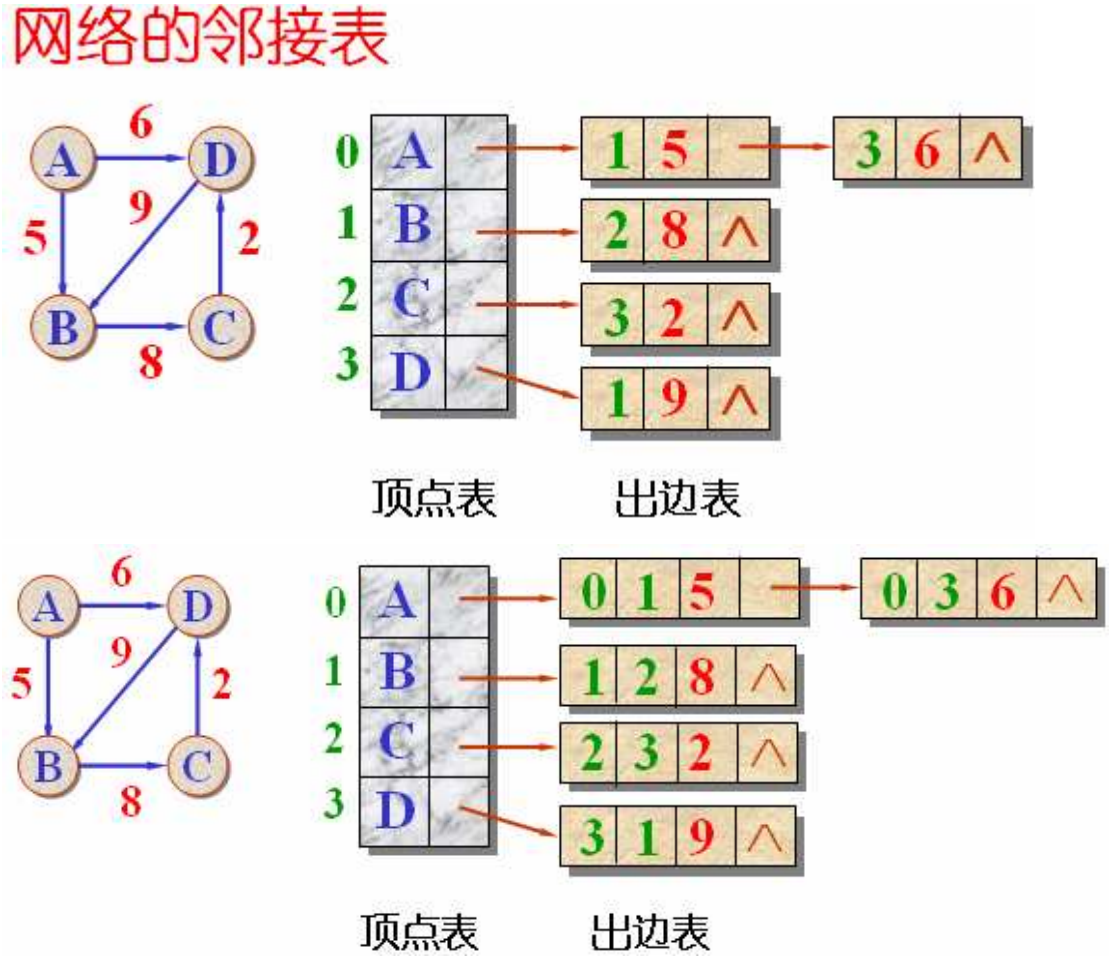
最短路径算法 Dijkstra

一、 图的邻接表存储结构及实现（回顾）



1. 头文件 graph.h

// Graph.h: interface for the Graph class.

#if !defined(AFX_GRAPH_H__C891E2F0_794B_4ADD_8772_55BA367C823E__INCLUDED_)

#define AFX_GRAPH_H__C891E2F0_794B_4ADD_8772_55BA367C823E__INCLUDED_

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <cassert>
#include <vector>
using namespace std;

#define NULL 0
typedef int weightType;
typedef char Type; //数据元素类型
class EdgeNode {                    // A singly-linked list node
public:
   weightType weight;                // Edge weight
   int v1;                           // Vertex edge comes from
   int v2;                           // Vertex edge goes to
   EdgeNode* next;                   // Pointer to next edge in list
   EdgeNode(int vt1, int vt2, weightType w, EdgeNode* nxt =NULL)
      { v1 = vt1; v2 = vt2; weight = w; next = nxt; } // Constructor
   EdgeNode(EdgeNode* nxt =NULL) { next = nxt; }    // Constructor
};
```

```
typedef EdgeNode* Edge;
struct VertexNode{
    Type data;
    Edge first;
    VertexNode(Type d,Edge e):data(d),first(e){}
};
typedef VertexNode Vertex;

class Graph {                    // Graph class: Adjacency list
private:
   vector<Vertex> list;          //The vertex list
   int numEdge;                  // Number of edges
   vector<bool> Mark;            // The mark array
public:
   Graph();                      // Constructor
   ~Graph();                     // Destructor
   int n();                      // Number of vertices for graph
   int e();                      // Number of edges for graph
   Edge first(int);              // Get the first edge for a vertex
   bool isEdge(Edge);            // TRUE if this is an edge
   Edge next(Edge);              // Get next edge for a vertex
   int v1(Edge);                 // Return vertex edge comes from
```

```
    int v2(Edge);                       // Return vertex edge goes to
    weightType weight(int, int);        // Return weight of edge
    weightType weight(Edge);            // Return weight of edge
    bool getMark(int);                  // Return a Mark value
    void setMark(int, bool);            // Set a Mark value
    void setVertex(int i,Type vertexData){
        assert(i>=0&&i<list.size()) ; list[i].data =vertexData; }
    Type getVertex(int i){
        assert(i>=0&&i<list.size()) ; return list[i].data; }
    void InsertVertex ( const Type & vertexData );
    void InsertEdge ( const int v1, const int v2, weightType weight );
    void RemoveVertex ( const int v );
    void RemoveEdge ( const int v1, const int v2 );
};

void Dijkstra_shortest_Path(Graph& G, int s,weightType D[],int P[]);

#endif
// !defined(AFX_GRAPH_H__C891E2F0_794B_4ADD_8772_55BA367
C823E__INCLUDED_)
```

2. cpp 文件 graph.cpp

```
// Graph.cpp: implementation of the Graph class.
```

```
#include "Graph.h"
#define INFINITY 1000000
Graph::Graph() {                    // Constructor
    numEdge = 0;
}

Graph::~Graph() {                   // Destructor: return allocated space
    // Remove all of the edges
    for (int v=0; v<list.size(); v++) { // For each vertex...
        Edge p = list[v].first;
        while (p != NULL) { // return its edges
            Edge temp = p;
            p = p->next;
            delete temp;
        }
    }

}

int Graph::n() { return list.size(); } // Number of vertices
int Graph::e() { return numEdge; }      // Number of edges
```

```
Edge Graph::first(int v)        // Get the first edge for a vertex
{ return list[v].first; }

bool Graph::isEdge(Edge w)    // TRUE if this is an edge
{ return w != NULL; }

Edge Graph::next(Edge w) {    // Get next edge for a vertex
    if (w == NULL) return NULL;
    else return w->next;
}

int Graph::v1(Edge w) { return w->v1; } // Vertex edge comes from

int Graph::v2(Edge w) { return w->v2; } // Vertex edge goes to

weightType Graph::weight(int i, int j) { // Return weight of edge
    for (Edge curr = list[i].first; curr != NULL; curr = curr->next)
        if (curr->v2 == j) return curr->weight;
        return INFINITY;
}

weightType Graph::weight(Edge w) // Return weight of edge
```

```
{ if (w == NULL) return INFINITY; else return w->weight; }

bool Graph::getMark(int v) { return Mark[v]; }
void Graph::setMark(int v, bool val) { Mark[v] = val; }

//----------插入或删除数据----------
void Graph::InsertVertex ( const Type & vertexData ){
    list.push_back( VertexNode(vertexData,NULL) );
    Mark.push_back(false);
}
void Graph::InsertEdge ( const int v1, const int v2, weightType weight ){
    Edge edge= new EdgeNode(v1,v2,weight);
    edge->next = list[v1].first;
    list[v1].first = edge;
    numEdge++;
}
void Graph::RemoveVertex ( const int v ){
}
void Graph::RemoveEdge ( const int v1, const int v2 ){
}
```

3. 测试程序 main.cpp

```
#include <iostream>
#include <stack>
#include "Graph.h"
void main(){
    Graph G;
    Type vdata;
    cout<<"请依次输入顶点数据，用'ctrl+Z' 'ctrl+Z'结束输入\n";
    while(cin>>vdata){
        G.InsertVertex(vdata);
    }
    cin.clear();   //置为正常状态

    int v1 ,v2;
    weightType weight;
    cout<<"请输入边信息(格式为  v1 v2 weight):";
    cin>>v1>>v2>>weight;
    while(v1>=0&&v2>=0){
        G.InsertEdge(v1,v2,weight);
        cout<<"请输入边信息(格式为  v1 v2 weight):";
        cin>>v1>>v2>>weight;
    }
    int i;
```

```
    cout<<"图中顶点数据为:";
    for( i = 0 ;i <G.n(); i++)
        cout<<G.getVertex(i)<<" ";
    cout<<endl;

    cout<<"图中边数据为:";
    for( i = 0 ;i <G.n(); i++){
        Edge edge = G.first(i);
        while(edge){
              cout<<"("<<edge->v1<<" "<<edge->v2<<
                    " "<<edge->weight<<") ";
            edge = G.next(edge);
        }
    }
    cout<<endl;
}
```

二、 Dijkstra 算法

1．Dijkstra 算法（Dijkstra_shortest_Path.cpp）：

```
#include "Graph.h"

#define INFINITY 1000000
const bool VISITED = true;
const bool UNVISITED = false;

// minVertex：在距离数组 D 中找最小的未加入 S 中的最短距离
int minVertex(Graph& G, int* D);


void Dijkstra_shortest_Path(Graph& G, int s,weightType D[],int P[])
{   // Compute shortest path distances
    //初始时，所有顶点都未加入到已经最短路径的顶点集合 S
    int i;
    for(i = 0 ;i< G.n(); i++)   G.setMark(s, UNVISITED);
    //将 s 作为起点，初始化距离数组和路径数组
    for ( i=0; i<G.n(); i++){          // Initialize
        D[i] = INFINITY; P[i] = s;
    }
    D[s] = 0; G.setMark(s, VISITED);           //add s to S
    for (Edge e = G.first(s); G.isEdge(e); e = G.next(e))
        D[G.v2(e)] =    G.weight(e);
```

```
    //在未加入 S 中的顶点中选择最短路径的那个顶点，
    //加入 S,并更新距离和路径数组
    for (i=0; i<G.n()-1; i++) {          //  最多进行 n-1 次
        //在不在 S 中的顶点中查找 D(v)最小的顶点 v
        int v = minVertex(G, D);
        if (D[v] == INFINITY) return;    //  没有可以到达的顶点了
        G.setMark(v, VISITED);
        //更新 v 的所有邻接点 v2 的 D(v2)和 P(v2)
        for (Edge e = G.first(v); G.isEdge(e); e = G.next(e))
            if (D[G.v2(e)] > (D[v] + G.weight(e))){
                D[G.v2(e)] = D[v] + G.weight(e);
                P[G.v2(e)]    = v;      //
            }
    }
    for(i = 0 ;i< G.n(); i++)    G.setMark(s, UNVISITED);
}

int minVertex(Graph& G, int* D) { // Find min cost vertex
    int v;    // Initialize v to any unvisited vertex;
    for (int i=0; i<G.n(); i++)
        if (G.getMark(i) == UNVISITED) { v = i; break; }
    for (i++; i<G.n(); i++)    // Now find smallest D value
```

```
        if ((G.getMark(i) == UNVISITED) && (D[i] < D[v]))
            v = i;
    return v;
}
```

2．修改后的测试程序：

```
#include <iostream>
#include <stack>
#include "Graph.h"
void main(){
    Graph G;
    Type vdata;
    cout<<"请依次输入顶点数据，用'ctrl+Z' 'ctrl+Z'结束输入\n";
    while(cin>>vdata){
        G.InsertVertex(vdata);
    }
    cin.clear();   //置为正常状态

    int v1 ,v2;
    weightType weight;
    cout<<"请输入边信息(格式为  v1 v2 weight):";
    cin>>v1>>v2>>weight;
```

```
    while(v1>=0&&v2>=0){
        G.InsertEdge(v1,v2,weight);
        cout<<"请输入边信息(格式为  v1 v2 weight):";
        cin>>v1>>v2>>weight;
    }

    int i;
    cout<<"图中顶点数据为:";
    for( i = 0 ;i <G.n(); i++)
        cout<<G.getVertex(i)<<" ";
    cout<<endl;

    cout<<"图中边数据为:";
    for( i = 0 ;i <G.n(); i++){
        Edge edge = G.first(i);
        while(edge){
            cout<<"("<<edge->v1<<"                    "<<edge->v2<<"
"<<edge->weight<<") ";
            edge = G.next(edge);
        }
    }
    cout<<endl;
```

```
//Dijkstra_shortest_Path 算法
weightType *D = new weightType[G.n()];
int *P = new int[G.n()];
int s;
cout<<"请输入起点的下标:";     cin>>s;
Type sdata = G.getVertex(s);
Dijkstra_shortest_Path(G, s, D, P);

cout<<"输出所有最短路径,格式（终点，最短路径长度，最短路径）\n";
stack<int> pathStack;
Type vertexdata;
for( i = 0; i<G.n(); i++){
    if(i==s) continue;
    vertexdata = G.getVertex(i);
    if(D[i]>=100000){
        cout <<"起点"<<sdata<<"到顶点"<<vertexdata
            <<"没有路径\n";
        continue;
    }
    for(int j = P[i];j!=s;j = P[j])
```

```
            pathStack.push(j);
        cout<<vertexdata<<" "<<D[i]<<" "<<sdata;

        while(!pathStack.empty()){
            int j = pathStack.top(); pathStack.pop();
            cout<<G.getVertex(j);
        }
        cout<<G.getVertex(i)<<"\n";
    }

    delete[] D; delete[] P;

}
```