# Algoritmo de Dijkstra

*Versión 1.0/Estructuras de datos no Lineales*

CAMAVINGAGII

2025

# ÍNDICE GENERAL

CAPÍTULO 1

# EXPLICACIÓN BREVE DE DIJKSTRA

- Dijkstra es un algoritmo que trabaja con un grafo (en concreto, con la matriz de costes de un grafo, y vectores auxiliares que posteriormente se mencionarán) para **encontrar el camino mínimo desde UN ÚNICO NODO hacia los demás** en el grafo.
- Para ello, lo que hace es lo siguiente (cabe recalcar que poseemos 3 vectores, uno de ellos guarda la información de los costes de ir desde el nodo de partida hacia los demás nodos, que es el vector denominado D):
  1. Primero, busca el nodo **más cercano (menor coste desde el origen hasta ese nodo)**.
  2. Posteriormente, a partir de ese nodo hallado como el más cercano, **halla los caminos a los demás nodos**, y, en el caso de que exista un camino cuyo **coste sea menor que el camino almacenado** inicialmente (en el vector D) desde el nodo de partida a ese nodo en concreto, se **actualiza** su valor en D para representar el nuevo coste y en P para saber que hemos de pasar por ese nodo.

CAPÍTULO 2

# CÓDIGO DEL ALGORITMO

- Antes de ver el código, debemos saber la existencia de 3 vectores:
  - **Vector S:** Vector de booleanos que indica los nodos que hemos utilizado para intentar mejorar los caminos encontrados
  - **Vector D:** Representa los costes de ir a los nodos (inicialmente desde el nodo de partida a los demás nodos). Es lo que devuelve la función.
  - **Vector P**: Se pasa por parámetro a la función. Guarda la información de los nodos que hay que tomar para llegar a otros nodos.
- Posteriormente, existen otras variables como v y w, las cuales una recorre los nodos del grafo y la otra va tomando el nodo más cercano no visitado, la función suma, que halla la suma de ir desde el nodo más cercano no visitado hasta el nodo v (si esa suma es menor que el valor almacenado en D en la posición v, es cuando se actualiza en D y en P).

```cpp
// Suma de costes
template <typename tCoste>
tCoste suma(tCoste x, tCoste y)
{
    const tCoste INFINITO = GrafoP<tCoste>::INFINITO;
    if (x == INFINITO || y == INFINITO)
        return INFINITO;
    else
        return x + y;
}


template <typename tCoste>
vector<tCoste> Dijkstra(const GrafoP<tCoste>& G,
typename GrafoP<tCoste>::vertice origen,
vector<typename GrafoP<tCoste>::vertice>& P)
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    vertice v, w; // v recorre los nodos del grafo, w es el nodo mas
        cercano no visitado
    const size_t n = G.numVert();
    vector<bool> S(n, false); // Conjunto de vertices vacio.
    vector<tCoste> D; // Costes minimos desde origen.
    // Iniciar D y P con caminos directos desde el vertice origen.
    D = G[origen];
    D[origen] = 0; // Coste origen-origen es 0.
    P = vector<vertice>(n, origen);

    // Calcular caminos de coste minimo hasta cada vertice.
    S[origen] = true; // Incluir vertice origen en S.
    for (size_t i = 1; i <= n-2; i++)
    {
    // Localizar vertice w no incluido en S con menor coste desde origen
    tCoste costeMin = GrafoP<tCoste>::INFINITO;
    //este bucle halla el vertice con menor coste no visitado desde el
        vertice tomado como origen
    for (v = 0; v <= n-1; v++)
    if (!S[v] && D[v] <= costeMin)
    {
    costeMin = D[v];
    w = v;
    }
    S[w] = true; // Incluir vertice w en S.
    // Recalcular coste hasta cada v no incluido en S, a traves de w.
    //este bucle se encarga de calcular el coste desde el vertice con
        menor coste w hasta cada vertice del grafo para ver si mejora el
        camino
    for (v = 0; v <= n-1; v++)
    if (!S[v])
    {
    tCoste Owv = suma(D[w], G[w][v]); //si la suma del coste de ir al
        vertice mas cercano con el coste de ir al otro nodo es menor que
        el coste de ir directamente desde el nodo de partida a ese nodo,
```

```
        actualizamos.
    if (Owv < D[v])
    {
        D[v] = Owv;
        P[v] = w;
    }
    }
    }
return D;
}
```

```
//definicion del grafo como matriz de adyacencia
#include <vector>
#include <limits>
template <typename T> class GrafoP { // Grafo ponderado
    public:
    typedef T tCoste;
    typedef size_t vertice; // un valor entre 0 y GrafoP::numVert()-1
    static const tCoste INFINITO; // peso arista inexistente
    explicit GrafoP(size_t n): costes(n, vector<tCoste>(n,INFINITO)){}
    size_t numVert() const {return costes.size();}
    const vector<tCoste>& operator [](vertice v) const {return costes[v
        ];}
    vector<tCoste>& operator [](vertice v) {return costes[v];}
    bool esDirigido() const;
    private:
    vector< vector<tCoste> > costes;
};

template <typename tCoste> typename GrafoP<tCoste>::tCamino
camino(typename GrafoP<tCoste>::vertice orig,
typename GrafoP<tCoste>::vertice v,
const vector<typename GrafoP<tCoste>::vertice>& P)
// Devuelve el camino de orig a v a partir de un vector
// P obtenido mediante la funcion Dijkstra().
{
    typename GrafoP<tCoste>::tCamino C;
    C.insertar(v, C.primera());
    do {
        C.insertar(P[v], C.primera());
        v = P[v];
} while (v != orig);
return C;
}
```