山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201900161140 | 姓名： 张文浩 | | 班级： 19级人工智能 |
| 实验题目：图 | | | |
| 实验学时：2 | | 实验日期： 10.8 | |
| 实验目的：  1、掌握图的基本概念，图的描述方法；图上的操作方法实现。  2、掌握图结构的应用。 | | | |
| 软件开发工具：  Vscode | | | |
| 1. 实验内容   **题目描述**：  创建无向图类，存储结构使用邻接链表，提供操作：插入一条边，删除一条边，BFS，DFS。  **输入输出格式**：  **输入**：  第一行四个整数n，m，s，t。n (10≤n≤100000) 代表图中点的个数，m (10≤m≤200000) 代表接下来共有m个操作，s代表起始点，t代表终点。  接下来m行，每行代表一次插入或删除边的操作，操作格式为：  0 u v 在点u和v之间增加一条边；  1 u v 删除点u和v之间的边。  **输出**：  第一行输出图中有多少个连通分量；  第二行输出所有连通子图中最小点的编号（升序），编号间用空格分隔；  第三行输出从s点开始的dfs序列长度；  第四行输出从s点开始的字典序最小的dfs序列；  第五行输出从t点开始的bfs序列的长度；  第六行输出从t点开始字典序最小的bfs序列；  第七行输出从s点到t点的最短路径，若是不存在路径则输出-1。   1. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法）   根据题目要求要用邻接表实现图，所以要构建几个结构体，分别是  **EdgeNode**边结点的结构体：  其中的成员包括这个边所指向的顶点的位置adjvex，指向下一条边的指针next，和这条边的权重（此题默认是1）  **HeadNode**表头节点的结构体： 这个结构体用于创建一个列表，假设创建的链表为HeadNode list,则list[i]所指向的就是从i这个顶点出发的第一条边节点结构体的指针。所以这个HeadNode结构体中包括的成员是指向EdgeNode的一个指针firstNode，和i这个顶点的节点信息val。  **ALGraph**邻接表结构体：  成员包括了HeadNode的一个数组list，意义如上面所述。还包括了整个表（图）的总的顶点个数和边数  最后就创建一个Graph的类：  其中的成员变量就是一个邻接表的结构体ALGraph G，然后根据题目要求写相关的函数。，包括  Graph(int n = 100) //构造函数  void insert(int u, int v); //增加边      void erase(int u, int v);  //删除边      int component\_count();     //返回图中有多少连通分量      void component\_print();    //输出连通子图最小节点编号      void dfs(int start);       //计算从s节点开始的dfs序列长度      void dfs\_count(int start); //为求连通分量使用的的dfs      void dfs\_min(int start);   //输出从s点开始的字典序最小的dfs序列      int bfs(int start);        //返回从t点开始的bfs序列的长度      void bfs\_min(int start);   //输出从t点开始字典序最小的bfs序列      void dijkstra(int s);      //输出从s点到t点的最短路径，若是不存在路径则输出-1      void show();               //输出整个图的邻接表，方便调试  **Graph(int n = 100) //构造函数**  构造函数就是初始化一下邻接表中HeadNode的相关信息。  **void insert(int u, int v); //增加边**  增加边的函数，因为本题里的都是双向边，所以加入一条边，在邻接表中要插入两个边。所以先新建两个边节点，其终点分别指向u和v，然后分别把这两个新建的边节点插入到邻接表中即可。  因为题目要求后面实现字典序最小的输出操作，所以在构建邻接表的时候保证链表是升序存储的。所以在插入的时候要保持链表的有序性。方法就是先根据起点u找到对应的链表的firstnode头指针，然后利用两个指针p和pp遍历链表，pp在p后面跟着，当p遍历到的adjvex比v大的时候就可以把刚才新建立的边节点插入到pp和p之间了，当然还要考虑一下边界情况，即要插入到头节点或插入到尾部的情况。然后再用相同的方法插入另一个边节点。  **void erase(int u, int v);  //删除边**  删除操作思路与插入操作差不多，也是要删除邻接表中的两个边节点。根据起点u找到对应的链表的头指针，利用两个指针p和pp遍历当前链表，pp跟在p后面，直到p对应的adjvex等于v，就让p的前一个节点pp指向p的下一个节点即可，最后再把p的空间释放掉。再用相同的方法删除另一个边节点。  **int component\_count();     //返回图中有多少连通分量**  计算图中的连通分量需要用到vis数组，来描述某个顶点是否被遍历过，防止被重复遍历，vis[i]=1的意义是i这个顶点一定被遍历过了，vis[i]=0的意义是i这个顶点还没有被遍历过。  然后就可以开始从第一个顶点开始遍历，从遍历到的每一个顶点都以这个顶点开始做一次dfs，把dfs能遍历到的顶点的vis都置为1，说明这些顶点与遍历到的这个顶点是连通的，当遍历到一个vis=0的顶点的时候，说明这个点之前没有顶点与它连通，他要自己通过dfs寻找他所在的连通分量的其他顶点，此时连通分量个数+1，最终把所有的顶点都遍历过之后，就能得到连通分量的总个数了。  **void component\_print();    //输出连通子图最小节点编号**  在上一个函数中，我们遍历顶点的顺序是从小到大进行的，所以我们开始找到每个连通分量的起点一定是这个连通分量的最小顶点，直到输出每个新连通分量时输出这个顶点即可。  **void dfs(int start);       //计算从s节点开始的dfs序列长度**  也是利用递归实现dfs，每进入一次dfs函数，序列长度结果+1即可。  **void dfs\_min(int start);   //输出从s点开始的字典序最小的dfs序列**  也是利用递归实现dfs，因为我们在构建邻接表的时候，里面的每一个链表都是升序排列的，所以直接dfs遍历的顺序就是字典序最小序列，每次进入dfs直接输出当前节点的元素值val即可。  **int bfs(int start);        //返回从t点开始的bfs序列的长度**  利用队列实现bfs，每次从队列中取出一个顶点，然后把与它连接的且没有被遍历过的(vis=0)的顶点送入队列，每次将序列长度结果+1即可，在队列空了之后就能得到结果了。  **void bfs\_min(int start);   //输出从t点开始字典序最小的bfs序列**  利用队列实现bfs，每次从队列中取出一个顶点，然后把与它连接的且没有被遍历过的(vis=0)的顶点送入队列，每次输出取出顶点的元素值val即可。  **void dijkstra(int s);      //输出从s点到t点的最短路径，若是不存在路径则输出-1**  利用dijkstra算法求最短路，每一条边的权重都是1即可。利用优先队列实现dijkstra，用dis数组表示整个顶点到s的距离，一开始将dis初始化为一个很大的数，每次再队列中取出dis距离最小的顶点x，然后遍历每个与这个顶点连接的顶点y，如果dis[y]>dis[x]+1，就进行松弛操作，把松弛后的顶点送入优先队列，每个顶点一定只会被松弛一次。在执行完dijkstra之后dis数组就建立好了，我们就得到了从s点为起点到其他所有顶点的最短路径。最后输出dis[t]即可。   1. 测试结果（测试输入，测试输出）   样例输入  10 20 4 5 0 6 4 0 10 3 0 4 8 0 4 10 1 4 10 0 2 1 0 5 8 0 5 2 0 10 7 0 9 6 0 9 1 0 7 1 0 8 10 0 7 5 0 8 3 0 6 7 1 6 4 1 8 3 0 7 8 0 9 2  输出  ————————  1->2->7->9  2->1->5->9  3->10  4->8  5->2->7->8  6->7->9  7->1->5->6->8->10  8->4->5->7->10  9->1->2->6  10->3->7->8  ————————  1  1  10  4 8 5 2 1 7 6 9 10 3  10  5 2 7 8 1 9 6 10 4 3  2  上面先展示了最终的邻接表的情况。  下面7行是题目要求的结果，分别是  第一行输出图中有多少个连通分量； 1  第二行输出所有连通子图中最小点的编号（升序），编号间用空格分隔； 1  第三行输出从s点开始的dfs序列长度； 10  第四行输出从s点开始的字典序最小的dfs序列；  4 8 5 2 1 7 6 9 10 3  第五行输出从t点开始的bfs序列的长度； 10  第六行输出从t点开始字典序最小的bfs序列；  5 2 7 8 1 9 6 10 4 3  第七行输出从s点到t点的最短路径，若是不存在路径则输出-1。 2   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   本次实验利用邻接表实现了图的结构，最后在求最短路的时候用到了**dijkstra算法**。  算法特点：  迪科斯彻算法使用了广度优先搜索解决赋权有向图或者无向图的单源最短路径问题，算法最终得到一个最短路径树。该算法常用于路由算法或者作为其他图算法的一个子模块。  算法的思路  Dijkstra算法采用的是一种贪心的策略，声明一个数组dis来保存源点到各个顶点的最短距离和一个保存已经找到了最短路径的顶点的集合：T，初始时，原点 s 的路径权重被赋为 0 （dis[s] = 0）。若对于顶点 s 存在能直接到达的边（s,m），则把dis[m]设为w（s, m）,同时把所有其他（s不能直接到达的）顶点的路径长度设为无穷大。初始时，集合T只有顶点s。  然后，从dis数组选择最小值，则该值就是源点s到该值对应的顶点的最短路径，并且把该点加入到T中，OK，此时完成一个顶点，  然后，我们需要看看新加入的顶点是否可以到达其他顶点并且看看通过该顶点到达其他点的路径长度是否比源点直接到达短，如果是，那么就替换这些顶点在dis中的值。  然后，又从dis中找出最小值，重复上述动作，直到T中包含了图的所有顶点。  在图的功能实现中，要注意所有节点在邻接表中存储的顺序是由小到大，否则不便于最小编号，以及最小字典序序列的输出。在进行完dfs以及bfs查询之后，一定要记得重置vis数组。因为没有重置该数组，我在进行测试时出现了许多次错误，最后提供调试才发现出问题的地方。另外，无向图可以看作是路径长度为原图两倍的有向图。在进行插入、删除操作的时候要对两个方向的边都进行同样的操作。  实现查找最短路径的功能可以利用许多种方法，例如迪杰斯特拉算法、spfa等，其本质思想都是通过bfs时更新维护dis数组（其中存储起始点到所有点的最短距离），在完成遍历之后可以得到最短路径的长度。  在题目涉及到大量指针操作的时候，要记得讨论指针是否为空的情况，否则很容易出现报错，导致程序无法正常运行，尤其是指针在链表首位、末位时的插入、删除操作等。   1. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）   #include <iostream>  #include <queue>  #include <cstring>  #define maxsize 100001 //最大顶点数  using namespace std;  int vis[maxsize], dis[maxsize];  int len; //序列长度  struct EdgeNode  {                                 //定义边节点      int adjvex;                   //该边所指向的顶点的位置      struct EdgeNode \*next; //指向下一条边的指针      int w;                        //边的权重  };  struct HeadNode //定义表头节点  {      EdgeNode \*firstNode; //指向第一条依附于表头的指针      int val;             //节点信息  };  struct ALGraph //邻接表  {      HeadNode list[maxsize];  //创建有maxsize个节点的图      int point\_num, edge\_num; //整个图的顶点数和边数  };  class Graph  {  public:      Graph(int n = 100) //构造函数      {          G.point\_num = n;          for (int i = 0; i <= n; i++)          {              G.list[i].val = i;              G.list[i].firstNode = NULL;          }      }      void insert(int u, int v); //增加边      void erase(int u, int v);  //删除边      int component\_count();     //返回图中有多少连通分量      void component\_print();    //输出连通子图最小节点编号      void dfs(int start);       //计算从s节点开始的dfs序列长度      void dfs\_count(int start); //为求连通分量使用的的dfs      void dfs\_min(int start);   //输出从s点开始的字典序最小的dfs序列      int bfs(int start);        //返回从t点开始的bfs序列的长度      void bfs\_min(int start);   //输出从t点开始字典序最小的bfs序列      void dijkstra(int s);      //输出从s点到t点的最短路径，若是不存在路径则输出-1      void show();               //输出整个图的邻接表，方便调试  private:      ALGraph G;  };  void Graph::insert(int u, int v)  {      //新建两个边节点      EdgeNode \*edge1 = new EdgeNode;      EdgeNode \*edge2 = new EdgeNode;      edge1->adjvex = v; //edge1边是从u指向v的      edge2->adjvex = u; //edge2边是从v指向u的      //因为为了方便后面的按字典序最小输出bfs和dfs序的题目要求      //要保证链表的有序性，即从小到大按顺序排列，所以插入的时候要放到合适的位置      EdgeNode \*p = G.list[u].firstNode, \*pp = NULL;      if (p == NULL || p->adjvex > v) //如果当前链表为空或者第一个节点的adjvex就比要插入的节点的adjvex大，就直接插入到链表首部      {          edge1->next = G.list[u].firstNode;          G.list[u].firstNode = edge1;      }      else      {          while (p != NULL && p->adjvex < v)          {              //cout<<"\*\*\*\*"<<p->adjvex<<endl;              pp = p;              p = p->next;          }          if (p == NULL) //当前链表中所有的边节点的adjvex都比要插入边的adjvex小，就就插入到链表的最后一个位置          {              pp->next = edge1;              edge1->next = NULL;          }          else          {              //这时p的adjvex比要插入的adjvex大，pp的adjvex比要插入的adjvex小，插入到p个pp之间即可              edge1->next = p;              pp->next = edge1;          }      }      //此时edge1插入完毕，下面用同样的方法插入edge2即可      p = G.list[v].firstNode;      pp = NULL;      if (p == NULL || p->adjvex > u) //如果当前链表为空或者第一个节点的adjvex就比要插入的节点的adjvex大，就直接插入到链表首部      {          edge2->next = G.list[v].firstNode;          G.list[v].firstNode = edge2;      }      else      {          while (p != NULL && p->adjvex < u)          {              pp = p;              p = p->next;          }          if (p == NULL) //当前链表中所有的边节点的adjvex都比要插入边的adjvex小，就就插入到链表的最后一个位置          {              pp->next = edge2;              edge2->next = NULL;          }          else          {              //这时p的adjvex比要插入的adjvex大，pp的adjvex比要插入的adjvex小，插入到p个pp之间即可              edge2->next = p;              pp->next = edge2;          }      }      G.edge\_num++;  }  void Graph::erase(int u, int v)  {      EdgeNode \*p = G.list[u].firstNode, \*pp = NULL;      while (p != NULL && p->adjvex != v)      { //遍历当前链表，找到adjvex=v的边          pp = p;          p = p->next;      }      if (p == NULL) //如果没找到      {          cout << "不存在，删除失败" << endl;          return;      }      //找到了      if (pp != NULL) //如果要删除的边不是链表的firstnode      {          pp->next = p->next;      }      else //如果要删除的边正好是这个链表的firstnode      {          G.list[u].firstNode = p->next;      }      delete p; //释放被删除边所占的空间      //用同样的方法删除对边      p = G.list[v].firstNode;      pp = NULL;      while (p != NULL && p->adjvex != u)      { //遍历当前链表，找到adjvex=v的边          pp = p;          p = p->next;      }      if (p == NULL) //如果没找到      {          cout << "不存在，删除失败" << endl;          return;      }      //找到了      if (pp != NULL) //如果要删除的边不是链表的firstnode      {          pp->next = p->next;      }      else //如果要删除的边正好是这个链表的firstnode      {          G.list[v].firstNode = p->next;      }      delete p; //释放被删除边所占的空间      G.edge\_num--;  }  void Graph::show()  {      cout << "-----------------" << endl;      for (int i = 1; i <= G.point\_num; i++)      {          cout << G.list[i].val;          EdgeNode \*p = G.list[i].firstNode;          while (p != NULL)          {              cout << "->" << G.list[p->adjvex].val;              p = p->next;          }          cout << endl;      }  }  void Graph::dfs\_count(int start)  {      vis[start] = 1;      EdgeNode \*p = G.list[start].firstNode;      while (p != NULL)      {          if (!vis[p->adjvex])              dfs\_count(p->adjvex);          p = p->next;      }  }  int Graph::component\_count() //返回图中有多少连通分量  {      int res = 0;      memset(vis, 0, sizeof(vis));      for (int i = 1; i <= G.point\_num; i++)      {          if (!vis[i]) //如果还没有被遍历过，就以这个点为起点再遍历，说明有一个新的连通分量          {              dfs\_count(i);              res++;          }      }      return res;  }  void Graph::component\_print() //输出连通子图最小节点编号  {      memset(vis, 0, sizeof(vis));      for (int i = 1; i <= G.point\_num; i++)      {          if (!vis[i]) //如果还没有被遍历过，就以这个点为起点再遍历，说明有一个新的连通分量          {              cout << i << " ";              dfs\_count(i);          }      }      cout << endl;  }  void Graph::dfs(int start) //返回从s节点开始的dfs序列长度  {      vis[start] = 1; //说明这个点已经被遍历过了，防止后面被重复遍历      EdgeNode \*p = G.list[start].firstNode;      len++;      while (p != NULL)      {          if (!vis[p->adjvex])              dfs(p->adjvex);          p = p->next;      }  }  void Graph::dfs\_min(int start) //输出从s点开始的字典序最小的dfs序列  {      cout << G.list[start].val << " ";      vis[start] = 1; //说明这个点已经被遍历过了      EdgeNode \*p = G.list[start].firstNode;      while (p != NULL)      {          if (!vis[p->adjvex])          {              dfs\_min(p->adjvex);          }          p = p->next;      }  }  int Graph::bfs(int start) //返回从t点开始的bfs序列的长度  {      memset(vis, 0, sizeof(vis));      int res = 0;      queue<int> q; //利用队列实现bfs      q.push(start);      int f;      while (!q.empty())      {          f = q.front();          q.pop();          res++;          vis[f] = 1;          EdgeNode \*p = G.list[f].firstNode;          while (p != NULL)          {              if (!vis[p->adjvex])              {                  vis[p->adjvex] = 1;                  q.push(p->adjvex);              }              p = p->next;          }      }      return res;  }  void Graph::bfs\_min(int start) //输出从t点开始字典序最小的bfs序列  {      memset(vis, 0, sizeof(vis));      queue<int> q; //利用队列实现bfs      q.push(start);      int f;      while (!q.empty())      {          f = q.front();          q.pop();          cout << G.list[f].val << " ";          vis[f] = 1;          EdgeNode \*p = G.list[f].firstNode;          while (p != NULL)          {              if (!vis[p->adjvex])              {                  vis[p->adjvex] = 1;                  q.push(p->adjvex);              }              p = p->next;          }      }      cout << endl;  }  void Graph::dijkstra(int start) //输出从s点到t点的最短路径，若是不存在路径则输出-1  {      priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> q;      for (int i = 1; i <= G.point\_num; i++)      {          dis[i] = maxsize;          vis[i] = 0;      }      dis[start] = 0;      q.push(make\_pair(0, start));      while (!q.empty())      {          int x = q.top().second;          q.pop();          if (vis[x])              continue;          vis[x] = 1;          EdgeNode \*p = G.list[x].firstNode;          while (p != NULL)          {              int nex = p->adjvex;              if (dis[nex] > dis[x] + 1)              {                  dis[nex] = dis[x] + 1;                  q.push(make\_pair(dis[nex], nex));              }              p = p->next;          }      }  }  int main()  {      int n, m, s, t, op, u, v;      cin >> n >> m >> s >> t;      Graph g(n);      while (m--)      {          cin >> op >> u >> v;          if (op == 0)          {              g.insert(u, v);          }          else if (op == 1)          {              g.erase(u, v);          }          g.show();      }      //第一行输出图中有多少个连通分量      cout << g.component\_count() << endl;      //第二行输出所有连通子图中最小点的编号（升序），编号间用空格分隔；      g.component\_print();      //第三行输出从s点开始的dfs序列长度；      memset(vis,0,sizeof(vis));      g.dfs(s);      cout << len << endl;      //第四行输出从s点开始的字典序最小的dfs序列；      memset(vis, 0, sizeof(vis));      g.dfs\_min(s);      cout << endl;      //第五行输出从t点开始的bfs序列的长度；      cout << g.bfs(t) << endl;      //第六行输出从t点开始字典序最小的bfs序列；      g.bfs\_min(t);      //第七行输出从s点到t点的最短路径，若是不存在路径则输出-1。      g.dijkstra(s);      if (dis[t] == maxsize)          cout << -1 << endl;      else          cout << dis[t] << endl;      // system("pause");      return 0;  } | | | |
|  | | | |