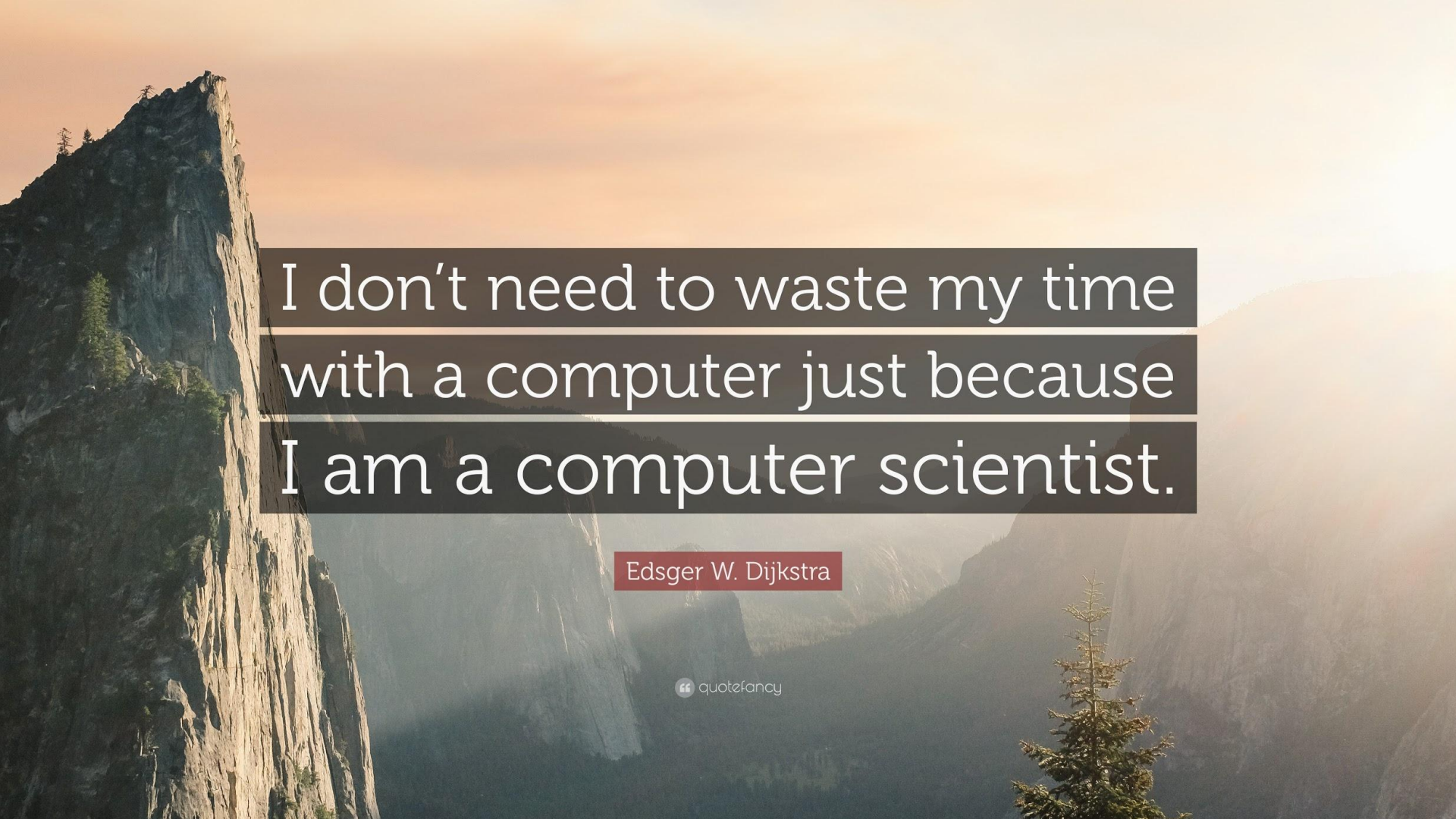
Edsger W. Dijkstra

“ quote fancy ”

Dijkstra

Róterdam, Países Bajos, 11 de mayo de
1930 - Nuenen, Países Bajos, 6 de
agosto de **2002**





I don't need to waste my time
with a computer just because
I am a computer scientist.

Edsger W. Dijkstra

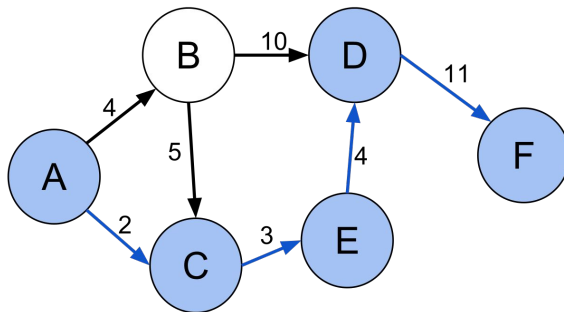
Camino más corto

Dijkstra

Algoritmo de Dijkstra

Es un algoritmo que nos permite saber el costo y el camino a todo vértice desde un origen dado.

Se aplica a grafos ponderados (*) **siempre y cuando las aristas no sean negativas.**

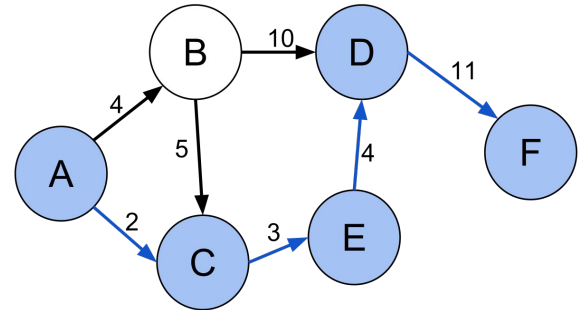


(*) se puede aplicar a grafos no ponderados tomando a las aristas con peso 1.

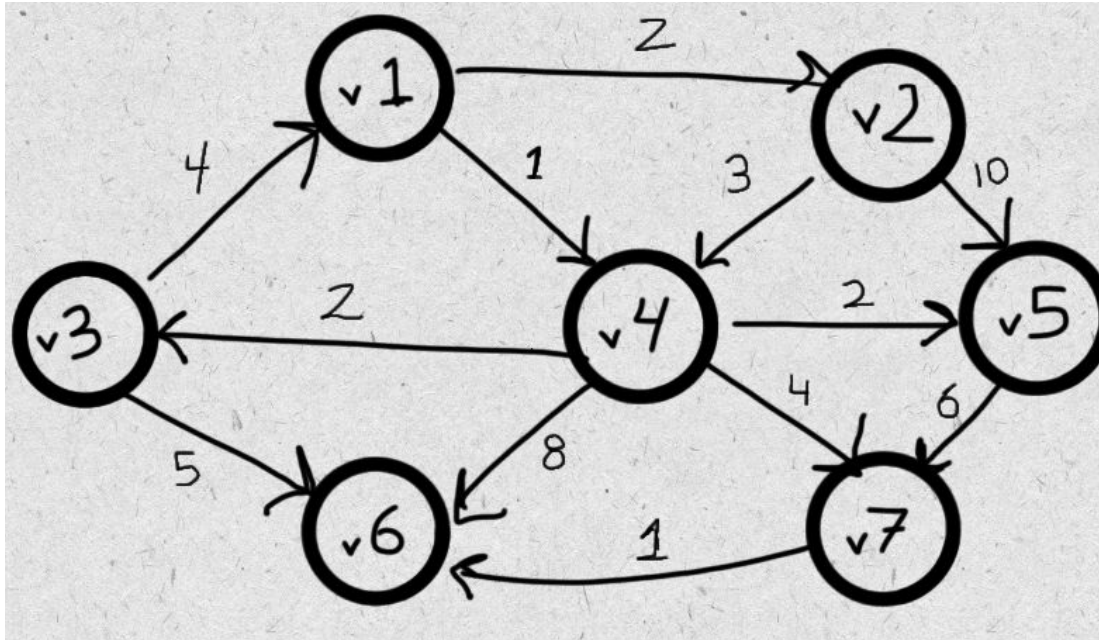
Algoritmo de Dijkstra - idea general

En un principio, todos los nodos están desconocidos y con “costo infinito” a excepción del nodo origen (con costo 0).

En cada iteración se busca el nodo desconocido con menor coste de llegada. Al procesar dicho nodo, se visitan los adyacentes y se actualiza el costo de llegada (tentativo hasta el momento de procesarlos).



Algoritmo de Dijkstra - ejemplo



Dijkstra ejemplo 1

Algoritmo de Dijkstra - ejemplo 2

<https://www.cs.usfca.edu/~galles/visualization/Dijkstra.html>

“What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, **which I designed in about twenty minutes.**”



Edsger Dijkstra, in an interview with Philip L. Frana, Communications of the ACM, 2001

“What is the shortest way to travel from Rotterdam to Groningen, in general: from given city to given city. It is the algorithm for the shortest path, **which I designed in about twenty minutes.**”




Edsger Dijkstra, in an interview with Philip L. Frana, Communications of the ACM, 2001

Pseudocódigo

Algoritmo de Dijkstra - pseudocódigo V1

```
void dijkstra(int origen)
    int* visitados = initVisitados(); // array V casilleros, todos en false
    int* costos = initCostos(origen); // array V casillero, todos en INF menos origen con 0
    int* vengo= initVengo(); // array V casilleros, todos en -1
    for(int i=0; i<V; i++)
        int v = verticeDesconocidoDeMenorCosto(visitados, costos); // vértice a procesar
        visitados[v] = true;
        para cada w adyacente a v
            if(costos[w] > costos[v] + dist(v,w))
                costos[w] = costos[v] + dist(v,w);
                vengo[w] = v;
```



Podría no haber!

Algoritmo de Dijkstra - pseudocódigo V1

```
void dijkstra(int origen)
    int* visitados = initVisitados(); // array V casilleros, todos en false
    int* costos = initCostos(origen); // array V casillero, todos en INF menos origen con 0
    int* vengo= initVengo(); // array V casilleros, todos en -1
    for(int i=0; i<V; i++)
        int v = verticeDesconocidoDeMenorCosto(visitados, costos); // vértice a procesar
        visitados[v] = true;
        para cada w adyacente a v
            if(costos[w] > costos[v] + dist(v,w))
                costos[w] = costos[v] + dist(v,w);
                vengo[w] = v;
```

Algoritmo de Dijkstra - pseudocódigo V1

```
void dijkstra(int origen)
    int* visitados = initVisitados(); //  $O(V)$ 
    int* costos = initCostos(origen); //  $O(V)$ 
    int* vengo = initVengo(); //  $O(V)$ 
    for(int i=0; i<V; i++) //  $O(V^2)$ 
        int v = verticeDesconocidoDeMenorCosto(visitados, costos); //  $O(V)$ 
        visitados[v] = true;
        para cada w adyacente a v //  $O(???)$ 
            if(costos[w] > costos[v] + dist(v,w))
                costos[w] = costos[v] + dist(v,w);
                vengo[w] = v;
```


Algoritmo de Dijkstra - pseudocódigo V1

Esta implementación específica es de $O(V^2 + A) = O(V^2)$ lo cual no está mal siempre y cuando sea un grafo denso.

¿ y para grafos dispersos ? -> se puede mejorar

Nuestro problema está en decidir cuál es el nuevo vértice a visitar.

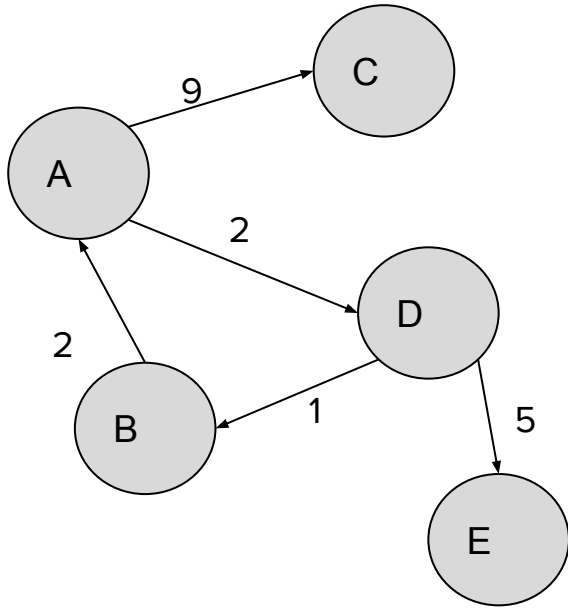
Debido a que están “ordenados” por costo podríamos usar una estructura auxiliar para decidir cuál es el próximo vértice a visitar/procesar.

Algoritmo de Dijkstra - pseudocódigo V2

```
void dijkstra(int origen)
    int* visitados = initVisitados();
    int* costos = initCostos(origen);
    int* vengo= initVengo();
    ColaPrioridad cp();
    cp.encolar(origen, 0); // se encola el origen con prioridad 0
    while(!cp.estaVacia())
        int v = cp.desencolar(); // extrae el vértice de menor costo no visitado
        visitados[v] = true;
        para cada w adyacente a v
            if(costos[w] > costos[v] + dist(v,w))
                costos[w] = costos[v] + dist(v,w);
                vengo[w] = v;
                cp.encolar(w, costos[w]); // se encola el origen con prioridad costos[w]
```

Algoritmo de Dijkstra - pseudocódigo V2

Origen: A



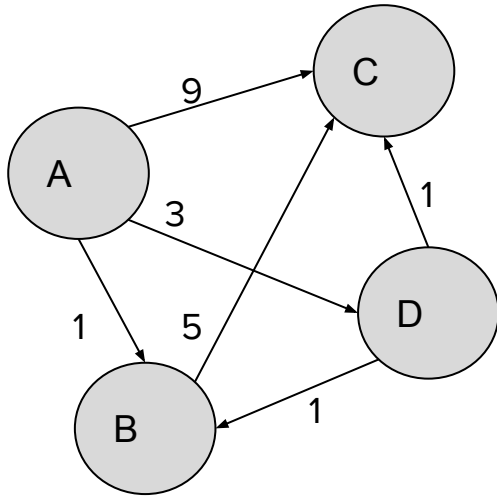
Momento	Cola de prioridad
Al principio	(A,0)
Luego de A	(D,2),(C,9)
Luego de D	(B,3),(E,7),(C,9)
Luego de B	(E,7),(C,9)
Luego de E	(C,9)
Luego de C	

Algoritmo de Dijkstra - pseudocódigo V2

```
void dijkstra(int origen)
    int* visitados = initVisitados();
    int* costos = initCostos(origen);
    int* vengo= initVengo();
    ColaPrioridad cp();
    cp.encolar(origen, 0); // se encola el origen con prioridad 0
    while(!cp.estaVacia())
        int v = cp.desencolar(); // extrae el vértice de menor costo no visitado
        visitados[v] = true;
        para cada w adyacente a v
            if(!visitados[w] && costos[w] > costos[v] + dist(v,w))
                costos[w] = costos[v] + dist(v,w);
                vengo[w] = v;
                cp.encolar(w, costos[w]); // PROBLEMA: múltiples inserciones del mismo vértice
```

Algoritmo de Dijkstra - pseudocódigo V2

Origen: A



Momento	Cola de prioridad
Al principio	(A,0)
Luego de A	(B,1),(D,3),(C,9)
Luego de B	(D,3),(C,6),(C,9)
Luego de D	(C,4),(C,6),(C,9)
Luego de C	(C,6),(C,9)

Algoritmo de Dijkstra - pseudocódigo V2

Soluciones:

- A. Usar una CP extendida: para actualizar la prioridad de los elementos que ya existen en la CP.
- B. Insertar repetidos teniendo en cuenta que al momento de procesar ya no este visitado.

Algoritmo de Dijkstra - pseudocódigo V2 - A

```
void dijkstra(int origen)
    int* visitados = initVisitados();
    int* costos = initCostos(origen);
    int* vengo= initVengo();
    ColaPrioridad cp();
    for(int i=1; i<=V; i++) { cp.encolar(i, costos[i]) } // se encolan todos los vértices con inf, menos el origen con 0
    while(!cp.estaVacia())
        int v = cp.desencolar(); // extrae el vértice de menor costo no visitado
        visitados[v] = true;
        para cada w adyacente a v
            if(!visitados[w] && costos[w] > costos[v] + dist(v,w))
                costos[w] = costos[v] + dist(v,w);
                vengo[w] = v;
                cp.actualizarPrioridad(w, costos[w]); // se actualiza la prioridad
```

Algoritmo de Dijkstra - pseudocódigo V2 - B

```
void dijkstra(int origen)
    int* visitados = initVisitados();
    int* costos = initCostos(origen);
    int* vengo= initVengo();
    ColaPrioridad cp();
    cp.encolar(origen, 0); // se encola el origen con prioridad 0
    while(!cp.estaVacia())
        int v = cp.desencolar(); // extrae el vértice de menor costo no visitado
        if(!visitados[v]) // se verifica que no haya sido visitado antes
            visitados[v] = true;
            para cada w adyacente a v
                if(!visitados[w] && costos[w] > costos[v] + dist(v,w))
                    costos[w] = costos[v] + dist(v,w);
                    vengo[w] = v;
                    cp.encolar(w, costos[w]); // se encola el origen con prioridad costos[w]
```


Algoritmo de Dijkstra - pseudocódigo V2.A orden

Cuando hablamos de grafos dispersos

$$O(A.\text{Log}V + V.\text{Log}V) = \mathbf{O((A+V).\text{Log}V)}$$

Es de suponer que ante grafos dispersos usemos V2 y ante grafos densos V1.

A codificar

Camino + corto con ponderación negativa

Camino + corto con ponderación negativa

El problema de Dijkstra es que una vez procesado el vértice asume que no se puede mejorar el costo de llegar hasta el.

La solución es una combinación entre camino + corto no ponderado y Dijkstra.

Camino + corto con ponderación negativa

```
void CaminoMasCortoPonderacionNegativa(int origen) {  
    int * costo = initCosto(origen); // array de V casilleros, todos con valor "INF" a excepción de origen (con 0)  
    int * vengo = initVengo(); // array de V casilleros, todos con valor -1  
  
    Cola<int> cola;  
    cola.encolar(origen);  
  
    while (!cola.estaVacia())  
        int v= cola.desencolar();  
        paraCada w adyacenteA vertice  
            if(costo[w] > costo[v] + dist(v,w))  
                costo[w] = costo[v] + dist(v,w);  
                vengo[w] = v;  
                if(!cola.existe(w)) // preguntamos la cola si ya existe el elementos, esto también se puede hacer un array en O(1)  
                    cola.encolar(w);  
}
```

Camino + corto con ponderación negativa

El orden del algoritmo es $O(V.A)$ ya que a lo sumo se puede mejorar el costo (encolar a la lista) A veces.

¿ ciclos negativos?

Camino + corto entre todo par de vértices

Floyd

Camino + corto entre todo par de vértices

Existe un algoritmo que resuelve este problema en $O(V^3)$ siendo un algoritmo muy simple de codificar: **el algoritmo de Floyd**

La idea general del algoritmo es usar un vértice intermediario para saber si por dicho vértice se puede encontrar un mejor camino.

Nota: funciona con aristas negativas

Floyd


```
void floyd() {  
    int** matriz = grafoMatriz(); // devuelve una copia de la matriz de adyacencia  
    for(int i=1; i<=V; i++)  
        matriz[i][i]=0; // “borramos” la diagonal  
  
    for(int k=1; k<=V; k++) // el nodo “intermedio”  
        for(int i=1; i<=V; i++)  
            for(int j=1; j<=V; j++)  
                if(matriz[i][j] > matriz[i][k] + matriz[k][j])  
                    matriz[i][j] = matriz[i][k] + matriz[k][j]
```

<https://www.cs.usfca.edu/~galles/visualization/Floyd.html>

Floyd

```
void floyd() {  
    int** matriz = grafoMatriz(); // devuelve una copia de la matriz de adyacencia  
    for(int i=1; i<=V; i++)  
        matriz[i][i]=0; // “borramos” la diagonal  
  
    for(int k=1; k<=V; k++) // el nodo “intermedio”  
        for(int i=1; i<=V; i++)  
            for(int j=1; j<=V; j++)  
                if(matriz[i][j] > matriz[i][k] + matriz[k][j])  
                    matriz[i][j] = matriz[i][k] + matriz[k][j]
```

Floyd

```
void floyd() {  
      
    int** matriz = grafoMatriz(); // devuelve una copia de la matriz de adyacencia  
    for(int i=1; i<=V; i++)  
        matriz[i][i]=0; // “borramos” la diagonal  
  
    for(int k=1; k<=V; k++) // el nodo “intermedio”  
        for(int i=1; i<=V; i++)  
            for(int j=1; j<=V; j++)  
                if(matriz[i][j] > matriz[i][k] + matriz[k][j])  
                    matriz[i][j] = matriz[i][k] + matriz[k][j]  
}
```

Floyd - camino

```
void floyd() {  
    int** matriz = grafoMatriz(); // devuelve una copia de la matriz de adyacencia  
    int** matrizCamino = initCaminos();  
    for(int i=1; i<=V; i++)  
        matriz[i][i]=0; // “borramos” la diagonal  
  
    for(int k=1; k<=V; k++) // el nodo “intermedio”  
        for(int i=1; i<=V; i++)  
            for(int j=1; j<=V; j++)  
                if(matriz[i][j] > matriz[i][k] + matriz[k][j])  
                    matriz[i][j] = matriz[i][k] + matriz[k][j]  
                    matrizCamino[i][j] = k;
```

<https://www.cs.usfca.edu/~galles/visualization/Floyd.html>

Floyd vs V Dijkstra's

Si es disperso: V Dijkstra's $\Rightarrow O(V^*(A+V).LogV)$

Si es denso o con aristas negativas: Floyd $\Rightarrow O(V^3)$

A codificar

Matriz de Clausura Transitiva

Warshall

Matriz de Clausura Transitiva

Muestra si existe un camino con todo par de vértices directa o indirectamente.

Utilizaremos el algoritmo de Warshall (muy parecido a floyd).

Warshall

```
void warshall()
```

```
    bool** matriz = grafoAristas(); // devuelve una matriz de VxV donde matriz[i][j] = true si hay una arista i->j
```

```
    for(int k=1; k<=V; k++) // el nodo “intermedio”
```

```
        for(int i=1; i<=V; i++)
```

```
            for(int j=1; j<=V; j++)
```

```
                matriz[i][j] = matriz[i][j] || matriz[i][k] && matriz[k][j];
```

Warshall - usos

Warshall - usos

Encontrar ciclos (grafos dirigidos)

Warshall - usos

Encontrar ciclos (grafos dirigidos)

Conexidad: conexo/no conexo, fuertemente conexo/débilmente conexo

Links de interés

- <https://clementmihailescu.github.io/Pathfinding-Visualizer/>