**算法分析与设计实验报告**

**第 3 次实验**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 杨杰 | 学号 | 201908010705 | | 班级 | 计科1907 |
| 时间 | 5.20 | 地点 | 软件大楼 | | | |
| 实验名称 | 用Dijkstra贪心算法求解单源最短路径问题 | | | | | |
| 实验目的 | 通过上机实验，要求掌握用Dijkstra贪心算法求解单源最短路径的问题描述、算法设计思想、程序设计。 | | | | | |
| 实验原理 | 通过Dijkstra计算图G中的最短路径时，需要指定源点s(即从顶点s开始计算)。此外，引进两个集合S和U。S的作用是记录已求出最短路径的顶点(以及相应的最短路径长度)，而U则是记录还未求出最短路径的顶点(以及该顶点到源点s的距离)。  初始时，S中只有源点s，U中是除s之外的顶点，并且U中顶点的路径是源点s到该顶点的路径。然后，从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点对应的路径。 然后，再从U中找出路径最短的顶点，并将其加入到S中；接着，更新U中的顶点对应的路径。重复该操作，直到所有顶点都加入S中。 | | | | | |
| 实验步骤 | **(1)**初始时，S只包含源点s；U包含除s外的其他顶点，且U中顶点的距离为源点s到该顶点的距离[例如，U中顶点v的距离为(s,v)的长度，若s和v不相邻，则v的距离为∞]。  **(2)**从U中选出距离最短的顶点k，并将顶点k加入到S中；同时，从U中移除顶点k。  **(3)**更新U中各个顶点到源点s的距离。之所以更新U中顶点的距离，是由于上一步中确定了k是求出最短路径的顶点，从而可以利用k来更新其它顶点的距离。例如，(s,v)的距离可能大于(s,k)+(k,v)的距离。  **(4)**重复步骤(2)和(3)，直到所有顶点都加入S中。 | | | | | |
| 关键代码 | *//最短路径 - Dijkstra算法 参数：图G、源点v*  void Dijkstra(Graph G, int v)  {  *//初始化*      int n = G.vexnum;*//n为图的顶点个数*      for (int i = 0; i < n; i++)      {          S[i] = false;          D[i] = G.Edge[v][i];          if (D[i] < INF)              Pr[i] = 0;*//v与i连接，v为前驱*          else              Pr[i] = -1;      }      S[v] = true;      D[v] = 0;  *//初始化结束,求最短路径，并加入S集*      for (int i = 1; i < n; i++)      {          int min = INF;          int temp;          for (int w = 0; w < n; w++)              if (!S[w] && D[w] < min)*//某点temp未加入s集，且为当前最短路径*              {                  temp = w;                  min = D[w];              }          S[temp] = true;  *//更新从源点出发至其余点的最短路径 通过temp*          for (int w = 0; w < n; w++)              if (!S[w] && D[temp] + G.Edge[temp][w] < D[w])              {                  D[w] = D[temp] + G.Edge[temp][w];                  Pr[w] = temp;              }      }  }  *//输出最短路径*  void Path(Graph G, int v, ofstream &outfile)  {      if (Pr[v] == 0)      {          outfile << "->" << G.Vex[v];      }      else      {          Path(G, Pr[v], outfile);          outfile << "->" << G.Vex[v];      }  } | | | | | |
| 算法复杂度分析 | Dijkstra函数中两层for循环嵌套，外层for循环执行n-1次，内层for循环执行n次，循环体时间复杂度为O(1)，所以总时间复杂度为O(n2).  Path函数时间复杂度为O(1).  算法需要一个二维数组来存储邻接矩阵，所以空间复杂度为O(n2). | | | | | |
| 测试结果  （含运行时间） | **小规模数据**    **中规模数据**      **大规模数据** | | | | | |
| 实验心得 | 贪心算法是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，算法得到的是在某种意义上的局部最优解。  贪心算法不是对所有问题都能得到整体最优解，关键是贪心策略的选择。朴素的Dijkstra算法时间复杂度为O(n2),可以使用堆优化来降低时间复杂度。  代码如下：  */\**  *O(eloge)堆优化dj算法，在n的数量级>=1e5时必须采用这种堆优化+邻接表方式*  *\*/*  struct node  {      int p, w;      node(int a, int b) : p(a), w(b) {}      bool operator<(const node &b) const      {          return w > b.w;      }  };  vector<node> g[N];  priority\_queue<node> sup;  void dijkstra(int start)  {      memset(dis, 0x3f, sizeof(dis));      dis[start] = 0;      pre[start] = start;      sup.push(node(start, 0));      while (!sup.empty())      {          node front = sup.top();          sup.pop();          int tempv = front.p;          if (visit[tempv])              continue;          visit[tempv] = true;          for (int i = 0; i < g[tempv].size(); i++)          {              int p = g[tempv][i].p;              if (!visit[p] && dis[tempv] + g[tempv][i].w < dis[p])              {                  dis[p] = dis[tempv] + g[tempv][i].w;                  pre[p] = tempv;                  sup.push(node(p, dis[p]));              }          }      }  }  还有一个需要特别注意的地方，开始的时候我在main函数中是这样写的：    但是当我把结点个数的宏定义改为1000时，我发现无论怎样函数都没有输出了，我以为是文件读写出了问题，但是不管用cout还是printf都没有输出，而当我把Graph G;这行代码注释掉以后，一切又都正常了，所以我断定问题出在这行代码上，经过仔细思考，我发现原来是宏定义太大导致二维数组太大，而局部变量G是存储在栈中的，太大会导致栈爆掉，程序不能正常运行。所以有两种解决办法，一个是将G改为全局变量，一个是用new开出一段空间。    经过这次实验，我对于用Dijkstra贪心算法求解单源最短路径问题的相关代码已基本熟悉，算法知识得到了复习与巩固。在写代码与调试的过程中，在解决问题过程中，丰富了个人编程的经历和经验，提高了个人解决问题的能力。 | | | | | |
| 实验得分 |  | 助教签名 | |  | | |

**附录：完整代码**

*//用Dijkstra贪心算法求解单源最短路径问题*

#include <cstdlib>

#include <string.h>

#include <iostream>

#include <fstream>

#include <windows.h>

#define MaxVerNum 1000*//顶点最大数目值*

#define VexType int*//顶点数据类型*

#define EdgeType int*//边数据类型,无向图时邻接矩阵对称，有权值时表示权值，没有时1连0不连*

#define INF 0x3f3f3f3f*//作为最大值*

using namespace std;

*//图的数据结构*

typedef struct Graph

{

    VexType Vex[MaxVerNum];*//顶点表*

    EdgeType Edge[MaxVerNum][MaxVerNum];*//边表*

    int vexnum, arcnum;*//顶点数、边数*

} Graph;

*//迪杰斯特拉算法全局变量*

bool S[MaxVerNum];*//顶点集*

int D[MaxVerNum];*//到各个顶点的最短路径*

int Pr[MaxVerNum];*//记录前驱*

*//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*基本操作函数\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//*

*//初始化函数 参数：图G 作用：初始化图的顶点表，邻接矩阵等*

void InitGraph(Graph &G)

{

    memset(G.Vex, '#', sizeof(G.Vex));*//初始化顶点表*

*//初始化边表*

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

        for (int j = 0; j < MaxVerNum; j++)

            G.Edge[i][j] = INF;

    G.vexnum = G.arcnum = 0;*//初始化顶点数、边数*

}

*//打印图的顶点表*

void PrintVex(Graph G, ofstream &outfile)

{

    for (int i = 0; i < G.vexnum; i++)

    {

        outfile << G.Vex[i] << " ";

    }

    outfile << endl;

}

*//打印图的边矩阵*

void PrintEdge(Graph G, ofstream &outfile)

{

    for (int i = 0; i < G.vexnum; i++)

    {

        for (int j = 0; j < G.vexnum; j++)

        {

            if (G.Edge[i][j] == INF)

                outfile << "∞ ";

            else

                outfile << G.Edge[i][j] << " ";

        }

        outfile << endl;

    }

}

*//创建图功能实现函数 参数：图G  InsertNode 作用：创建图*

void CreateGraph(Graph &G, ifstream &infile, ofstream &outfile)

{

    VexType v, w;

    outfile << "请输入顶点数目:" << endl;

    infile >> G.vexnum;

    outfile << G.vexnum << endl;

    outfile << "请输入所有顶点名称:" << endl;

    for (int i = 0; i < G.vexnum; i++)

    {

        infile >> G.Vex[i];

        outfile << G.Vex[i] << ' ';

    }

    outfile << endl;

    outfile << "请输入邻接矩阵:" << endl;

    for (int i = 0; i < G.vexnum; i++)

    {

        for (int j = 0; j < G.vexnum; j++)

        {

            infile >> G.Edge[i][j];

            outfile << G.Edge[i][j] << ' ';

            if (G.Edge[i][j] == -1)

                G.Edge[i][j] = INF;

        }

        outfile << endl;

    }

    outfile << "图的顶点表为:\n";

    PrintVex(G, outfile);

    outfile << "图的边矩阵为:\n";

    PrintEdge(G, outfile);

}

*//最短路径 - Dijkstra算法 参数：图G、源点v*

void Dijkstra(Graph G, int v)

{

*//初始化*

    int n = G.vexnum;*//n为图的顶点个数*

    for (int i = 0; i < n; i++)

    {

        S[i] = false;

        D[i] = G.Edge[v][i];

        if (D[i] < INF)

            Pr[i] = 0;*//v与i连接，v为前驱*

        else

            Pr[i] = -1;

    }

    S[v] = true;

    D[v] = 0;

*//初始化结束,求最短路径，并加入S集*

    for (int i = 1; i < n; i++)

    {

        int min = INF;

        int temp;

        for (int w = 0; w < n; w++)

            if (!S[w] && D[w] < min)*//某点temp未加入s集，且为当前最短路径*

            {

                temp = w;

                min = D[w];

            }

        S[temp] = true;

*//更新从源点出发至其余点的最短路径 通过temp*

        for (int w = 0; w < n; w++)

            if (!S[w] && D[temp] + G.Edge[temp][w] < D[w])

            {

                D[w] = D[temp] + G.Edge[temp][w];

                Pr[w] = temp;

            }

    }

}

*//输出最短路径*

void Path(Graph G, int v, ofstream &outfile)

{

    if (Pr[v] == 0)

    {

        outfile << "->" << G.Vex[v];

    }

    else

    {

        Path(G, Pr[v], outfile);

        outfile << "->" << G.Vex[v];

    }

}

*//调用最短路径-Dijkstra算法 参数：图G、源点v*

void Shortest\_Dijkstra(Graph &G, ifstream &infile, ofstream &outfile)

{

    int vname;

    int v = -1;

    outfile << "请输入源点名称:" << endl;

    infile >> vname;

    outfile << vname << endl;

    for (int i = 0; i < G.vexnum; i++)

        if (G.Vex[i] == vname)

        {

            v = i;

            break;

        }

    Dijkstra(G, v);

    outfile << "目标点 最短路径值 最短路径" << endl;

    for (int i = 0; i < G.vexnum; i++)

    {

        if (i != v)

        {

            if (D[i] != INF)

            {

                outfile << G.Vex[i] << "      " << D[i] << "        ";

                outfile << G.Vex[v];

                Path(G, i, outfile);

            }

            else

                outfile << G.Vex[i] << "      INF";

            outfile << "\n";

        }

    }

}

int main()

{

    Graph \*G=new Graph();

    ifstream infile("Dijkstra\_in.txt", ios::in);

    ofstream outfile("Dijkstra\_out.txt", ios::out);

    InitGraph(\*G);

    CreateGraph(\*G, infile, outfile);

    double time = 0;

    LARGE\_INTEGER nFreq, nBeginTime, nEndTime;

    QueryPerformanceFrequency(&nFreq);

    QueryPerformanceCounter(&nBeginTime);*//开始计时*

    Shortest\_Dijkstra(\*G, infile, outfile);

    QueryPerformanceCounter(&nEndTime);*//停止计时*

    time = (double)(nEndTime.QuadPart - nBeginTime.QuadPart) / (double)nFreq.QuadPart;*//计算程序执行时间单位为s*

    outfile << "程序耗时" << time \* 1000 << "ms" << endl;

    infile.close();

    outfile.close();

    return 0;

}

*//Dijkstra单源最短路径测试数据生成器*

#include <iostream>

#include <fstream>

#include <time.h>

#define random(a, b) (rand() % (b - a + 1) + a)

using namespace std;

int main()

{

    ofstream outfile("Dijkstra\_in.txt", ios::out);

    srand((int)time(NULL));

    int n;

    cin >> n;

    outfile << n << endl;

    for (int i = 1; i <= n; i++)

        outfile << i << " ";

    outfile << endl;

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < n; j++)

            if (i == j)

                outfile << 0 << " ";

            else

                outfile << random(1, 50) << " ";

        outfile << endl;

    }

    outfile << 1;

    outfile.close();

    return 0;

}