山东大学 软件 学院

数据结构 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201400301034 | 姓名： 石兴帮 | | 班级： 软件八班 |
| 实验题目：图的操作 | | | |
| 实验学时：4h | | 实验日期： 2015-12-13 | |
| 实验目的：  掌握图的基本概念，表述方法，遍历方法。 | | | |
| 硬件环境：  MacBook Pro  OS X Yosemite 10.10.3 | | | |
| 软件环境：  Xcode 6.4 | | | |
| 实验内容与设计：   1. 实验内容（题目内容，输入要求，输出要求）   1.创建图类。二叉树的存储结构使用邻接矩阵或邻接链表。  2.提供操作：遍历，BFS，DFS  3.对建立好的图，执行上述各操作。  4.输出生成树。  5.输出最小生成树。  2.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）  二叉树使用了邻接矩阵储存。  遍历操作：遍历二维数组，若当前位置元素值不为0，说明有边，输出横纵坐标。  BFS：BFS(int v , int reach[], int label)，参数v表示从点v开始遍历，reach用于保存元素，lebel用于标记已访问元素。从点v开始，将v放入队列，再提出，然后将与v相邻的点放入队列，同时打上label标记，直至没有点可以放入队列，BFS结束。  DFS：DFS(int v , int reach[], int label)，参数v表示从点v开始遍历，reach用于保存元素，lebel用于标记已访问元素。从点v开始，将v放入堆栈，再提出，然后将与v相邻的点放入堆栈，同时打上label标记，直至没有点可以放入堆栈，DFS结束。  生成树：在BFS／DFS的基础上，在将点A提出和将与点A相邻的点B放入的过程中间，访问A和B之间的线段。    最小生成树：分别用了普里姆算法和克鲁斯卡尔算法实现。  普里姆算法：先任意取一个顶点作为根节点，然后在边集中找到一条到这个顶点距离最小的边，将这条边的另一个顶点放入点集，与原来的点形成新的点集。之后在剩下的点中，不断寻找到这个点集距离最小的点，依次放进点集。  克鲁斯卡尔算法：将边排序，依次取出当前权值最小的边，若这条边的两个顶点属于不同的集合，则将这两个顶点所属的集合合并，直到凑够n－1条边。  3.测试结果（测试输入，测试输出）  **1  2**  **1  6**  **2  3**  **2  7**  **3  4**  **4  5**  **5  6**  **5  7**  **BFS:**  **包含顶点: 3**  **包含顶点: 4**  **包含顶点: 2**  **包含顶点: 5**  **包含顶点: 7**  **包含顶点: 1**  **包含顶点: 6**    **DFS:**  **包含顶点: 3**  **包含顶点: 2**  **包含顶点: 1**  **包含顶点: 6**  **包含顶点: 5**  **包含顶点: 7**  **包含顶点: 4**    **一棵宽度优先生成树:**  **(3 ,4)**  **(3 ,2)**  **(4 ,5)**  **(2 ,7)**  **(2 ,1)**  **(5 ,6)**    **加权图:**  **(1  ,2) value: 28**  **(1  ,6) value: 10**  **(2  ,3) value: 16**  **(2  ,7) value: 14**  **(3  ,4) value: 12**  **(4  ,5) value: 22**  **(4  ,7) value: 18**  **(5  ,6) value: 25**  **(5  ,7) value: 24**  **最小生成树(普里姆算法):**  **(1 ,6)**  **(5 ,6)**  **(4 ,5)**  **(3 ,4)**  **(2 ,3)**  **(2 ,7)**  Sum : 99  4.实现源代码（程序风格清晰易理解，有充分的注释）  //  //  shiyan8.h  //  c++初体验  //  Copyright (c) 2015年 apple. All rights reserved.  //    #ifndef c\_\_\_\_\_\_shiyan8\_h  #define c\_\_\_\_\_\_shiyan8\_h    #include<iostream>  #include "shiyan5.h"        //    需要用queue stack  using namespace std;      struct Edge  {      int row;      int col;      int weight;      bool selected;  };    void kuaisu(Edge shuzu[], int low, int high)  {      if(low >= high)      {          return;      }      int first = low;      int last = high;      int key = shuzu[first].weight;            while(first < last)      {          while(first < last && shuzu[last].weight >= key)          {              --last;          }            shuzu[first] = shuzu[last];/\*将比第一个小的移到低端\*/            while(first < last && shuzu[first].weight <= key)          {              ++first;          }            shuzu[last] = shuzu[first];          /\*将比第一个大的移到高端\*/      }      shuzu[first].weight = key;/\*枢轴记录到位\*/      kuaisu(shuzu, low, first-1);      kuaisu(shuzu, first+1, high);  }    Edge edge[100];             //边集和    int ranka[100];              //已找到的最小生成树其中一部分的秩    int parent[100];        //已找到的最小生成树其中一部分的头结点 用来判断一条边的2个端点是否在一个集合中，即加上这条边是否会形成回路    int find\_set(int x)         //找出每一集合的头结点  {      if(x != parent[x] )          parent[x] = find\_set(parent[x]);        return parent[x];    }    void union\_set(int x,int y,int w,int &sum)  {      if(x==y)          return;      if(ranka[x]>ranka[y])          parent[y]=x;      else      {          if(ranka[x]==ranka[y])              ranka[y]++;          parent[x]=y;      }      sum += w;  }      //  MARK:  邻接矩阵的基类  //  包含有向图，无向图，加权图的共同点。    template<class T>  void Make2DArray(T\*\* &x, int rows, int cols)  {  //    创建一个二维数组  //    不捕获异常    //    创建行指针      x = new T \*[rows];    //    为每一行分配空间      for (int i=0; i < rows; i++) {          x[i] = new int[cols];      }  }    template<class T>  void Delete2DArray(T\*\* &x, int rows)  {  //    删除二维数组  //    释放为每一行所分配的空间      for (int i=0; i < rows; i++)          delete [] x[i];  //        删除行指针      delete []x;      x = 0;    }    template<class T>  class AdjacencyWDigraph {      public:      AdjacencyWDigraph(int Vertices = 10, T noEdge = 0);      ~AdjacencyWDigraph() {Delete2DArray(a , n+1);}      bool Exist(int i, int j) const;      int Edges() const { return e;}      int Vertices() const { return n;}      AdjacencyWDigraph<T>& Add (int i, int j, const T& w);      AdjacencyWDigraph<T>& Delete(int i, int j);      int OutDegree(int i) const;      int InDegree(int i) const;          T NoEdge;   //  用于没有边存在的情形。在加权图中，各个权的值不同，若两点之间不通，用NoEdge表示。      int n;      //  顶点数目      int e;      //  边数      T \*\*a;      //  二维数组      Position \*poss ;      Edge \*posss;  };    template<class T>  AdjacencyWDigraph<T>::AdjacencyWDigraph(int Vertices, T noEdge)  {  //    构造函数      n = Vertices;      e = 0;      NoEdge = noEdge;      Make2DArray(a , n+1 , n+1);  //    初始化为没有边的图      poss = new Position[Vertices\*Vertices];      posss = new Edge[Vertices\*Vertices];      for (int i=1; i <= n; i++) {          for (int j=1; j <= n; j++) {              a[i][j] = NoEdge;          }      }  }    template<class T>  bool AdjacencyWDigraph<T>::Exist(int i, int j) const  {  //    边(i,j)存在      if (i<1 || j<1 || i>n || j>n || a[i][j] == NoEdge) {          return false;      }      return  true;  }    template<class T>  AdjacencyWDigraph<T>& AdjacencyWDigraph<T>::Add(int i, int j, constT& w)  {  //    如果边(i,j)不存在，则将该边加入到图中      if (i<1 || j<1 || i>n || j>n || i==j || a[i][j] != NoEdge) {          cout << "给的数据位置有问题，或是图中已经存在这条线。" << endl;      }      a[i][j] = w;      e++;      return \*this;  }    template <class T>  AdjacencyWDigraph<T>& AdjacencyWDigraph<T>::Delete(int i, int j)  {  //    删除边(i,j)      if (i<1 || j<1 || i>n || j>n || a[i][j] == NoEdge) {          cout << "给的数据位置有问题，或是图中本来就不存在这条线 " << endl;      }      a[i][j] = NoEdge;      e--;      return \*this;  }    template<class T>  int AdjacencyWDigraph<T>::OutDegree(int i) const  {      if (i<1 || i>n ) {  //        throw BadInput();      }  //    计算顶点i的出度      int sum = 0;      for (int j = 1; j <= n; j++) {          if (a[i][j] != NoEdge) {              sum++;          }      }      return sum;  }    template<class T>  int AdjacencyWDigraph<T>::InDegree(int i) const  {      if (i<1 || i>n ) {  //        throw BadInput();      }  //    计算顶点i的入度      int sum = 0;      for (int j = 1; j <= n; j++) {          if (a[j][i] != NoEdge) {              sum++;          }      }      return sum;  }      //  MARK:加权图  class AdjacencyWGraph : public AdjacencyWDigraph<int>  {  public:      AdjacencyWGraph(int Vertices = 10) : AdjacencyWDigraph<int>(Vertices, 0){}      AdjacencyWGraph& Add(int i, int j, const int w)      {          AdjacencyWDigraph<int>::Add(i, j, w);          int q = e;            edge[q].row = i;          edge[q].col = j;          edge[q].weight = w;          edge[q].selected = false;            parent[edge[q].row] = edge[q].row;          parent[edge[q].col] = edge[q].col;          ranka[edge[q].row] = 0;          ranka[edge[q].col] = 0;            return \*this;      }      void GetGraph();      void OutputGraph();        void GetMinTree();        void Prim();  };    void AdjacencyWGraph::Prim()  {      int \*dianji = new int[Vertices()+1];      Edge \*bianji = new Edge[Vertices()];      int count=0;      int e=0;      int minWeight = 100000;      bool canAdd = true;  //cout << edge[1].row << endl;        dianji[1] = edge[1].row;      int sum = 0;    for (int c=1; c<Vertices(); c++) {        for (int i=1; i<=Edges(); i++) {            for (int j=1; j<=Vertices(); j++) {                if (dianji[j] != 0 && (edge[i].row==dianji[j] || edge[i].col==dianji[j])) {                  canAdd = true;                  if (edge[i].row==dianji[j]) {                      for (int k=1; k<=Vertices(); k++)   {                          if (edge[i].col == dianji[k]) {  //cout << " 这条边row和当前点重复并且col也重复" << endl;                              canAdd = false;                          }                      }                  }                  else if (edge[i].col==dianji[j]) {                      for (int k=1; k<=Vertices(); k++) {                          if (edge[i].row == dianji[k]) {  //cout << " 这条边col和当前点重复并且row也重复" << endl;                              canAdd = false;                          }                      }                  }                    if (canAdd && edge[i].weight < minWeight) {                      minWeight = edge[i].weight;                      e = i;                  }              }            }  //cout << edge[i].weight << "..." << minWeight << endl;        }  //cout << e << endl;  //cout << "cao!!!!" << endl;      bianji[++count] = edge[e];      sum += edge[e].weight;      minWeight = 1000000;        bool addrow = true;      for (int i=1; i<=Vertices(); i++) {          if (dianji[i] == bianji[count].row) {              addrow = false;  //cout << "row重复，说明这条边加进去的是col   " << bianji[count].col << endl;              break;          }      }        bool addcol = true;      for (int i=1; i<=Vertices(); i++) {          if (dianji[i] == bianji[count].col) {              addcol = false;  //cout << "col重复，说明这条边加进去的是row   " << bianji[count].row << endl;              break;          }      }        if (addrow) {          dianji[count+1] = bianji[count].row;  //cout << "count+1:  " << count+1 << endl;  //cout << dianji[count+1] << endl;      } else      {          dianji[count+1] = bianji[count].col;  //cout << "count+1:  " << count+1 << endl;  //cout << dianji[count+1] << endl;      }    }      for (int i=1; i<Vertices(); i++) {          cout << "(" << bianji[i].row << " ," << bianji[i].col << ")" << endl;  //        cout << edge[i].row << " ," << edge[i].col << endl;      }      cout << "Sum : " << sum << endl;  }    void AdjacencyWGraph::GetMinTree(){      kuaisu(edge, 0, e);      int sum = 0;      for(int i=1;i<=e;i++)      {          int x,y;          x = find\_set(edge[i].row);          y = find\_set(edge[i].col);            //判断加上这条边是否会形成回路            if(x != y )          {              //选择这条边              edge[i].selected = true;              //合并不会形成回路的二个集合              union\_set(x,y,edge[i].weight,sum);          }      }        cout<<"最小生成树的边集为："<<endl;        for(int i=1;i<=e;i++)      {          if(edge[i].selected)          {              cout<<"(" << edge[i].row << " ,"<< edge[i].col << ")   weight:" << edge[i].weight <<endl;          }      }      cout<<"最小生成树的权值为："<<sum<<endl;  }    void AdjacencyWGraph::GetGraph()  {      int k=0;      for (int i=1; i<=n ; i++) {          for (int j=1 ; j<=n ; j++) {              if (a[i][j] != 0) {                  cout << "(" << i << "  ,"<< j << ") value: " << a[i][j] << endl;                  posss[k].row = i;                  posss[k].col = j;                  k++;              }          }      }  }    //  MARK:图  class AdjacencyGraph : public AdjacencyWDigraph<int>  {  public:      AdjacencyGraph(int Vertices = 10) : AdjacencyWDigraph<int>(Vertices, 0){}      AdjacencyGraph& Add(int i, int j)      {          AdjacencyWDigraph<int>::Add(i, j, 1);          return \*this;      }      AdjacencyGraph& Delete(int i, int j)      {          AdjacencyWDigraph<int>::Delete(i, j);          return \*this;      }          void GetGraph();      void OutputGraph();        int \*\*pos;      void InitializePos() { Make2DArray(pos, n+1, n+1);}        int Begin(int i);      int NextVertex(int i);        void BFS(int v, int reach[], int label);      void DFS(int v, int reach[], int label);        void BFSTree(int v , int reach[], int label);        void build(Position \*str, int\*tag, int n, int siz);      int DFSForTree(int v , int reach[] , int label);        void buildMin(Position \*str, int\*tag, int n, int siz);  };      void AdjacencyGraph::GetGraph()  {      int k=0;      for (int i=1; i<=n ; i++) {          for (int j=1 ; j<=n ; j++) {              if (a[i][j] != 0) {                  cout << i << "  "<< j<< endl;                  poss[k].row = i;                  poss[k].col = j;                  k++;              }          }      }      }    void AdjacencyGraph::BFS(int v , int reach[], int label)  {  //    宽度优先搜索      Queue<int> \*q = new Queue<int>(100);      reach[v] = label;      q->Add(v );      while (!q->IsEmpty()) {          int w;          q->Delete(w);          cout << "包含顶点: " ;          cout << w << " " << endl;  //        对尚未标记的，邻接自w的顶点进行标记          for (int u=1; u <= n ; u++) {              if (a[w][u] != NoEdge && !reach[u]) {                  q->Add(u);                  reach[u] = label;              }          }          for (int t=1; t <= w; t++) {              if (a[t][w] != NoEdge && !reach[t]) {                  q->Add(t);                  reach[t] = label;              }          }      }  }    void AdjacencyGraph::DFS(int v , int reach[] , int label)  {  //    深度优先搜索      Stack<int> \*s = new Stack<int>(100);      reach[v] = label;      s->Add(v );      while (!s->IsEmpty()) {          int w;          s->Delete(w);          cout << "包含顶点: ";          cout << w << " " << endl;  //        对尚未标记的，邻接自w的顶点进行标记          for (int u=1; u <= n ; u++) {              if (a[w][u] != NoEdge && !reach[u]) {                  s->Add(u);                  reach[u] = label;              }          }          for (int t=1; t <= w; t++) {              if (a[t][w] != NoEdge && !reach[t]) {                  s->Add(t);                  reach[t] = label;              }          }      }  }    void AdjacencyGraph::BFSTree(int v , int reach[], int label)  {      //    宽度优先搜索      Queue<int> \*q = new Queue<int>(100);      reach[v] = label;      q->Add(v );      while (!q->IsEmpty()) {          int w;          q->Delete(w);  //        cout << "包含顶点: " ;  //        cout << w << " " << endl;          //        对尚未标记的，邻接自w的顶点进行标记          for (int u=1; u <= n ; u++) {              if (a[w][u] != NoEdge && !reach[u]) {                  cout << "(" << w << " ," << u << ")" << endl;                  q->Add(u);                  reach[u] = label;              }          }          for (int t=1; t <= w; t++) {              if (a[t][w] != NoEdge && !reach[t]) {                  cout << "(" << w << " ," << t << ")" << endl;                  q->Add(t);                  reach[t] = label;              }          }      }  }      int AdjacencyGraph::DFSForTree(int v , int reach[] , int label)  {      //    深度优先搜索for 生成树      int sum = 0;        Stack<int> \*s = new Stack<int>(100);      reach[v] = label;      s->Add(v );      while (!s->IsEmpty()) {          int w;          s->Delete(w);          sum += w;          //        对尚未标记的，邻接自w的顶点进行标记          for (int u=1; u <= n ; u++) {              if (a[w][u] != NoEdge && !reach[u]) {                  s->Add(u);                  reach[u] = label;              }          }          for (int t=1; t <= w; t++) {              if (a[t][w] != NoEdge && !reach[t]) {                  s->Add(t);                  reach[t] = label;              }          }      }      return sum;  }      static int treeCount = 1;    void AdjacencyGraph::build(Position \*str, int\*tag, int n, int siz)  {  //    cout << " ???" << endl;      if (n == siz) {            AdjacencyGraph \*tmp = new AdjacencyGraph(Vertices());          int \*biaoji = new int[Vertices()+1];          for (int i=1; i<=Vertices(); i++) {              biaoji[i] = 1;          }            for (int i=0; i<siz; i++) {              if (tag[i] == 1) {                  tmp->Add(str[i].row, str[i].col);                  biaoji[str[i].row] = 0;                  biaoji[str[i].col] = 0;              }          }            bool flag = true;          for (int i=1; i<=Vertices(); i++) {              if (biaoji[i] == 1) {                  flag = false;              }          }            if (flag != true) {              return;          }            int sum=0; int \*reach = new int[Vertices()+1];          for (int i=1; i<=Vertices(); i++) {              sum+=i;          }            bool flag2 = false;          if (sum == tmp->DFSForTree(1, reach, 100)) {              flag2 = true;          }          if (flag == true && flag2) {              cout << "第 " << treeCount << "棵生成树： " << endl;              treeCount++;              for (int i=0; i<siz; i++) {                  if (tag[i] == 1) {                      cout << str[i].row << "   " << str[i].col << endl;                  }              }          }            return;      }      tag[n] = 0;      build(str, tag, n+1, siz);      tag[n] = 1;      build(str, tag, n+1, siz);  }    static int mintreeCount = 1;  static int mintreeSize = 10000;    void AdjacencyGraph::buildMin(Position \*str, int\*tag, int n, intsiz)  {      if (n == siz) {          int treesize = 0;          AdjacencyGraph \*tmp = new AdjacencyGraph(Vertices());          int \*biaoji = new int[Vertices()+1];          for (int i=1; i<=Vertices(); i++) {              biaoji[i] = 1;          }            for (int i=0; i<siz; i++) {              if (tag[i] == 1) {                  tmp->Add(str[i].row, str[i].col);                  biaoji[str[i].row] = 0;                  biaoji[str[i].col] = 0;                  treesize++;              }          }            bool flag = true;          for (int i=1; i<=Vertices(); i++) {              if (biaoji[i] == 1) {                  flag = false;              }          }            if (flag != true) {              return;          }            int sum=0; int \*reach = new int[Vertices()+1];          for (int i=1; i<=Vertices(); i++) {              sum+=i;          }            bool flag2 = false;          if (sum == tmp->DFSForTree(1, reach, 100)) {              flag2 = true;          }          if (flag == true && flag2 && treesize<= mintreeSize) {              mintreeSize = treesize;              cout << "第 " << mintreeCount << "棵最小生成树： " << endl;              mintreeCount++;              for (int i=0; i<siz; i++) {                  if (tag[i] == 1) {                      cout << str[i].row << "   " << str[i].col << endl;                  }              }          }            return;      }      tag[n] = 0;      buildMin(str, tag, n+1, siz);      tag[n] = 1;      buildMin(str, tag, n+1, siz);  }      #endif      //#include "shiyan8.h"    //int main()  //{  //    AdjacencyGraph \*ag = new AdjacencyGraph(7);  //    ag->Add(1, 2);  //    ag->Add(1, 6);  //    ag->Add(2, 3);  //    ag->Add(2, 7);  //    ag->Add(3, 4);  //    ag->Add(4, 5);  //    ag->Add(5, 7);  //    ag->Add(5, 6);  //  //      //    ag->GetGraph();      //      //    int \*reach = new int[ag->Vertices()+1];      //    cout << " BFS: " << endl;      //    ag->BFS(3, reach, 100);      //    cout << endl;      //      //    cout << " DFS: " <<endl;      //    int \*reach2 = new int[ag->Vertices()+1];      //    ag->DFS(3, reach2 , 100);      //    cout << endl;      //      //    cout << " 一棵宽度优先生成树: " << endl;      //    int \*reach3 = new int[ag->Vertices()+1];      //    ag->BFSTree(3, reach3 , 100);      //    cout << endl;  //  //    AdjacencyWGraph \*awg = new AdjacencyWGraph(7);  //    awg->Add(1, 6 ,10);  //    awg->Add(1, 2 ,28);  //    awg->Add(2, 3 ,16);  //    awg->Add(2, 7 ,14);  //    awg->Add(3, 4 ,12);  //    awg->Add(4, 5 ,22);  //    awg->Add(4, 7 ,18);  //    awg->Add(5, 6 ,25);  //    awg->Add(5, 7, 24);  //    //    awg->Add(1, 2, 1);  //    //    awg->Add(1, 3, 2);  //    //    awg->Add(1, 4, 3);  //    //    awg->Add(2, 3, 4);  //    //    awg->Add(3, 4, 5);  //  //    cout << "加权图: " << endl;  //    awg->GetGraph();  //  //    //    最小生成树 （普里姆算法）  //    cout << "最小生成树(普里姆算法):" << endl;  //    awg->Prim();  //  //    //    最小生成树 （克鲁斯卡尔算法）（似乎有点小问题。。。）  //    //    awg->GetMinTree();  //  //  //    //    下面这三行其实是输出生成子图和最小生成子图Orz，所以点h文件里的build和buildMin就不要看了。。  //    //    int tag[ag->Edges()];  //    //    ag->build(ag->poss, tag, 0, ag->Edges());  //    //    ag->buildMin(ag->poss, tag, 0, ag->Edges());  //  //} | | | |
| 结论分析与体会：  一开始把输出生成树写普里姆算法的时候出了点小bug，改了一段时间之后改好了，挺好玩的。 | | | |