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：贪婪算法 | | | |
| 实验学时：2 | | 实验日期：2019.12.11 | |
| 实验目的：   1. 掌握最小生成树的Prim算法和Kruskal算法及其实现 | | | |
| 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. 创建加权无向图类。设图没有重边和自环，存储结构分别使用邻接矩阵和邻接链表。提供必要的基本操作。 3. 键盘输入图中顶点的个数n和边的数目e，以三元组（i，j，w）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，w）表示顶点i和顶点j之间拥有权值为w的边，建立图。 4. 对建立好的图，分别使用Prim算法和Kruskal算法求最小生成树，输出求得的最小生成树。 5. 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） 6. Prim算法：任意以一个点为开始，把这个初始点加入集合A中，从集合B中减去这个点。寻找与它相邻的点中路径最短的点，如后把这个点也加入集合A中,从集合B中减去这个点。更新未被访问的节点的dis[]值。重复上述过程。一直到所有的点都在A集合中结束。 7. Kruskal算法：将所有边储存下来，记录每条边关联的两个点的编号以及边权，按照边权从小到大排序。然后顺序遍历每条边，如果加入这条边不会产生环则将其加入生成树中，直至生成树中有n-1条边为止。 8. 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）   Prim算法与Dijkstra算法极为相似，Dijkstra算法dis记录的是到源点的最短路，Prim中的dis记录的是到生成树的最短路。   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<iostream>  #include<cstring>  #include<string>  #include<cstdio>  #include<algorithm>  using namespace std;  const int N = 101000, M = 400010;  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;  };  struct edge{  int u, v, w;  }e[M];  int n, m, q;  int fa[N], val[N];  inline int cmp(const edge &a, const edge &b) {return a.w < b.w; }  inline int getf(int x) {val[x] = val[fa[x]]; return fa[x] == x ? x : fa[x] = getf(fa[x]);}  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 {  typedef pair<int, T> p;  protected:  chain<p>\* e;  bool\* vis;  int n, 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) {  arrayQueue<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 << endl;  vis[u] = true;  for (auto it = e[u].begin(); it != e[u].end(); ++it) {  auto v = it ->first;  if (!vis[v]) dfs(v);  }  }  int dis(int s, int t) const {  T\* dis = new T[n + 1];  int\* vis = new int[n + 1];  memset(vis, 0, n \* 4);  for (auto i = 1; i <= n; ++i) dis[i] = 0x3f3f3f3f;  arrayQueue<int> q;  q.push(s);  dis[s] = 0;  while(!q.empty()) {  int u = q.front();  for (auto it = e[u].begin(); it != e[u].end(); ++it) {  int v = it ->first;  T& w = it ->second;  if (dis[u] + w < dis[v]){  dis[v] = dis[u] + w;  if (!vis[v]){  vis[v] = 1;  q.push(v);  }  }  }  }  auto ans = dis[t];  delete[] dis;  return ans;  }  };  inline void kruskal(){  sort(e + 1, e + m + 1, cmp);  for (int i = 1; i <= n; ++i) fa[i] = i;  for (int i = 1; i <= m; ++i){  int fx = getf(e[i].u), fy = getf(e[i].v);  if (fx != fy) {  fa[fy] = fx;  val[fx] += val[fy] + e[i].w;  val[fy] = val[fx];  }  }  }  int main(){  scanf("%d%d%d", &n, &m, &q);  for (int i = 1; i <= m; ++i) scanf("%d%d%d", &e[i].u, &e[i].v, &e[i].w);  kruskal();  for (int i = 1; i <= q; ++i){  int x;  scanf("%d", &x);  printf("%d\n", val[getf(x)]);  }  } | | | |