**附件：**

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：202000130138 | 姓名：宋璎航 | | 班级： 20.3 |
| 实验题目：图 | | | |
| 实验学时：4 | | 实验日期：2021/12/23 | |
| 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。 | | | |
| 软件开发环境：  VSC | | | |
| 1. 实验内容   **1、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）   连通分量的计算：对每个点进行遍历dfs，并记录之前是否访问过该点，进行几次dfs就是有几个LTFS  最小点编号：在上述遍历过程中记录dfs所经过的最小点即可  Dfs与bfs序列长度与输出序列：即在进行dfs时输出即可  最短路径：设计bfs用dijikstra算法即可   1. 测试结果（测试输入，测试输出）  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  1  10  4 8 5 2 1 7 6 9 10 3  10  5 2 7 8 1 9 6 10 4 3  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）   结果均正确   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 "bits/stdc++.h"  using namespace std;  struct cnode  {  int element;  cnode \*next;  cnode(int theElement)  {  element = theElement;  next = NULL;  }  cnode(int theElement, cnode \*theNext)  {  element = theElement;  next = theNext;  }  };  class glist  {  // cnode \*firstNode;  // cnode \*lastNode;  cnode \*\*aList;  int gSize;  int \*r, \*a;  public:  int dfsLen, bfsLen;  glist(int n);  void flush();  void insert(int, int);  void remove(int, int);  void outputMin();  void bfs(int);  void \_bfs(int);  void dfs(int);  void \_dfs(int);  int countOfLTFT();  void shortPath(int s, int t);  };  void glist::flush()  {  dfsLen = 0;  bfsLen = 0;  for (int i = 1; i <= gSize; ++i)  r[i] = 0;  }  glist::glist(int n)  {  gSize = n;  aList = new cnode \*[n + 1];  r = new int[n + 1];  a = new int[n + 1];  for (int i = 1; i <= gSize; ++i)  {  aList[i] = NULL;  r[i] = 0;  a[i] = 0;  }  }  void glist::insert(int x, int y)  {  cnode \*p1 = aList[x];  cnode \*p2 = aList[y];  cnode \*pp = NULL, \*p = NULL;  if (p1 == NULL)  aList[x] = new cnode(y);  else if (p1->element > y)  {  p = new cnode(y, p1);  aList[x] = p;  }  else  {  while (p1 != NULL and y > p1->element)  {  pp = p1;  p1 = p1->next;  }  if (p1 == NULL)  pp->next = new cnode(y, p1);  else  {  if (p1->element == y)  return;  else  pp->next = new cnode(y, p1);  }  }  pp = NULL;  p = NULL;  if (p2 == NULL)  aList[y] = new cnode(x);  else if (p2->element > x)  {  p = new cnode(x, p2);  aList[y] = p;  }  else  {  while (p2 != NULL and x > p2->element)  {  pp = p2;  p2 = p2->next;  }  if (p2 == NULL)  pp->next = new cnode(x, p2);  else  {  if (p2->element == x)  return;  else  pp->next = new cnode(x, p2);  }  }  return;  }  void glist::remove(int x, int y)  {  cnode \*p1 = aList[x];  cnode \*p2 = aList[y];  cnode \*pp1 = NULL, \*pp2 = NULL;  while (p1 != NULL and p1->element != y)  {  pp1 = p1;  p1 = p1->next;  }  if (pp1 == NULL)  {  if (p1 == NULL)  return; //error：not exist such an edge  else  {  aList[x] = p1->next;  delete p1;  }  }  else  {  if (p1 == NULL)  return; //error：not exist such an edge  else  {  pp1->next = p1->next;  delete p1;  }  }  while (p2 != NULL and p2->element != x)  {  pp2 = p2;  p2 = p2->next;  }  if (pp2 == NULL)  {  if (p2 == NULL)  return; //error：not exist such an edge  else  {  aList[y] = p2->next;  delete p2;  }  }  else  {  if (p2 == NULL)  return; //error：not exist such an edge  else  {  pp2->next = p2->next;  delete p2;  }  }  }  void glist::outputMin()  {  int k = -1;  for (int i = 1; i <= gSize; ++i)  {  if (r[i] == 0)  {  dfs(i);  a[++k] = i;  cout << i << ' ';  }  }  cout << '\n';  }  void glist::dfs(int y)  {  dfsLen++;  r[y] = 1;  cnode \*p = aList[y];  while (p != NULL)  {  if (r[p->element] == 0)  dfs(p->element);  p = p->next;  }  }  void glist::\_dfs(int y)  {  cout << y << ' ';  dfsLen++;  r[y] = 1;  cnode \*p = aList[y];  while (p != NULL)  {  if (r[p->element] == 0)  \_dfs(p->element);  p = p->next;  }  }  void glist::bfs(int y)  {  queue<int> q;  q.push(y);  r[y] = 1;  while (!q.empty())  {  int F = q.front();  bfsLen++;  q.pop();  for (cnode \*u = aList[F]; u != NULL; u = u->next)  {  if (r[u->element] == 0)  {  q.push(u->element);  r[u->element] = 1;  }  }  }  }  void glist::\_bfs(int y)  {  queue<int> q;  q.push(y);  r[y] = 1;  while (!q.empty())  {  int F = q.front();  cout << F << ' ';  bfsLen++;  q.pop();  for (cnode \*u = aList[F]; u != NULL; u = u->next)  {  if (r[u->element] == 0)  {  q.push(u->element);  r[u->element] = 1;  }  }  }  }  int glist::countOfLTFT()  {  int ans = 0;  for (int i = 1; i <= gSize; ++i)  {  if (r[i] == 0)  {  dfs(i);  ans++;  }  }  return ans;  }  void glist::shortPath(int s, int t)  {  queue<int> q;  q.push(s);  r[q.front()] = 1;  int path[gSize + 1];  for (int i = 1; i <= gSize; ++i)  path[i] = 0;  while (!q.empty())  {  int w = q.front();  q.pop();  cnode \*p = aList[w];  while (p != NULL)  {  if (r[p->element] == 0)  {  if (p->element == t)  {  cout << path[w] + 1 << endl;  return;  }  path[p->element] = path[w] + 1;  q.push(p->element);  r[p->element] = 1;  }  p = p->next;  }  }  cout << -1 << endl;  }  int main()  {  int n, m, s, t;  cin >> n >> m >> s >> t;  glist G(n);  while (m--)  {  bool opt;  cin >> opt;  if (!opt)  {  int t1, t2;  cin >> t1 >> t2;  G.insert(t1, t2);  }  else  {  int t1, t2;  cin >> t1 >> t2;  G.remove(t1, t2);  }  }  G.flush();  cout << G.countOfLTFT() << endl;  G.flush();  G.outputMin();  G.flush();  G.dfs(s);  cout << G.dfsLen << endl;  G.flush();  G.\_dfs(s);  cout << '\n';  G.flush();  G.bfs(t);  cout << G.bfsLen << endl;  G.flush();  G.\_bfs(t);  cout << '\n';  G.flush();  G.shortPath(s, t);  return 0;  } | | | |