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：201805130155 | 姓名：赵雨晗 | | 班级：计科18.3 |
| 实验题目：图 | | | |
| 实验学时：4 | | 实验日期： 2019.12.04 | |
| 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 Code | | | |
| 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）   #include<cstdio>  #include<iostream>  #include<cstring>  #include<string>  #include<algorithm>  #include<stdexcept>  using namespace std;  template<typename T>  struct chainNode  {  T element;  chainNode<T>\* \_next;  chainNode(const T& \_element, chainNode<T>\* \_next = NULL) : element(\_element), \_next(\_next) {}  chainNode(const chainNode<T>\*& c) : element(c->element), \_next(c->\_next) {}  };  template<typename T>  class chain{  public:  chain(int = 10);  chain(const chain<T>&);  ~chain();  bool empty() const;  int size() const;  int find(const T&) const;  void erase(int);  void insert(int, const T&);  void clear();  void push\_back(const T&);    chain<T>& operator=(const chain<T>&);  T& operator[](int);  const T& operator[](int) const;  class iterator;  class const\_iterator;  iterator begin() {return iterator(pHead -> \_next);}  iterator end() {return iterator(NULL);}  const\_iterator begin() const {return const\_iterator(pHead -> \_next);}  const iterator end() const {return const\_iterator(NULL);}  class iterator{  public:  typedef forward\_iterator\_tag iterator\_category;  typedef T value\_type;  typedef ptrdiff\_t difference\_type;  typedef T\* pointer;  typedef T& reference;  iterator(chainNode<T>\* theNode = NULL) :node(theNode) {}  T& operator\*() {return node -> element;}  T\* operator->() {return &node->element; }  iterator& operator++(){  node = node -> \_next;  return \*this;  }  iterator operator++(int){  iterator old = \*this;  node = node->\_next;  return old;  }    bool operator==(const iterator right) const {return node == right.node;}  bool operator!=(const iterator right) const {return node != right.node;}  protected:  chainNode<T>\* node;  };  class const\_iterator{  public:  typedef forward\_iterator\_tag iterator\_category;  typedef T value\_type;  typedef ptrdiff\_t difference\_type;  typedef T\* pointer;  typedef T& reference;  const\_iterator(chainNode<T>\* theNode) :node(theNode) {};  const T& operator\*() { return node->element; }  const T\* operator->() { return &node->element; }  const\_iterator& operator++()  {  node = node->\_next;  return \*this;  }  const\_iterator operator++(int)  {  const\_iterator old = \*this;  node = node->\_next;  return old;  }  bool operator==(const const\_iterator right) const { return node == right.node; }  bool operator!=(const const\_iterator right) const { return node != right.node; }  protected:  chainNode<T>\* node;  };  protected:  chainNode<T>\* pHead;  chainNode<T>\* pTail;  int listSize;  void checkIndex(int) const;  };  template<typename T>  chain<T>::chain(int initialCapacity){  if (initialCapacity < 1) throw out\_of\_range("the initial Capacity of arrayList must > 0");  listSize = 0;  pHead = new chainNode<T>(T());  pTail = pHead;  }  template<typename T>  chain<T>::chain(const chain<T>& c){  pHead = new chainNode<T>(c.pHead->element);  pTail = pHead;  chainNode<T>\* sourceNode = c.pHead ->\_next;  chainNode<T>\* currentNode = pHead;  while(sourceNode != NULL){  pTail = currentNode->\_next;  sourceNode = sourceNode->\_next;  }  listSize = c.listSize;  }  template<typename T>  chain<T>::~chain(){  chainNode<T>\* currentNode = pHead ->\_next;  chainNode<T>\* deleteNode;  while(currentNode != NULL){  deleteNode = currentNode;  currentNode = currentNode ->\_next;  delete deleteNode;  }  delete pHead;  }  template<typename T>  bool chain<T>::empty() const {return listSize == 0;}  template<typename T>  int chain<T>::size() const {return listSize;}  template<typename T>  int chain<T>::find(const T& theElement) const{  int index = 0;  chainNode<T>\* currentNode = pHead ->\_next;  while(currentNode != NULL){  if (currentNode->element == theElement) return index;  currentNode = currentNode->\_next;  ++index;  }  return -1;  }  template<typename T>  void chain<T>::erase(int theIndex){  checkIndex(theIndex);  chainNode<T>\* deleteNode;  chainNode<T>\* pre = pHead;  for (int i = 0; i < theIndex; ++i) pre = pre->\_next;  if (theIndex == listSize - 1) pTail = pre;  deleteNode = pre->\_next;  pre->\_next = pre->\_next->\_next;  --listSize;  delete deleteNode;  }  template<typename T>  void chain<T>::insert(int theIndex, const T& theElement){  if (theIndex < 0 || theIndex > listSize) throw out\_of\_range("illegalIndex");  chainNode<T>\* pre = pHead;  for (int i = 0; i < theIndex; ++i) pre = pre->\_next;  pre->\_next = new chainNode<T>(theElement, pre->\_next);  if (theIndex == listSize) pTail = pre->\_next;  ++listSize;  }  template<typename T>  void chain<T>::clear(){  chainNode<T>\* currentNode = pHead->\_next;  chainNode<T>\* deleteNode;  while(currentNode != NULL){  deleteNode = currentNode;  currentNode = currentNode->\_next;  delete deleteNode;  }  listSize = 0;  pHead ->\_next = NULL;  pTail = pHead;  }  template<typename T>  void chain<T>::push\_back(const T& theElement){  pTail->\_next = new chainNode<T>(theElement, pTail->\_next);  pTail = pTail->\_next;  listSize++;  }  template<typename T>  chain<T>& chain<T>::operator=(const chain<T>& c){  if (this == &c) return \*this;  clear();  chainNode<T>\* currentNode = pHead;  chainNode<T>\* sourceNode = c.pHead->\_next;  while (sourceNode != NULL)  {  pTail = currentNode->\_next = new chainNode<T>(sourceNode->element);  currentNode = currentNode->\_next;  sourceNode = sourceNode->\_next;  }  listSize = c.listSize;  return \*this;  }  template<typename T>  T& chain<T>::operator[](int index){  checkIndex(index);  chainNode<T>\* currentNode = pHead->\_next;  for (int i = 0; i < index; ++i) currentNode = currentNode->\_next;  return currentNode->element;  }  template<typename T>  const T& chain<T>::operator[](int index) const{  checkIndex(index);  chainNode<T> \*currentNode = pHead->\_next;  for (int i = 0; i < index ; ++i) currentNode = currentNode->\_next;  return currentNode->element;  }  template<typename T>  void chain<T>::checkIndex(int theIndex) const{  if (theIndex < 0 || theIndex >= listSize)  throw out\_of\_range("the index is out of range");  }  template<typename T>  class arrayQueue{  public:  arrayQueue(int = 10);  arrayQueue(const arrayQueue<T>&);  ~arrayQueue() {delete[]queue;}  bool empty() const {return queueTail == queueFront;}  int size() const {return (queueTail - queueFront + queueSize) % queueSize;}  T front() const;  T back() const;  void pop();  void push(const T&);  void clear();  arrayQueue<T>& operator=(const arrayQueue<T>&);  protected:  T\* queue;  int queueFront;  int queueTail;  int queueSize;  };  template<typename T>  arrayQueue<T>::arrayQueue(int initialCapcity){  if (initialCapcity < 1) throw invalid\_argument("Queue size must > 0");  queueSize = initialCapcity;  queueFront = queueTail = 0;  queue = new T[queueSize];  }  template<typename T>  arrayQueue<T>::arrayQueue(const arrayQueue<T>& q){  queueSize = q.queueSize;  queueFront = queueSize - 1;//??  queueTail = q.size() - 1;  queue = new T[queueSize];  for (int i = 0; i < q.size(); ++i) queue[i] = q.queue[(i + q.queueFront + 1) % queueSize];  }  template<typename T>  T arrayQueue<T>::front() const{  if (empty()) throw out\_of\_range("queue is empty");  return queue[(queueFront + 1) % queueSize];  }  template<typename T>  T arrayQueue<T>::back() const{  if (empty()) throw out\_of\_range("queue is empty");  return queue[queueTail];  }  template<typename T>  void arrayQueue<T>::pop(){  if (empty()) throw out\_of\_range("queue is empty");  queueFront++;  if (queueFront >= queueSize) queueFront -= queueSize;  queue[queueFront].~T();  }  template<typename T>  void arrayQueue<T>::push(const T& Ele){  if ((queueTail + 1) % queueSize == queueFront){  T\* newQueue = new T[2 \* queueSize];  int start = (queueFront + 1) % queueSize;  if (start < 2)  copy(queue + start, queue + start + queueSize - 1, newQueue);  else {  copy(queue + start, queue + queueSize, newQueue);  copy(queue, queue + queueTail + 1, newQueue + queueSize - start);  }  queueFront = 2 \* queueSize - 1;  queueTail = queueSize - 2;  queueSize \*= 2;  delete[] queue;  queue = newQueue;  }  queueTail++;  if (queueTail >= queueSize) queueTail -= queueSize;  queue[queueTail] = Ele;  }  template<typename T>  void arrayQueue<T>::clear(){  queueFront = queueTail = 0;  for (int i = 0; i < queueSize; ++i) queue[i].~T();  }  template<typename T>  arrayQueue<T>& arrayQueue<T>::operator=(const arrayQueue<T>& q){  if (this == &q) return \*this;  queueSize = q.queueSize;  queueFront = queueSize - 1;  queueTail = q.size() - 1;  delete[] queue;  queue = new T[queueSize];  for (int i = 0; i < q.size(); ++i) queue[i] = q.queue[(i + q.queueFront + 1) % queueSize];  return \*this;  }  template<typename T>  class linkedGraph {  protected:  chain<int>\* e;  bool\* vis;  int \*tmp, \*tmp2;  int n, m;  public:  int te, te2;  explicit linkedGraph(const int n) :n(n) {  e = new chain<int> [n + 1];  vis = new bool[n + 1];  tmp = new int[n + 1];  tmp2 = new int[n + 1];  m = 0;  }  ~linkedGraph() {  delete[] e;  delete[] vis;  }  void add(int u, int v) {  e[u].push\_back(v);  ++m;  }    void erase(int u, int v) {  auto i = 0;  for (auto it = e[u].begin(); it != e[u].end() && \*it != v; ++it, ++i);  e[u].erase(i);  --m;  }  int bfs(int s, int t)  {  memset(tmp2, 0, (n + 1) \* 4);  tmp2[s] = 0;  arrayQueue<int> q;  memset(vis, 0, n + 1);  vis[s] = true;    q.push(s);  while(!q.empty()) {  auto u = q.front(); q.pop();  tmp[++te] = u;  for (auto it = e[u].begin(); it != e[u].end(); ++it) {  auto v = \*it;  if (!vis[v]) {  tmp2[v] = tmp2[u] + 1;  q.push(v);  vis[v] = true;  }  }  }  return tmp2[t] ? tmp2[t] : -1;  }  void dfs(int u) {  tmp[++te] = u;  vis[u] = true;  for (auto it = e[u].begin(); it != e[u].end(); ++it) {  auto v = \*it;  if (!vis[v]) dfs(v);  }  }  void print(){  for (int i = 1; i <= te; ++i)  printf("%d ", tmp[i]);  printf("\n");  }  void print2(){  for (int i = 1; i <= te2; ++i)  printf("%d ", tmp2[i]);  printf("\n");  }  int scc(){  int res = 0;  memset(vis, 0, n + 1);  for (int i = 1; i <= n; ++i){  if (!vis[i]) tmp2[++res] = i, dfs(i);  }  te2 = res;  return res;  }  void solve(int u){  int p = e[u].size();  int \*node = new int[p + 1];  int i = 1;  for (auto it = e[u].begin(); it != e[u].end(); ++it, ++i){  node[i] = \*it;  }  sort(node + 1, node + p + 1);  i = 1;  for (auto it = e[u].begin(); it != e[u].end(); ++it, ++i){  \*it = node[i];  }  }  void presolve(){  for (int i = 1; i <= n; ++i) solve(i);  }  void re\_Vis(){  memset(vis, 0, n + 1);  }  };  int main(){  int n, m, s, t, op, u, v;  cin>>n>>m>>s>>t;  linkedGraph<int> G(n);  for (int i = 1; i <= m; ++i){  scanf("%d%d%d", &op, &u, &v);  if (op & 1) G.erase(u, v), G.erase(v, u);  else G.add(u, v), G.add(v, u);  }  G.presolve();  G.te = G.te2 = 0;  int scc = G.scc();  printf("%d\n", scc);  G.print2();  G.re\_Vis();  G.te = G.te2 = 0;  G.dfs(s);  printf("%d\n", G.te);  G.print();  G.te = 0;  int ans = G.bfs(t, s);  printf("%d\n", G.te);  G.print();  printf("%d\n", ans);  // system("pause");    }  /\*  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  \*/ | | | |