Dijkstra算法

20123211 高雨晴

0. 概述

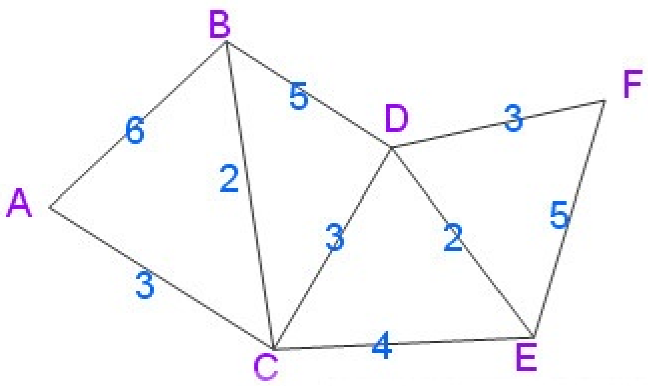
- 实验内容：实现Dijkstra算法。
- 实验方法：使用C++语言，通过MinGW-w64 C/C++编译器
- 实验结果：输出结果和预期相符。

1. 实验设计

1.1 算法实现原理

Dijkstra算法通过保留目前为止所找到的每个顶点v和从出发点s到v的最短路径来工作。初始时，原点s的路径权重被赋为0，同时把所有其他顶点的路径长度设为无穷大，即表示我们不知道任何通向这些顶点的路径。当算法结束时，d[v]中储存的时从s到v的最短路径，或者路径不存在的话是无穷大。其伪代码可以如下表示：

```
function Dijkstra(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)              //实际上的操作是将每个除原点外的顶点的d[v]置为无穷大，d[s] = 0
  S ← ∅
  Q ← s                                       //Q是顶点V的一个优先队列，以顶点的最短路径估计排序
  while(Q ≠ ∅)
        do u ← EXTRACT − MIN(Q)               //选取u为Q中最短路径估计最小的顶点
     S ← S ∪ u
     for each vertex v ∈ Adj[u]
              do RELAX(u, v, w)               //松弛成功的结点会被加入到队列中
```



以这张图为例，假设从A出发，目的地为F。

- 首先，假设A距离其他点的距离都是无穷大，而A距离自己的距离是0，因此标记A。
- 考虑A连接的B、C两点，距离分别为6和3. 3比6更小，所以标记C。
- 考虑C连接的除了A以外的节点。C-B: 2, C-D: 3, C-E: 4。从A出发经过C再到B的路程是5，小于A直接到B的6，因此更新A-B之间距离5. 从A经过C到节点D的路程是6，小于无穷大，因此更新A-D间距离为6. 以此类推，A-E 间距离为7。在未标记的节点中考虑距离最小的点，为B。因此标记B。
- 重复以上过程，直至目的地节点被标记。再根据目的地此时记录的最小距离，回溯原来的路径即可。

- 经过计算后可以得知，从A到F的最短路径是A,C,D,F, 最小代价是9.

1.2 算法设计与分析

```
void dijkstra(int G[MAX_V][MAX_V],int n,int startnode) {
   int cost[MAX_V][MAX_V],distance[MAX_V],pred[MAX_V];
   int visited[MAX_V],count,mindistance,nextnode,i,j;
   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
   if(G[i][j]==0)
      cost[i][j]=INF;   // 防止从本点访问本点
   else
      cost[i][j]=G[i][j];
   for(i=0;i<n;i++) {
      distance[i]=cost[startnode][i];
      pred[i]=startnode;
      visited[i]=0;
   }
   distance[startnode]=0;
   visited[startnode]=1;
   count=1;
   while(count<n-1) {
      mindistance=INF;
      for(i=0;i<n;i++)
         if(distance[i]<mindistance&&!visited[i]) {
         mindistance=distance[i];
         nextnode=i;
      }
      visited[nextnode]=1;
      for(i=0;i<n;i++)
         if(!visited[i])
      if(mindistance+cost[nextnode][i]<distance[i]) {
         distance[i]=mindistance+cost[nextnode][i];
         pred[i]=nextnode;
      }
      count++;
   }
   for(i=0;i<n;i++)
   if(i!=startnode) {
      cout<<"\nDistance of node"<<i<<"="<<distance[i];
      cout<<"\nPath="<<i;
      j=i;
      do {
         j=pred[j];
         cout<<"<-"<<j;
      }while(j!=startnode);
   }
}
```

2. 实验结果说明与分析

2.1 实验设置

首先写出我们例子中的图的邻接表

	A	B	C	D	E	F
A	0	6	3			
B	6	0	2	5		
C	3	2	0	3	4	
D		5	3	0	2	3
E			4	2	0	5
F				3	5	0

在主函数中调用dijkstra算法。

```
void dijkstra(int G[MAX_V][MAX_V],int n,int startnode);
int main() {
   int G[MAX_V][MAX_V]={
        {  0,  6,  3,INF,INF,INF},
        {  6,  0,  2,  5,INF,INF},
        {  3,  2,  0,  3,  4,INF},
        {INF,  5,  3,  0,  2,  3},
        {INF,INF,  4,  2,  0,  5},
        {INF,INF,INF,  3,  5,  0}
    };
   int n=6;
   int u=0;
   dijkstra(G,n,u);
   system("pause");
   return 0;
}
```

2.2 实验结果以及相应的分析

输出结果:

```
Distance of node1=5
Path=1<-2<-0
Distance of node2=3
Path=2<-0
Distance of node3=6
Path=3<-2<-0
Distance of node4=7
Path=4<-2<-0
Distance of node5=9
Path=5<-3<-2<-0
```

故从A到F的最小路径是A-C-D-F，距离是9. 这和我们一开始得出的结果是一样的。

3. 总结

对于没有任何优化的戴克斯特拉算法，实际上等价于每次遍历了整个图的所有结点来找到Q中满足条件的元素（即寻找最小的顶点是$O(|V|)$的），此外实际上还需要遍历所有的边一遍，因此算法的复杂度是$O(|V|^2+|E|)$。

4. 附录

```
#include<iostream>
#include<stdio.h>
using namespace std;
#define INF 32767
#define MAX_V 6
void dijkstra(int G[MAX_V][MAX_V],int n,int startnode);
int main() {
   int G[MAX_V][MAX_V]={
        {  0,  6,  3,INF,INF,INF},
        {  6,  0,  2,  5,INF,INF},
        {  3,  2,  0,  3,  4,INF},
        {INF,  5,  3,  0,  2,  3},
        {INF,INF,  4,  2,  0,  5},
        {INF,INF,INF,  3,  5,  0}
    };
   int n=6;
   int u=0;
   dijkstra(G,n,u);
   system("pause");
   return 0;
}
void dijkstra(int G[MAX_V][MAX_V],int n,int startnode) {
   int cost[MAX_V][MAX_V],distance[MAX_V],pred[MAX_V];
   int visited[MAX_V],count,mindistance,nextnode,i,j;
   for(i=0;i<n;i++)
      for(j=0;j<n;j++)
   if(G[i][j]==0)
      cost[i][j]=INF;
   else
      cost[i][j]=G[i][j];
   for(i=0;i<n;i++) {
      distance[i]=cost[startnode][i];
      pred[i]=startnode;
      visited[i]=0;
   }
   distance[startnode]=0;
   visited[startnode]=1;
   count=1;
   while(count<n-1) {
      mindistance=INF;
      for(i=0;i<n;i++)
         if(distance[i]<mindistance&&!visited[i]) {
         mindistance=distance[i];
         nextnode=i;
      }
      visited[nextnode]=1;
```

```
      for(i=0;i<n;i++)
         if(!visited[i])
      if(mindistance+cost[nextnode][i]<distance[i]) {
         distance[i]=mindistance+cost[nextnode][i];
         pred[i]=nextnode;
      }
      count++;
   }
   for(i=0;i<n;i++)
   if(i!=startnode) {
      cout<<"\nDistance of node"<<i<<"="<<distance[i];
      cout<<"\nPath="<<i;
      j=i;
      do {
         j=pred[j];
         cout<<"<-"<<j;
      }while(j!=startnode);
   }
}
```