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：201705130113 | 姓名：黄瑞哲 | | 班级：计科17.3 |
| 实验题目：图 | | | |
| 实验学时：4 | | 实验日期： 2018.12.20 | |
| 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 | | | |
| 软件环境：  Visual Studio Community 2017 | | | |
| 1. 实验内容（题目内容，输入要求，输出要求） 2. 创建无向图类，存储结构分别使用邻接矩阵和邻接链表。提供操作：插入一条边、删除一条边、遍历、BFS、DFS等。 3. 键盘输入图中顶点的个数n和边的数目e，以定点对（i，j）形式依次输入图的每一条边或随机生成含e条边的图，其中（i，j）表示顶点i和顶点j之前有边相连，建立图。 4. 判断图是否连通。若不连通，输出该图的连通分量的个数及每个连通分量中的顶点； 5. 对建立好的连通图，键盘输入一顶点，输出从该顶点开始的一个DFS序列和BFS序列。 6. 键盘输入两顶点，输出两顶点之间的最短路径。 7. 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） 8. 邻接矩阵采用二维数组存储，a[i][j]表示顶点i到顶点j的边权。邻接链表采用数组链表存储，a[i]是一个链表，存储了与顶点i相邻接的点和边。 9. Bfs采用队列，每次遍历的时候将当前队首顶点u的所有未被访问的邻接点加入队列中。Dfs采用递归实现，对于当前点u，遍历未被访问的邻接点，每个点开始递归。 10. 判断图的连通性：如果从一个点开始bfs或dfs能够遍历所有的点则说明图是连通的。求连通块个数：从没有被访问过的点开始遍历，遍历的次数为连通块的个数。 11. 求最短路径采用dijkstra算法或floyd算法。 12. 测试结果（测试输入，测试输出，结果分析）      1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   对于无权图可以采用bfs求最短路。Bfs中第一次出现的点即为该点的最短路。   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）   /\*pch.h\*/  #ifndef PCH\_H  #define PCH\_H  // TODO: 添加要在此处预编译的标头  #include <graph/adjacencyGrap.h>  #include <graph/linkedGraph.h>  #endif //PCH\_H  /\*main.cpp\*/  #include "pch.h"  #include <iostream>  using namespace std;  int main() {  int n, m;  cin >> n >> m;  adjacencyGraph<int> g(n);  for (auto i = 0; i < m; ++i) {  int u, v, w;  cin >> u >> v >> w;  g.add(u, v, w);  g.add(v, u, w);  }  cout << g.connectedComponent(true);  cout << endl;  cout << g.connectedComponent(false);  cout << endl << g.dis(5, 2) << endl;  return 0;  }  /\*  5 6  1 3 3  2 3 1  4 1 5  4 5 2  5 1 2  5 2 1  \*/    /\*linkedGraph.h\*/  #ifndef linkedGraph\_  #define linkedGraph\_  #include <linearList/chain.h>  #include <queue/linkedQueue.h>  #include <tree/Heap.h>  #include <algorithm>  #include <iostream>  using namespace std;  template<typename T>  class linkedGraph {  typedef pair<int, T> p;  protected:  chain<p>\* e;  bool\* vis;  int n;  int m;  public:  explicit linkedGraph(const int n) :n(n) {  e = new chain<p>[n + 1];  vis = new bool[n + 1];  m = 0;  }  ~linkedGraph() {  delete[] e;  delete[] vis;  }  void add(int u, int v, const T& w) {  e[u].push\_back(make\_pair(v, w));  ++m;  }  void erase(int u, int v) {  auto i = 0;  for (auto it = e[u].begin(); it != e[u].end() && it->first != v; ++it, ++i);  e[u].erase(i);  --m;  }  void bfs(int s) {  linkedQueue<int> q;  vis[s] = true;  q.push(s);  while (!q.empty()) {  auto u = q.front(); q.pop();  cout << u << ' ';  for (auto it = e[u].begin(); it != e[u].end(); ++it) {  auto v = it->first;  if (!vis[v]) {  q.push(v);  vis[v] = true;  }  }  }  cout << endl;  }  void dfs(int u) {  cout << u << ' ';  vis[u] = true;  for (auto it = e[u].begin(); it != e[u].end(); ++it) {  auto v = it->first;  if (!vis[v])  dfs(v);  };  }  int connectedComponent(bool method = false) {  for (auto i = 1; i <= n; ++i) vis[i] = false;  auto tot = 0;  for (auto i = 1; i <= n; ++i) {  if (vis[i]) continue;  ++tot;  if (method) {  this->dfs(i);  cout << endl;  }  else this->bfs(i);  }  return tot;  }  int dis(int s, int t) const {  T\* dis = new T[n + 1];  for (auto i = 1; i <= n; ++i) dis[i] = 0x6f6f6f6f;  struct dh {  int u;  T d;  bool operator<(const dh& x) const { return d < x.d; }  };  Heap<dh, less<>> q;  dis[s] = 0;  q.push({ s, 0 });  while (!q.empty()) {  dh uu = q.top(); q.pop();  int u = uu.u;  if (dis[u] < uu.d) continue;  for (auto it = e[u].begin(); it != e[u].end(); ++it) {  int v = it->first;  T& w = it->second;  if (dis[v] > dis[u] + w) {  dis[v] = dis[u] + w;  q.push({ v, dis[v] });  }  }  }  auto ans = dis[t];  delete[] dis;  return ans;  }  };  #endif //linkedGraph\_  /\*adjacencyGrap.h\*/  #ifndef adjacencyGraph\_  #define adjacencyGraph\_  #include <cstring>  #include <queue/linkedQueue.h>  #include <iostream>  using namespace std;  template<typename T>  class adjacencyGraph {  public:  explicit adjacencyGraph(const int n, T MAX = 0x3fffffff) :n(n), MAX(MAX) {  e = new T\*[n + 1];  for (auto i = 0; i <= n; ++i) e[i] = new T[n + 1];  for (auto i = 0; i <= n; ++i)  for (auto j = 0; j <= n; ++j) {  if (i == j) e[i][j] = 0;  else e[i][j] = MAX;  }  g = new T\*[n + 1];  for (auto i = 0; i <= n; ++i) g[i] = new T[n + 1];  vis = new bool[n + 1];  for (auto i = 0; i <= n; ++i) vis[i] = false;  changed = true;  m = 0;  }  ~adjacencyGraph() {  for (auto i = 0; i <= n; ++i) delete[] e[i];  for (auto i = 0; i <= n; ++i) delete[] g[i];  delete[] e;  delete[] g;  delete[] vis;  };  void add(int u, int v, const T w) {  changed = true;  e[u][v] = w;  ++m;  }  void erase(int u, int v) {  changed = true;  e[u][v] = MAX;  --m;  }  void bfs(int s) {  linkedQueue<int> q;  vis[s] = true;  q.push(s);  while (!q.empty()) {  auto u = q.front(); q.pop();  cout << u << ' ';  for (auto i = 1; i <= n; ++i)  if (!vis[i] && e[u][i] < MAX) {  q.push(i);  vis[i] = true;  }  }  cout << endl;  }  void dfs(int u) {  cout << u << ' ';  vis[u] = true;  for (auto i = 1; i <= n; ++i)  if (!vis[i] && e[u][i] < MAX)  dfs(i);  }  int connectedComponent(bool method = false) {  for (auto i = 1; i <= n; ++i) vis[i] = false;  auto tot = 0;  for (auto i = 1; i <= n; ++i) {  if (vis[i]) continue;  ++tot;  if (method) {  this->dfs(i);  cout << endl;  }  else this->bfs(i);  }  return tot;  }  int dis(int u, int v) {  if (changed) floyd();  return g[u][v] >= MAX ? -1 : g[u][v];  }  protected:  T\*\* e;  T\*\* g;  bool\* vis;  bool changed;  int n;  int m;  T MAX;  void floyd() {  for (auto i = 1; i <= n; ++i)  for (auto j = 1; j <= n; ++j)  g[i][j] = e[i][j];  for (auto k = 1; k <= n; ++k)  for (auto i = 1; i <= n; ++i)  for (auto j = 1; j <= n; ++j)  g[i][j] = (g[i][j] < g[i][k] + g[k][j] ? g[i][j] : g[i][k] + g[k][j]);  changed = false;  }  };  #endif //adjacencyGrap\_ | | | |