

Algoritmos de caminos de coste mínimo

Algoritmo de Floyd

```
template <typename tCoste>
matriz<tCoste> Floyd(const GrafoP<tCoste>& G,
                    matriz<typename GrafoP<tCoste>::vertice>& P)
```

Calcula los caminos de coste mínimo entre cada par de vértices del grafo **G**.

Salida:

- Una matriz de costes mínimos de tamaño $n \times n$, con $n = \mathbf{G.numVert}()$
- **P**, una matriz de vértices de tamaño $n \times n$, tal que **P[i][j]** es un vértice intermedio por el que pasa el camino de coste mínimo desde **i** a **j**.

```

/*-----*/
/*  matriz.h                                     */
/*-----*/
#ifndef MATRIZ_H
#define MATRIZ_H

#include <vector>

using std::vector;

// matriz cuadrada
template <typename T> class matriz {
public:
    matriz() {}
    explicit matriz(size_t n, const T& x = T())
        : m(n, vector<T>(n, x)) {}
    size_t dimension() const { return m.size(); }
    const vector<T>& operator [](size_t i) const { return m[i]; }
    vector<T>& operator [](size_t i) { return m[i]; }
private:
    vector< vector<T> > m;
};

#endif // MATRIZ_H

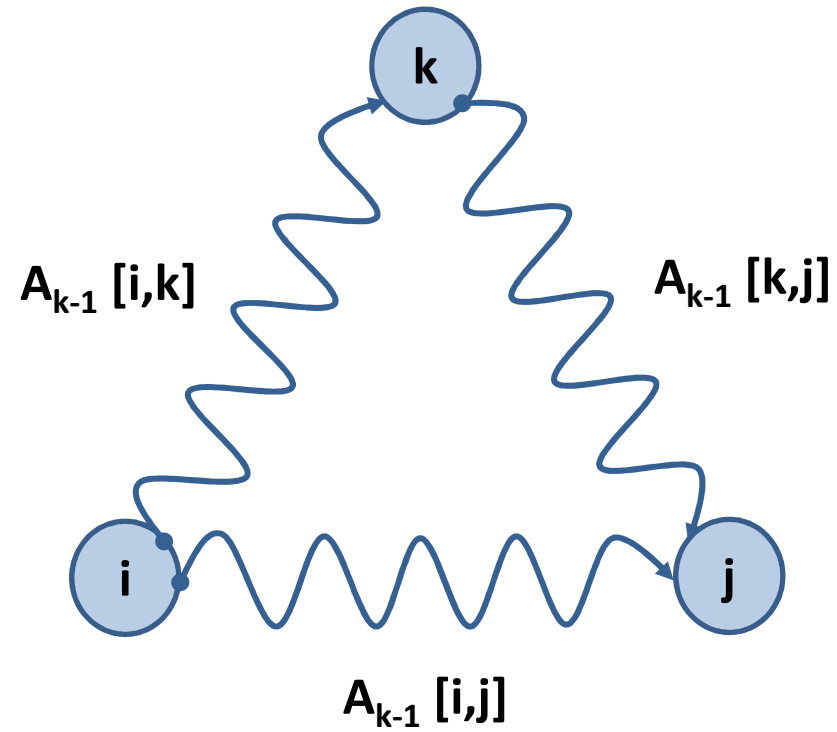
```

```

#include "matriz.h"

template <typename tCoste>
matriz<tCoste> Floyd(const GrafoP<tCoste>& G,
                    matriz<typename GrafoP<tCoste>::vertice>& P)
{
    typedef typename GrafoP<tCoste>::vertice vertice;
    const size_t n = G.numVert();
    matriz<tCoste> A(n); // matriz de costes mínimos
    // Iniciar A y P con caminos directos entre cada par de vértices.
    P = matriz<vertice>(n);
    for (vertice i = 0; i <= n-1; i++) {
        A[i] = G[i];           // copia costes del grafo
        A[i][i] = 0;           // diagonal a 0
        P[i] = vector<vertice>(n, i); // caminos directos
    }
    // Calcular costes mínimos y caminos correspondientes
    // entre cualquier par de vértices i, j
    for (vertice k = 0; k <= n-1; k++)
        for (vertice i = 0; i <= n-1; i++)
            for (vertice j = 0; j <= n-1; j++) {
                tCoste ikj = suma(A[i][k], A[k][j]);
                if (ikj < A[i][j]) {
                    A[i][j] = ikj;
                    P[i][j] = k;
                }
            }
    return A;
}

```



$$A_k[i,j] = \min \{A_{k-1}[i,j], A_{k-1}[i,k] + A_{k-1}[k,j]\}$$

```

#include "listaenla.h"

template <typename T> class GrafoP {
public:
    typedef Lista<vertice> tCamino;
    // ...
};

template <typename tCoste> typename GrafoP<tCoste>::tCamino
caminoAux(typename GrafoP<tCoste>::vertice v,
          typename GrafoP<tCoste>::vertice w,
          const matriz<typename GrafoP<tCoste>::vertice>& P)
// Devuelve el camino de coste mínimo entre v y w, excluidos estos,
// a partir de una matriz P obtenida mediante la función Floyd().
{
    typename GrafoP<tCoste>::tCamino C1, C2;
    typename GrafoP<tCoste>::vertice u;

    u = P[v][w];
    if (u != v) {
        C1 = caminoAux<tCoste>(v, u, P);
        C1.insertar(u, c1.fin());
        C2 = caminoAux<tCoste>(u, w, P);
        C1 += C2; // Lista<vertice>::operator +=(), concatena C1 y C2
    }
    return C1;
}

```

```
template <typename tCoste> typename GrafoP<tCoste>::tCamino
camino(typename GrafoP<tCoste>::vertice v,
       typename GrafoP<tCoste>::vertice w,
       const matriz<typename GrafoP<tCoste>::vertice>& P)
// Devuelve el camino de coste mínimo desde v hasta w a partir
// de una matriz P obtenida mediante la función Floyd().
{
    typename GrafoP<tCoste>::tCamino C;

    C = caminoAux<tCoste>(v, w, P);
    C.insertar(v, C.primer());
    C.insertar(w, C.fin());
    return C;
}
```

Algoritmos de caminos de coste mínimo

Algoritmo de Warshall

```
matriz<bool> Warshall(const Grafo& G)
```

Determina si hay un camino entre cada par de vértices del grafo no ponderado G.

Salida:

- Una matriz booleana cuadrada de tamaño **G.numVert()**, tal que una posición **[i][j]** es **true** si existe al menos un camino entre el vértice **i** y el vértice **j**, y **false** si no existe ningún camino entre estos vértices.

```

#include "matriz.h"

matriz<bool> Warshall(const Grafo& G)
{
    typedef Grafo::vertice vertice;
    const size_t n = G.numVert();
    matriz<bool> A(n);

    // Inicializar A con la matriz de adyacencia de G
    for (vertice i = 0; i <= n-1; i++) {
        A[i] = G[i];
        A[i][i] = true;
    }
    // Comprobar camino entre cada par de vértices i, j
    // a través de cada vértice k
    for (vertice k = 0; k <= n-1; k++)
        for (vertice i = 0; i <= n-1; i++)
            for (vertice j = 0; j <= n-1; j++)
                if (!A[i][j])
                    A[i][j] = A[i][k] && A[k][j];
    return A;
}

```