Algoritmos de Floyd y Warshall

Versión 1.0/EDNL

CAMAVINGAGII

2025

ÍNDICE GENERAL

1 Capítulo 1
Floyd

1.1 Definición/Funcionamiento general *3*
1.1.1 Ejemplo explicado *4*
1.2 Código *5*
1.3 Obtención del camino *7*

2 Capítulo 2
Algoritmo de Warshall

2.1 Definición/Funcionamiento general *8*
2.2 Código *9*

CAPÍTULO 1

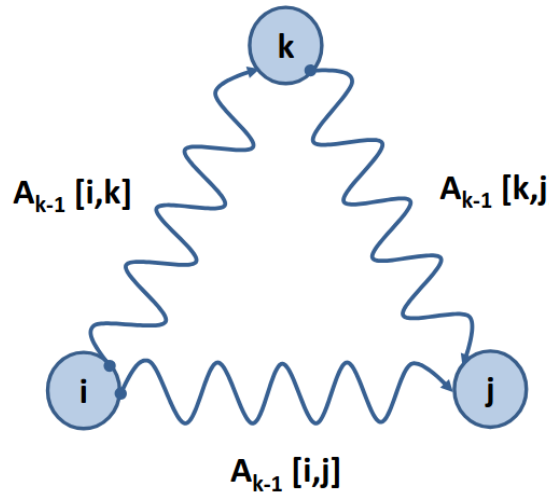
FLOYD

1.1 Definición/Funcionamiento general

- El algoritmo de Floyd es una **generalización del algoritmo de Dijkstra.** Se encarga de **hallar el camino de coste mínimo de todos los nodos con todos los nodos**, usando, como en Dijkstra, un nodo intermedio, en este caso k.
- Por tanto, el algoritmo se basa en, **para cada nodo, hallar la distancia de ese nodo con los demás** usando un **nodo intermedio k**. Si esa distancia es menor, entonces guarda en la matriz el vértice por el que he de pasar para ir de i a j (en la matriz P),y y por otro lado, guardo en la matriz de costes (la que devuelve el algoritmo, A) el coste de ir de i a j (que sería la suma de ir de i a k y de k a j)
- Nota, el algoritmo al principio **inicializa la diagonal principal de la matriz a 0**, ya que el coste de ir de un nodo a sí mismo es 0. Cada fila de la matriz a devolver se inicializa en un principio con los costes de ir de cada nodo a todos los demás nodos (esto se obtiene de la matriz de costes.)

1.1.1. Ejemplo explicado

- Veamos el siguiente ejemplo:



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

Figura 1.1: Ejemplo explicado

- Esta imagen representa la mejora que se ha llevado a cabo para el vértice i utilizando como nodo auxiliar desde el nodo 0 hasta el k-1. Ahora probaremos a mejorar con el nodo k.
- Bien, pues en este caso lo que refleja la imagen es que si ir desde el vértice i al vértice k (A[i,k]) y desde el vértice k al vértice j (A[k,j]) reduce la distancia de ir directamente desde i hasta j A([i,j]), entonces, el valor se actualiza.

1.2 Código

```cpp
/*------------------------------------------------*/
/* 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
```

```
    //algoritmo de Floyd
    #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 minimos
        // Iniciar A y P con caminos directos entre cada par de vertices.
        P = matriz<vertice>(n); //inicializamos la matriz P con un valor
            que indica que no podemos ir en principio a ningun nodo
        for (vertice i = 0; i <= n-1; i++) {
        A[i] = G[i]; // copia costes del grafo
        A[i][i] = 0; // diagonal a 0, coste de ir de un nodo a si mismo
        P[i] = vector<vertice>(n, i); // caminos directos
    }
// Calcular costes minimos y caminos correspondientes
// entre cualquier par de vertices i, j
    for (vertice k = 0; k <= n-1; k++) //vertice que uso para mejorar
    for (vertice i = 0; i <= n-1; i++) //todos los caminos a intentar
        mejorar (origen)
    for (vertice j = 0; j <= n-1; j++)  //destino del camino
    {
    tCoste ikj = suma(A[i][k], A[k][j]); //suma de ir de i a k y de k a j
    if (ikj < A[i][j])
    {
        A[i][j] = ikj; //guardo el nuevo coste de ir de i a j
        P[i][j] = k; //guardo el vertice por el que he de pasar para
            llegar de i a j
    }
    }
    return A;
}
```

1.3 Obtención del camino

- A partir de la matriz obtenida por Floyd, podemos obtener el camino de coste mínimo de ir desde v a w (o de i a j) a partir de los siguientes algoritmos:

```
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 minimo desde v hasta w a partir
    // de una matriz P obtenida mediante la funcion Floyd().
    {
        typename GrafoP<tCoste>::tCamino C;
        C = caminoAux<tCoste>(v, w, P);
        C.insertar(v, C.primera());
        C.insertar(w, C.fin());
        return C;
    }

#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 minimo entre v y w, excluidos estos,
// a partir de una matriz P obtenida mediante la funcion Floyd().
    typename GrafoP<tCoste>::tCamino C1, C2; //listas de vertices
    typename GrafoP<tCoste>::vertice u;
    u = P[v][w];
    if (u != v) //si u es distinto de v significa que se ha usado un nodo
         intermedio para mejorar
    {
        C1 = caminoAux<tCoste>(v, u, P); //insertamos el coste de ir de v
             a u (recursivo)
        C1.insertar(u, c1.fin());
        C2 = caminoAux<tCoste>(u, w, P); //insertamos el coste de ir de u
             a w (recursivo)
        C1 += C2; // Lista<vertice>::operator +=(), concatena C1 y C2
    }
return C1;
}
```

CAPÍTULO

2

ALGORITMO DE WARSHALL

2.1 Definición/Funcionamiento general

- Este algoritmo simplemente **a partir de una matriz de adyacencia de un grafo** (una matriz de adyacencia es una matriz de booleanos en la que en la posición i,j de la matriz hay un true si hay un camino directo de i a j o false en caso contrario), **va probando con nodos intermedios** (siempre y cuando desde i a j no haya un true desde un principio) **y mirando si con esos nodos intermedios es posible ir de i a k y de k a j**. Si es posible, la **posición i,j** de la matriz se pone a **true**. (puede ser que haya un nodo intermedio que minimice el camino de ir desde i a k y desde k a j, pero en este algoritmo no se mira eso, solo se mira si es posible ir desde el nodo i a un nodo intermedio y desde ese nodo al nodo destino)
- **NOTA**: Solo módifico (o intento modificar) la posición i,j de la matriz si no hay camino (si es false), si hay, no hago nada.

2.2 Código

```
#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 vertices i, j
// a traves de cada vertice 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]) //solo modifico si es false (no hay camino)
        A[i][j] = A[i][k] && A[k][j];
return A;
}
```