Algoritmo de Prim

El **algoritmo de Prim** es un [algoritmo](http://es.wikipedia.org/wiki/Algoritmo) perteneciente a la [teoría de los grafos](http://es.wikipedia.org/wiki/Teor%C3%ADa_de_los_grafos) para encontrar un [árbol recubridor mínimo](http://es.wikipedia.org/wiki/%C3%81rbol_recubridor_m%C3%ADnimo) en un [grafo](http://es.wikipedia.org/wiki/Grafo) [conexo](http://es.wikipedia.org/wiki/Teor%C3%ADa_de_los_grafos#Grafos_conexos), **no** dirigido y cuyas [aristas](http://es.wikipedia.org/wiki/Arista_%28Teor%C3%ADa_de_grafos%29) están etiquetadas.

En otras palabras, el [algoritmo](http://es.wikipedia.org/wiki/Algoritmo) encuentra un subconjunto de [aristas](http://es.wikipedia.org/wiki/Arista_%28Teor%C3%ADa_de_grafos%29) que forman un [árbol](http://es.wikipedia.org/wiki/%C3%81rbol_%28teor%C3%ADa_de_grafos%29) con todos los [vértices](http://es.wikipedia.org/wiki/V%C3%A9rtice_%28Teor%C3%ADa_de_grafos%29), donde el peso total de todas las [aristas](http://es.wikipedia.org/wiki/Arista_%28Teor%C3%ADa_de_grafos%29) en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el [árbol recubridor mínimo](http://es.wikipedia.org/wiki/%C3%81rbol_recubridor_m%C3%ADnimo) para uno de los componentes conexos que forman dicho grafo no conexo.

**public** **class** Algorithms

{

**public** **static** Graph PrimsAlgorithm (Graph g, **int** s)

{

**int** n = g.getNumberOfVertices ();

Entry[] table = **new** Entry [n];

**for** (**int** v = 0; v < n; ++v)

table [v] = **new** Entry ();

table [s].distance = 0;

PriorityQueue queue =

**new** BinaryHeap (g.getNumberOfEdges());

queue.enqueue (

**new** Association (**new** Int (0), g.getVertex (s)));

**while** (!queue.isEmpty ())

{

Association assoc = (Association) queue.dequeueMin();

Vertex v0 = (Vertex) assoc.getValue ();

**int** n0 = v0.getNumber ();

**if** (!table [n0].known)

{

table [n0].known = **true**;

Enumeration p = v0.getEmanatingEdges ();

**while** (p.hasMoreElements ())

{

Edge edge = (Edge) p.nextElement ();

Vertex v1 = edge.getMate (v0);

**int** n1 = v1.getNumber ();

Int wt = (Int) edge.getWeight ();

**int** d = wt.intValue ();

**if** (!table[n1].known && table[n1].distance>d)

{ table [n1].distance = d;

table [n1].predecessor = n0;

queue.enqueue (

**new** Association (**new** Int (d), v1));

}

}

}

}

Graph result = **new** GraphAsLists (n);

**for** (**int** v = 0; v < n; ++v)

result.addVertex (v);

**for** (**int** v = 0; v < n; ++v)

**if** (v != s)

result.addEdge (v, table [v].predecessor);

**return** result;

}

}

Algoritmo de Dijkstra

El **algoritmo de Dijkstra**, también llamado **algoritmo de caminos mínimos**, es un [algoritmo](http://es.wikipedia.org/wiki/Algoritmo) para la determinación del [camino más corto](http://es.wikipedia.org/wiki/Problema_de_los_caminos_m%C3%A1s_cortos) dado un [vértice](http://es.wikipedia.org/wiki/V%C3%A9rtice_%28Teor%C3%ADa_de_grafos%29) origen al resto de vértices en un [grafo](http://es.wikipedia.org/wiki/Grafo) dirigido y con pesos en cada [arista](http://es.wikipedia.org/wiki/Arista_%28Teor%C3%ADa_de_grafos%29).

#include <cmath>

#include <cstring>

#include <iostream>

using namespace std;

int destino, origen, vertices = 0;

int \*costos = NULL;

void dijkstra(int vertices, int origen, int destino, int \*costos) {

int i, v, cont = 0;

int \*ant, \*tmp;

int \*z; */\* vertices para los cuales se conoce el camino minimo \*/*

double min;

double \*dist = new double[vertices]; */\* vector con los costos de dos caminos \*/*

*/\* aloca las lineas de la matriz \*/*

ant = new int[vertices];

tmp = new int[vertices];

z = new int[vertices];

for (i = 0; i < vertices; i++) {

if (costos[(origen - 1) \* vertices + i] !=- 1) {

ant[i] = origen - 1;

dist[i] = costos[(origen-1)\*vertices+i];

}

else {

ant[i]= -1;

dist[i] = HUGE\_VAL;

}

z[i]=0;

}

z[origen-1] = 1;

dist[origen-1] = 0;

*/\* Bucle principal \*/*

do {

*/\* Encontrando el vertice que debe entrar en z \*/*

min = HUGE\_VAL;

for (i=0;i<vertices;i++)

if (!z[i])

if (dist[i]>=0 && dist[i]<min) {

min=dist[i];v=i;

}

*/\* Calculando las distancias de los nodos vecinos de z \*/*

if (min != HUGE\_VAL && v != destino - 1) {

z[v] = 1;

for (i = 0; i < vertices; i++)

if (!z[i]) {

if (costos[v\*vertices+i] != -1 && dist[v] + costos[v\*vertices+i] < dist[i]) {

dist[i] = dist[v] + costos[v\*vertices+i];

ant[i] =v;

}

}

}

} while (v != destino - 1 && min != HUGE\_VAL);

*/\* Muestra el resultado de la búsqueda \*/*

cout << "**\t**De " << origen << " para "<<destino<<" **\t**";

if (min == HUGE\_VAL) {

cout <<"No Existe**\n**";

cout <<"**\t**Coste: **\t**- **\n**";

}

else {

i = destino;

i = ant[i-1];

while (i != -1) {

// printf("<-%d",i+1);

tmp[cont] = i+1;

cont++;

i = ant[i];

}

for (i = cont; i > 0; i--) {

cout<< tmp[i-1]<<" -> ";

}

cout << destino;

cout <<"**\n\t**Coste: " << dist[destino-1] <<"**\n**";

}

delete (dist);

delete (ant);

delete (tmp);

delete (z);

}

int menu(void) {

int opcion;

cout <<" Implementacion del Algoritmo de Dijkstra**\n**";

cout <<" Menu:**\n**";

cout <<" >> 1. Crear el grafo**\n** >> 2. Determinar el menor camino del grafo**\n** >> 0. Salir del programa**\n**";

cout <<endl;

cout << " Opcion: ";

cin>>opcion;

while(opcion<0 || opcion>2){

cout<<" Opcion Invalida. Digitela nuevamente: ";

cin>>opcion;

}

return opcion;

}

void add(void) {

do {

cout <<"**\n**Ingrese el numero de vertices ( no minimo de 2 ): ";

cin>>vertices;

} while (vertices < 2 );

if (!costos)

delete(costos);

costos = new int[vertices \* vertices];

for (int i = 0; i <= vertices \* vertices; i++)

costos[i] = -1;

cout <<" Nº Vertices = "<< vertices<<endl;

cout <<"Ahora unamos los vertices:**\n**" ;

bool sigo=true;

int origen;

int destino;

while (sigo){

cout << " Escoja el primer vertice de la arista: " <<endl;

do{

cin >> origen;

if (origen>vertices){

cout << " El numero del vertice debe ser menor de " << vertices<<endl;

}

}while(origen > vertices);

cout << " Escoja el segundo vertice de la arista: " <<endl;

do{

cin >> destino;

if (destino>vertices){

cout << " El numero de vertice debe ser menor de " << vertices<<endl;

}

}while(destino> vertices);

int peso=0;

cout <<" Peso: " <<endl;

cin>>peso;

costos[(origen-1) \* vertices + destino - 1] = peso;

costos[(destino-1) \* vertices + origen - 1] = peso;

int seguir=1;

cout << "Desea anadir otra arista? (0 - NO, 1 - SI, por defecto 1): " ;

cin >>seguir;

sigo = (seguir==1);

}

}

void buscar(void) {

int i, j;

cout <<" Lista de los Menores Caminos en Grafo Dado: **\n**";

for (i = 1; i <= vertices; i++) {

for (j = 1; j <= vertices; j++)

dijkstra(vertices, i,j, costos);

cout<<endl;

}

cout <<"<Presione ENTER para volver al menu principal. **\n**";

}

int main(int argc, char \*\*argv) {

int opcion;

do {

opcion = menu();

switch(opcion) {

case 1:

add();

break;

case 2:

buscar();

break;

}

} while (opcion!= 0);

delete(costos);

cout<<"**\n**Hasta la proxima...**\n\n**";

system("pause");

return 0;

}

# Algoritmo de Floyd-Warshall

En [informática](zim://A/Inform%C3%A1tica.html), el **algoritmo de Floyd-Warshall**, descrito en 1959 por Bernard Roy, es un [algoritmo](zim://A/Algoritmo.html) de análisis sobre [grafos](zim://A/Grafo.html) para encontrar el [camino mínimo](zim://A/Problema_de_los_caminos_m%C3%A1s_cortos.html) en grafos dirigidos ponderados. El algoritmo encuentra el camino entre todos los pares de vértices en una única ejecución. El algoritmo de Floyd-Warshall es un ejemplo de [programación dinámica](zim://A/Programaci%C3%B3n_din%C3%A1mica_%28computaci%C3%B3n%29.html).

## Código en C++

// Declaraciones en el archivo .h

int cn; //cantidad de nodos

vector< vector<int> > ady;

// Devuelve una matriz con las distancias minimas de cada nodo al resto de los vertices.

vector< vector<int> > Grafo :: floydWarshall(){

vector< vector<int> > path = this->ady;

/\* No tendría porqué ser necesario si ya hay ceros en las diagonales de ady \*/

for(int i = 0; i < cn; i++)

path[i][i] = 0;

for(int k = 0; k < cn; k++)

for(int i = 0; i < cn; i++)

for(int j = 0; j < cn; j++){

int dt = path[i][k] + path[k][j];

if(path[i][j] > dt)

path[i][j] = dt;

}

return path;

}

**/\*Ordenacion Rapida o QuickSort**

**El mejor método para ordenar arrays conocido hasta el momento.**

**\*En la mayoría de los casos su eficiencia es de orden O(n log(n))**

**\*En los peores casos, su eficiencia puede ser de orden O(n2), aunque es muy improbable**

**\*Su eficiencia depende de la elección del pivote**

**\*/**

private static void quickSort(int vector[], int izq, int der) {

int pivote = vector[(izq + der) / 2];

int i = izq, j = der;

do {

while (vector[i] < pivote) {

i++;

}

while (vector[j] > pivote) {

j--;

}

if (i <= j) {

int aux = vector[i];

vector[i] = vector[j];

vector[j] = aux;

i++;

j--;

}

} while (i <= j);

if (izq < j) {

quickSort(vector, izq, j);

}

if (i < der) {

quickSort(vector, i, der);

}

}

public static void quickSort(int [] vector){

quickSort(vector,0,vector.length-1);

}

public static boolean isPrime(int n) {

if (n == 1)

return false;

if (n == 2)

return true;

if (n % 2 == 0)

return false;

for (int i = 3; i \* i <= n; i += 2)

if (n % i == 0)

return false;

return true;

}

public static double logaritmo(double valor, double base) {

return Math.log(valor) / Math.log(base);

}

public static int GCD1(int a, int b) {// MÁXIMO COMUN DIVISOR

BigInteger \_a = BigInteger.valueOf(a);

BigInteger \_b = BigInteger.valueOf(b);

return (\_a.gcd(\_b)).intValue();

}

public static int GCD2(int a, int b) {//OTRO MÁXIMO COMUN DIVISOR

while (b > 0) {

a = a % b;

a ^= b;

b ^= a;

a ^= b;

}

return a;

}

public static int LCM(int a, int b) {// MÍNIMO COMUN MÚLTIPLO

return (a \* b) / GCD1(a, b);

}

public static double raizNsimaDeX(double x, double n){//RAIZ NSIMA DE X

return Math.pow(x, 1.0 / n);

}

**COMO USAR BUFFERDREADER**

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int n = Integer.valueOf(br.readLine());

int n2 = Integer.parseInt(br.readLine());

String ok = br.readLine();

float b = Float.parseFloat(br.readLine());