**《（课程名称）》实验报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **2019级计算机科学与技术卓越02班** | | | **姓名** | **李燕琴** |
| **实验题目** | 图的操作 | | | | | |
| **实验时间** | **2020/11/17** | | **实验地点** | **DS1401** | | |
| **实验成绩** |  | | **实验性质** | **□验证性 √设计性 □综合性** | | |
| 教师评价：  □算法/实验过程正确； □源程序/实验内容提交 □程序结构/实验步骤合理；  □实验结果正确； □语法、语义正确； □报告规范；  其他：  评价教师签名： | | | | | | |
| 一、实验目的  1. 掌握图的基本概念，描述方法；遍历方法。  2. 掌握图的最短路径和最小生成树算法。 | | | | | | |
| 1. 实验项目内容   1、图的存储结构使用邻接矩阵。  2、创建图操作类，支持BFS遍历、DFS遍历、求单源最短路径、求最小生成树、判断是否存在回路等五个功能，这些功能封装成图操作类的成员函数。  3、输入图的节点数n（不超过10个）、边数m，节点分别用0到n-1表示。  4、采用“起始节点，终止节点，权值”输入图的m条边，创建图。  5、输出从节点0开始的BFS遍历、DFS遍历节点遍历顺序。  6、输出从节点0到其余节点的最短路径及最短路径长度，如果没有路经，输出0。  7、输出图的最小生成树包含的边，边用“起始节点，终止节点，权值”表示。  注意：最后提交完整的实验报告和源程序，主程序里应对实验所要求的功能进行充分测试。 | | | | | | |
| 三、实验过程或算法（源程序）  1、BFS借助树的层次遍历的思想，通过队列实现。具体算法如下：    在每一个功能函数的开头，最好都进行一次初始化数据，如这里的mark重置。  2、DFS递归实现。具体算法如下：    其中mark数组重置，需要在第一次DFS的时候重置，过程中有在这里摔了跟头，debug了很久。  3、单源最短路径，使用Dijkstra算法，并在每次更新数据的时候    记录影响该数据的节点即parent[i]=v; 进而使用堆栈输出数据，即为s到i的最短路径。数据初始化时，别忘了将parent[i]=s；  4、最小生成树的路径输出：    使用prim算法，并记录最小生成树涉及到的每一个边（u,v,w），按照题目要求，对边进行优先降序排序（本程序使用选择排序），优先级为w>u>v。  5、无向图的有无环分析  基于有向图拓扑排序，对无向图的有无环分析进行理解，和有向图不同的是，无向图中要考虑到两个节点之间的回路不是“环”，即入度为0或入度为1的节点任务都可以放在任务队列中等待。具体算法如下： | | | | | | |
| 四、实验结果及分析和（或）源程序调试过程  ——————————结果——————————  测试样例一：  6 9  1 2 10  1 4 20  1 6 2  2 3 3  2 4 5  3 5 15  4 5 11  4 6 10  5 6 3    自定义样例二：  4 6  1 2 2  2 3 2  2 4 1  1 3 5  3 4 3  1 4 4    自定义样例二：  4 3  1 2 4  2 3 5  2 4 7    ——————————调试过程——————————  1、最开始没有认真读题，写的程序中，都是节点0开始的，之后改程序的时候，忘记修改图的初始化函数里面的数组new的大小，导致程序“不能执行”。清楚题意，认真修改很重要。  2、优先降序排序问题，需要逻辑思维在线，要不然会卡很久  3、写程序不是很上手，整个实验，花了将近四个小时，其中debug就占了一半多的时间，平时写程序的时候就应该边写边编译测试运行一下，等到最后运行的时候，就不会那么悲惨！  4、深入理解无向图的有无环判断，更有向图的判断很相近，也可以用拓扑排序问题解决，但是在判断条件上要多加思考。  ——————————————程序———————————————  #include<iostream>  #include<queue>  #include<stack>  using namespace std;  const int INF = 0x7fffffff;  enum Visit{ UNVISITED, VISITED };  class Graph{  private:      void operator = (const Graph&) {}      Graph(const Graph&) {}  public:      Graph(){}      virtual ~Graph() {}      virtual void Init(int n) = 0;      virtual int n() = 0;      virtual int e() = 0;      virtual int first(int v) = 0; // the first neighbor of v      virtual int next(int v, int w) = 0; // the next neighbor of v      virtual void setEdge(int v1, int v2, int wght) = 0;      virtual void delEdge(int v1, int v2) = 0;      virtual bool isEdge(int i, int j) = 0;      virtual int weight(int v1, int v2) = 0;      virtual int getMark(int v) = 0;      virtual void setMark(int v, int val) = 0;  };  class MGraph{  private:      int numV, numE;      int \*\*matrix;      int \*mark;  public:      MGraph(int numV){ Init(numV); }      ~MGraph(){          delete[] mark;          for (int i = 0; i <= numV;i++) delete[] matrix[i];          delete[] matrix;      }      void Init(int n){          numV = n;          numE = 0;          mark = new int[numV+1];          matrix = (int \*\*)new int \*[numV+1];          for (int i = 0; i <= numV;i++){              mark[i] = UNVISITED;              matrix[i] = new int[numV+1];              for (int j = 0; j <= numV;j++)                  matrix[i][j]=INF;          }      }      int n() { return numV; }      int e() { return numE; }      int first(int v){          for (int i = 1; i <= numV;i++)              if(matrix[v][i]!=INF) return i;          return numV+1;      }      int next(int v,int w){          for (int i = w+1; i <= numV;i++)              if(matrix[v][i]!=INF) return i;          return numV+1;      }      void setEdge(int u,int v,int w){          // Assert(w > 0, "负数权重出现");          if(matrix[u][v]==INF) numE++;          matrix[u][v] = w;      }      void delEdge(int u,int v){          if(matrix[u][v]!=INF) numE--;          matrix[u][v] = INF;      }      bool isEdge(int u,int v){ return matrix[u][v] != INF; }      int weight(int u, int v) { return matrix[u][v]; }      int getMark(int u) { return mark[u]; }      void setMark(int u, int val) { mark[u] = val; }      void printBFS(int s){          // 每一步操作都需要重置mark          fill(mark, mark + numV + 1, UNVISITED);          queue<int> q;          mark[s] = VISITED;          cout << s;          q.push(s);          while(!q.empty()){              int t = q.front();              q.pop();              for (int i = 1; i <= numV;i++){                  if(matrix[t][i] != INF && mark[i]==UNVISITED){                      cout << " " << i;                      q.push(i);                      mark[i] = VISITED;                  }              }          }          cout << endl;      }      void printDFS(int s){          static bool flag = true; // 是否是第一个元素          if(flag){              fill(mark, mark + numV + 1, UNVISITED); // mark重置              cout << s;          }          else cout << " " << s;          mark[s]=VISITED;          for (int i = first(s); i <= numV;i = next(s,i)){              flag = false;              if(mark[i]==UNVISITED && i<=numV)                  printDFS(i);          }      }      void printDijkstra(int s,int\*d,int \*parent){          fill(mark, mark + numV + 1, UNVISITED);          fill(parent, parent + numV + 1, -1);          for (int i = 1; i <= numV;i++){              d[i] = matrix[s][i];              if(d[i]!=INF) parent[i]=s;          }          d[s] = 0;          mark[s] = VISITED;          while(true){              int v=-1, min = INF;              for (int i = 1; i <= numV;i++){                  if(d[i]<min && mark[i]==UNVISITED){                      v = i;                      min = d[i];                  }              }              if(v==-1) break;              mark[v]=VISITED;              for (int i = first(v); i <= numV;i=next(v,i)){                  if(d[i]>d[v]+matrix[v][i]){                      d[i] = d[v] + matrix[v][i];                      parent[i] = v; // 记路径上i的parent为v                  }              }          }          // 路径打印          for (int i = 1; i <= numV;i++){              stack<int> spath;              if(i == s) continue;              if(d[i]==INF || i == s) {                  cout<<"0"<<endl;                  continue;              }              int temp = i;              while(1){                  if(temp==-1) break;                  spath.push(temp);                  temp = parent[temp];              }              while(!spath.empty()){                  cout << spath.top() << " ";                  spath.pop();              }              cout << d[i] << endl;          }      }      void printPrim(){          int parent[numV+1];          int d[numV + 1];          int pu[numV + 1];          int pv[numV + 1];          int pw[numV + 1];          int count = 0;          for (int i = 0; i <= numV;i++){              mark[i] = UNVISITED;          }          int s = -1, min = INF;          // 找到最小的节点（最小生成树）          for (int i = 1; i <= numV;i++){              for (int j = 1; j <= numV;j++){                  if(matrix[i][j]<min) {                      s = i;                      min = matrix[i][j];                  }              }          }          // 数据初始化          for (int i = 1; i <= numV;i++){              d[i] = matrix[s][i];              parent[i] = s;          }          d[s] = 0;          mark[s] = VISITED;          while(true){              int v=-1, min = INF;              for (int i = 1; i <= numV;i++){                  if(d[i]<min && mark[i]==UNVISITED){                      v = i;                      min = d[i];                  }              }              if(v==-1) break;              pu[count] = parent[v]<v?parent[v]:v;              pv[count] = parent[v]>v?parent[v]:v;              pw[count++] = matrix[parent[v]][v];              mark[v]=VISITED;              for (int i = first(v); i <= numV;i=next(v,i)){                  if(d[i] > matrix[v][i]){                      d[i] = matrix[v][i];                      parent[i] = v; // 记路径上i的parent为v                  }              }          }          // 排序打印输出          for (int i = 0;i<count;i++){              for (int j = i + 1;j<count;j++){                  if(pw[i]>pw[j] || (pw[i]==pw[j]&&pu[i]>pu[j])) {                      swap(pu[i], pu[j]);                      swap(pv[i], pv[j]);                      swap(pw[i], pw[j]);                      // int temp;                      // temp = pu[i];pu[i] = pu[j];pu[j] = temp;                      // temp = pv[i];pv[i]=pv[j];pv[j]=temp;                      // temp = pw[i];pw[i]=pw[j];pw[j]=temp;                  }              }          }          for (int i = 0;i<count;i++){              cout << pu[i] << " " << pv[i] << " " << pw[i] << endl;          }      }        bool tophelp(){          // 无向图判断是否有环          queue<int> task;          int count[numV + 1];          for (int i = 1; i <= numV;i++) count[i] = 0;          for (int i = 1; i <= numV;i++){              for (int j = 1; j <= numV;j++){                  if(matrix[i][j]!=INF) count[j]++;              }          }          for (int i = 1; i <= numV;i++){              if(count[i]==0 || count[i]==1) task.push(i);          }          while(!task.empty()){              int t = task.front();              task.pop();              for (int i = 1; i <= numV;i++){                  if(matrix[t][i]!=INF) count[i]--;                  if(count[i]==0 || count[i]==1) task.push(i);              }          }          for (int i = 1; i <= numV;i++){              if(count[i]>1) return true; // 有环          }          return false; // 无环      }  };  int main(){      int n, m;      cin >> n >> m;      MGraph mGraph(n);      for (int i = 0; i < m;i++){          int u, v, w;          cin >> u >> v >> w;          mGraph.setEdge(u, v, w);          mGraph.setEdge(v, u, w);      }      mGraph.printBFS(1);      mGraph.printDFS(1);      cout << "\n";      int d[n+1];      int parent[n+1];      int s = 1;      mGraph.printDijkstra(s, d, parent);      mGraph.printPrim();      if(mGraph.tophelp()) cout<<"YES";      else cout << "NO";      return 0;  } | | | | | | |