数据结构与算法 II 上机实验 (11.17)

中国人民大学　信息学院　崔冠宇　2018202147

注：请使用支持 C++17 或以上标准的编译器！

上机题 1　实现 Huffman 算法。

一、问题描述

利用 Huffman 树的贪心构建算法，根据给定频率表构建 Huffman 树，并可以进行编码和译码。

1. 输入：字符-频率表 $T =<< c_1, f_1 >, < c_2, f_2 >, \cdots, < c_n, f_n >>$（其中 $c_i \in \Sigma$ 是单字符，$f_i \in \mathbb{R}$ 表示各字符出现频率）。
2. 输出：字符-编码表 $T' =<< c_1, s_1 >, < c_2, s_2 >, \cdots, < c_n, s_n >>$（其中 c_i 同上，$s_i \in \{0,1\}^*$ 是 c_i 对应的 Huffman 编码）。

二、算法基本思路

在课堂上，我们已经证明过了 Huffman 算法具有最优子结构性质与贪心选择性质（具体证明此处不再赘述），所以可以用贪心算法构建 Huffman 树和 Huffman 编码。

下面给出构建 Huffman 编码的伪代码（其中用到了一些有关二叉树的操作）：

```
// Huffman 算法
// 参数:
//      T: 字符-频率表
// 返回值:
//      HTree: Huffman 树
HUFFMAN-TREE(T):
    // 优先队列，里面存放各子树的根节点，按权值递增
    pQueue = Priority-Queue()
    // 初始节点
    for i = 0 to T.size() - 1:
        // 将各字符-频率对变为树的节点
        node = BiTreeNode(T[i].c, T[i].f)
        pQueue.INSERT(node)
    // 一直合并，直到仅剩一个节点
    while pQueue.size() >= 2:
        // 弹出两个节点
        lNode = pQueue.EXTRACT-MIN()
        rNode = pQueue.EXTRACT-MIN()
```

```
        // 合并两个节点
        newRoot = BiTreeNode('\0', lNode.f + rNode.f)
        newRoot.setLChild(lNode)
        newRoot.setRChild(rNode)
        // 放回队列中
        pQueue.INSERT(newRoot)
    HTree = pQueue.EXTRACT-MIN()
    return HTree
```

构建完 Huffman 树之后，再给出根据 Huffman 树输出 Huffman 编码表的伪代码：

```
// 遍历树以获得 Huffman 编码
// 参数：
//      codeTable: 字符-编码表 (key: 字符, value: 编码0/1串)
//      s: 遍历至此的0/1串
//      nowRoot: 当前子树的根节点
parseTable(&codeTable, &s, nowRoot):
    // 到叶节点
    if nowRoot.isLeaf():
        // 获得一个字符的编码
        codeTable.insert(nowRoot.c, &s)
        s.pop_back()
        return
    // 递归左子树
    s += "0"
    parseTable(codeTable, s, nowRoot.lChild)
    // 递归右子树
    s += "1"
    parseTable(codeTable, s, nowRoot.rChild)
    // 上一层的编码
    s.pop_back()

// 获得 Huffman 编码
// 参数：
//      HTree: Huffman 树
// 返回值：
//      CTable: 字符-编码表
HUFFMAN-CODE(HTree):
```

```
    CTable = std::map<char, std::string>()
    path = ""
    parseTable(CTable, path, HTree.root)
    return CTable
```

三、算法复杂性分析

记问题规模 n 为节点的个数。

先分析从字符-频率表构建 Huffman 树的 HUFFMAN-TREE 算法的复杂度：

1. 时间复杂度：
   算法首先向优先队列中连续插入 n 个节点，可以将其等效为将 n 个元素的数组建成最小堆，所以该操作的时间是 $O(n)$ 的（也可以利用聚合分析的方法来分析，结果相同）。

   接下来算法进行了 $n-1$ 次合并，每次取出队头两个元素，将其合并成一个节点，然后插回队列。在队列中还有 $k \le n$ 个元素时，一次上述操作的时间是 $O(\log k) + O(\log(k-1)) + O(\log(k-2)) = O(\log k)$，对 k 求和，可以得到合并步骤的时间复杂度是 $O(n \log n)$ 的。

   综上，整个 HUFFMAN-TREE 算法的时间复杂度是 $T(n) = O(n \log n)$。

2. 空间复杂度：
   算法工作时需要用优先队列保存各个节点的信息，所以算法的空间复杂度是 $S(n) = O(n)$ 的。

再分析从 Huffman 树构建 Huffman 编码表的时间复杂度：

HUFFMAN-CODE 算法本质上是调用了 parseTable 函数进行二叉树的遍历，同时动态维护遍历至今的路径以生成 Huffman 编码。因为 Huffman 树外节点比内节点多一个，当有 n 个字符时，树总共有 $n + (n-1) = 2n - 1$ 个节点，而遍历树会遍历每个节点，所以算法时间复杂度 $T(n) = O(n)$（忽略向编码表插入的时间）。

四、程序源代码

二叉树定义及相关操作: **bitree.h**:

```
#ifndef BITREE_H
#define BITREE_H

#include <cstddef>
#include <algorithm>
#include <functional>

// 二叉树的节点
```

```cpp
template <typename T>
class BiTreeNode
{
    public:
        // 构造析构
        BiTreeNode(T _data);
        ~BiTreeNode();
        // 判断是否是叶子
        bool isLeaf() const;
        // 左子节点
        void setLChild(BiTreeNode<T> * l);
        BiTreeNode<T> * getLChild();
        // 右子节点
        void setRChild(BiTreeNode<T> * r);
        BiTreeNode<T> * getRChild();
        // 值操作
        void setData(T _newData);
        const T & getData() const;
    private:
        // 孩子与父亲
        BiTreeNode<T> * lChild;
        BiTreeNode<T> * rChild;
        BiTreeNode<T> * parent;
        // 数据
        T data;
};

// 构造函数
template <typename T>
BiTreeNode<T>::BiTreeNode(T _data)
:data(_data), parent(nullptr), lChild(nullptr), rChild(nullptr){}

template <typename T>
BiTreeNode<T>::~BiTreeNode()
{}

template<typename T>
```

```
46 bool BiTreeNode<T>::isLeaf() const
47 {
48     return ((lChild == nullptr) && (rChild == nullptr));
49 }
50
51 template <typename T>
52 void BiTreeNode<T>::setLChild(BiTreeNode<T> * l)
53 {
54     lChild = l;
55     l -> parent = this;
56 }
57
58 template <typename T>
59 BiTreeNode<T> * BiTreeNode<T>::getLChild()
60 {
61     return lChild;
62 }
63
64 template <typename T>
65 void BiTreeNode<T>::setRChild(BiTreeNode<T> * r)
66 {
67     rChild = r;
68     r -> parent = this;
69 }
70
71 template <typename T>
72 BiTreeNode<T> * BiTreeNode<T>::getRChild()
73 {
74     return rChild;
75 }
76
77 template <typename T>
78 void BiTreeNode<T>::setData(T _newData)
79 {
80     data = _newData;
81 }
82
```

```cpp
template <typename T>
const T & BiTreeNode<T>::getData() const
{
    return data;
}

// 二叉树
template <typename T>
class BiTree
{
    public:
        // 构造析构
        BiTree();
        BiTree(BiTreeNode<T> * root);
        ~BiTree();
        void clear();
        //清除以_root为根的子树
        void clear(BiTreeNode<T> * _root);
        void setRoot(BiTreeNode<T> * _root);
        BiTreeNode<T> * getRoot() const;

        bool isEmpty();
        unsigned int depth();
        //以_root为根的子树的深度
        unsigned int depth(BiTreeNode<T> * _root);
        //先序遍历
        void preOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit);
        void preOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit,
    BiTreeNode<T> * _root);
        //中序遍历
        void inOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit);
        void inOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit, BiTreeNode
    <T> * _root);
        //后序遍历
        void postOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit);
        void postOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit,
    BiTreeNode<T> * _root);
```

```
117     private:
118         BiTreeNode<T> * root;
119 };
120 
121 template <typename T>
122 BiTree<T>::BiTree()
123 :root(nullptr){}
124 
125 template <typename T>
126 BiTree<T>::BiTree(BiTreeNode<T> * _root)
127 :root(_root){}
128 
129 template <typename T>
130 BiTree<T>::~BiTree()
131 {
132     clear(root);
133 }
134 
135 template <typename T>
136 void BiTree<T>::clear()
137 {
138     clear(root);
139 }
140 
141 template <typename T>
142 void BiTree<T>::clear(BiTreeNode<T> * _root)
143 {
144     if(_root == nullptr)
145     {
146         return;
147     }
148 
149     this -> clear(_root -> getLChild());
150     this -> clear(_root -> getRChild());
151     delete _root;
152 }
153 
```

```
154 template <typename T>
155 void BiTree<T>::setRoot(BiTreeNode<T> * _root)
156 {
157     clear();
158     root = _root;
159 }
160
161 template <typename T>
162 BiTreeNode<T> * BiTree<T>::getRoot() const
163 {
164     return root;
165 }
166
167 template <typename T>
168 bool BiTree<T>::isEmpty()
169 {
170     return (root == nullptr);
171 }
172
173 template <typename T>
174 unsigned int BiTree<T>::depth()
175 {
176     return depth(root);
177 }
178
179 template <typename T>
180 unsigned int BiTree<T>::depth(BiTreeNode<T> * _root)
181 {
182     if(_root == nullptr)
183     {
184         return 0;
185     }
186     unsigned int left = depth(_root -> getLChild());
187     unsigned int right = depth(_root -> getRChild());
188     return std::max(left, right) + 1;
189 }
190
```

```cpp
template <typename T>
void BiTree<T>::preOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit)
{
    preOrderTraverse(visit, this -> root);
}

template <typename T>
void BiTree<T>::preOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit,
    BiTreeNode<T> * _root)
{
    if(_root == nullptr)
    {
        return;
    }
    visit(_root);
    preOrderTraverse(visit, _root -> getLChild());
    preOrderTraverse(visit, _root -> getRChild());
}

template <typename T>
void BiTree<T>::inOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit)
{
    inOrderTraverse(visit, this -> root);
}

template <typename T>
void BiTree<T>::inOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit,
    BiTreeNode<T> * _root)
{
    if(_root == nullptr)
    {
        return;
    }
    preOrderTraverse(visit, _root -> getLChild());
    visit(_root);
    preOrderTraverse(visit, _root -> getRChild());
}
```

```
226
227 template <typename T>
228 void BiTree<T>::postOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit)
229 {
230     postOrderTraverse(visit, this -> root);
231 }
232
233 template <typename T>
234 void BiTree<T>::postOrderTraverse(std::function<bool(BiTreeNode<T> *)> visit,
        BiTreeNode<T> * _root)
235 {
236     if(_root == nullptr)
237     {
238         return;
239     }
240     preOrderTraverse(visit, _root -> getLChild());
241     preOrderTraverse(visit, _root -> getRChild());
242     visit(_root);
243 }
244
245 #endif
```

Huffman 树的定义及相关操作: **huffman.h**:

```
1  #ifndef HUFFMAN_H
2  #define HUFFMAN_H
3
4  #include <iostream>
5  #include <string>
6  #include <vector>
7  #include <map>
8  #include <queue>
9  #include "bitree.h"
10
11 // 字符-权值对
12 typedef std::pair<char, double> CWPair;
13
14 // 节点大小比较
```

```
15 class HNodeGreater
16 {
17     public:
18         bool operator()(const BiTreeNode<CWPair> * A, const BiTreeNode<CWPair> * B)
19         {
20             return A -> getData().second > B -> getData().second;
21         }
22 };
23
24 // Huffman 树
25 class HuffmanCode
26 {
27     public:
28         // 构造
29         HuffmanCode(const std::vector<std::pair<char, double> > &);
30         //打印字母表
31         void printTable();
32         //编码
33         std::string getCode(const std::string &);
34         //译码
35         std::string getOriginal(const std::string &);
36
37     private:
38         //内部的二叉树
39         BiTree<CWPair> tree;
40         //编码出来的表
41         std::map<char, std::string> codeTable;
42         //遍历树获得字母编码表
43         void parseTable(std::string & path, BiTreeNode<CWPair> * nowRoot);
44 };
45
46 HuffmanCode::HuffmanCode(const std::vector<std::pair<char, double> > & freq)
47 {
48     //优先队列
49     std::priority_queue<BiTreeNode<CWPair> *, std::vector<BiTreeNode<CWPair> *>,
    HNodeGreater> pQueue;
50     //初始节点
```

```
51      for(size_t i = 0; i < freq.size(); i++)
52      {
53          pQueue.push(new BiTreeNode<CWPair>(freq.at(i)));
54      }
55      BiTreeNode<CWPair> * newRoot = nullptr;
56      //两个以上节点才有必要
57      while(pQueue.size() >= 2)
58      {
59          //弹出两个权值最小的
60          BiTreeNode<CWPair> * l = pQueue.top();
61          pQueue.pop();
62          BiTreeNode<CWPair> * r = pQueue.top();
63          pQueue.pop();
64          //组合成为一棵新树
65          CWPair rootPair = {'\0', l -> getData().second + r -> getData().second};
66          newRoot = new BiTreeNode<CWPair>(rootPair);
67          newRoot -> setLChild(l);
68          //l -> setParent(newRoot);
69          newRoot -> setRChild(r);
70          //r -> setParent(newRoot);
71          //插回去
72          pQueue.push(newRoot);
73      }
74      //处理完之后
75      tree.setRoot(newRoot);
76      std::string path;
77      parseTable(path, newRoot);
78  }
79
80  void HuffmanCode::parseTable(std::string & path, BiTreeNode<CWPair> * nowRoot)
81  {
82      //遍历来获得
83      if(nowRoot -> isLeaf())
84      {
85          codeTable.insert(std::pair<char, std::string>(nowRoot -> getData().first,
        path));
86          path.pop_back();
```

```
87          return;
88      }
89      //递归
90      //左边
91      path += "0";
92      parseTable(path, nowRoot -> getLChild());
93
94      //右边
95      path += "1";
96      parseTable(path, nowRoot -> getRChild());
97
98      //处理完左右向上一层
99      path.pop_back();
100 }
101
102 void HuffmanCode::printTable()
103 {
104     for(std::map<char, std::string>::iterator i = codeTable.begin(); i != codeTable
        .end(); i++)
105     {
106         std::cout << i -> first << " " << i -> second << std::endl;
107     }
108 }
109
110 std::string HuffmanCode::getCode(const std::string & str)
111 {
112     std::string retVal;
113     for(size_t i = 0; i < str.size(); i++)
114     {
115         retVal += codeTable.at(str.at(i));
116     }
117     return retVal;
118 }
119
120 std::string HuffmanCode::getOriginal(const std::string & str)
121 {
122     std::string retVal;
```

```
123     BiTreeNode<CWPair> * nowNode = tree.getRoot();
124     for(size_t i = 0; i < str.size(); i++)
125     {
126         if(nowNode -> isLeaf())
127         {
128             retVal.push_back(nowNode -> getData().first);
129             nowNode = tree.getRoot();
130         }
131         if(str.at(i) == '0')
132         {
133             nowNode = nowNode -> getLChild();
134         }
135         else if(str.at(i) == '1')
136         {
137             nowNode = nowNode -> getRChild();
138         }
139     }
140     //最后必然是叶子
141     if(nowNode -> isLeaf())
142     {
143         retVal.push_back(nowNode -> getData().first);
144         nowNode = tree.getRoot();
145     }
146     else
147     {
148         std::out_of_range e("编码意外终止");
149         throw e;
150     }
151     return retVal;
152 }
153 #endif
```

Huffman 树的使用: **huffman.cpp**:

```
1 #include <iostream>
2 #include <vector>
3 #include <utility>
4
```

```cpp
#include <fstream>
#include "huffman.h"

// 从文件读取字符-权值
bool readFile(const std::string & fileName, std::vector<std::pair<char, double> > & dest)
{
    std::fstream file(fileName, std::fstream::in);
    if(!file.is_open())
    {
        std::cout << "打不开文件!" << std::endl;
        return false;
    }
    while(!file.eof())
    {
        char c;
        double w;
        file.get(c);
        file.ignore();
        file >> w;
        file.ignore();
        dest.push_back(CWPair(c, w));
    }
    return true;
}

// 从终端读取字符-权值表
bool readConsole(std::vector<std::pair<char, double> > & dest)
{
    std::cout << "请输入字母表字符数:";
    int count;
    std::cin >> count;
    if(count <= 0)
    {
        std::cout << "参数不合法!" << std::endl;
        return false;
    }
```

```
41      std::cout << "请按行输入字母表以及对应权值(e.g. A 0.08167): " << std::endl;
42
43      for(unsigned int i = 0; i < count; i++)
44      {
45          char c;
46          double w;
47          //忽略回车
48          std::cin.ignore();
49          c = std::getchar();
50          std::cin.ignore();
51          std::cin >> w;
52          dest.push_back(CWPair(c, w));
53      }
54      return true;
55  }
56
57
58  int main(int argc, char * argv[])
59  {
60      std::cout << "Huffman Code." << std::endl;
61      std::cout << "Written By G.Cui" << std::endl;
62      std::vector<std::pair<char, double> > freqVec;
63      while(true)
64      {
65          std::cout << "请选择输入方式(f: 文件, c: 控制台):";
66          std::string input;
67          std::cin >> input;
68          if(input == "f")
69          {
70              std::string fName;
71              std::cout << "请输入读取文件名:";
72              std::cin >> fName;
73              //成功->继续
74              if(readFile(fName, freqVec))
75              {
76                  std::cout << "读取成功!" << std::endl;
77                  break;
```

```
78              }
79          }
80          else if(input == "c")
81          {
82              if(readConsole(freqVec))
83              {
84                  std::cout << "读取成功!" << std::endl;
85                  break;
86              }
87          }
88      }
89      HuffmanCode coder(freqVec);
90      coder.printTable();
91      while(true)
92      {
93          std::string str;
94
95          std::cout << "请选择操作(0: 退出, 1: 编码, 2: 译码): ";
96          std::string op;
97          std::cin >> op;
98
99          //退出
100         if(op == "0")
101         {
102             break;
103         }
104         //编码
105         else if(op == "1")
106         {
107             std::cout << "请输入欲编码的字符串:";
108             //std::cin >> str;
109             //空白符
110             std::cin >> std::ws;
111             std::getline(std::cin, str);
112             try
113             {
114                 std::cout << "结果:" << coder.getCode(str) << std::endl;
```

```
            }
            catch(const std::out_of_range &)
            {
                std::cout << "非法字符!" << std::endl;
            }
        }
        //译码
        else if(op == "2")
        {
            std::cout << "请输入欲译码的字符串(仅限01字符串):";
            std::cin >> str;
            try
            {
                std::cout << "结果:" << coder.getOriginal(str) << std::endl;
            }
            catch(const std::out_of_range &)
            {
                std::cout << "非法编码!" << std::endl;
            }
        }
        else
        {
            std::cout << "非法输入, 请重新选择!" << std::endl;
        }
    }

    return 0;
}
```

五、运行结果截图

首先将课件中的权值作为测试用例输入文件:

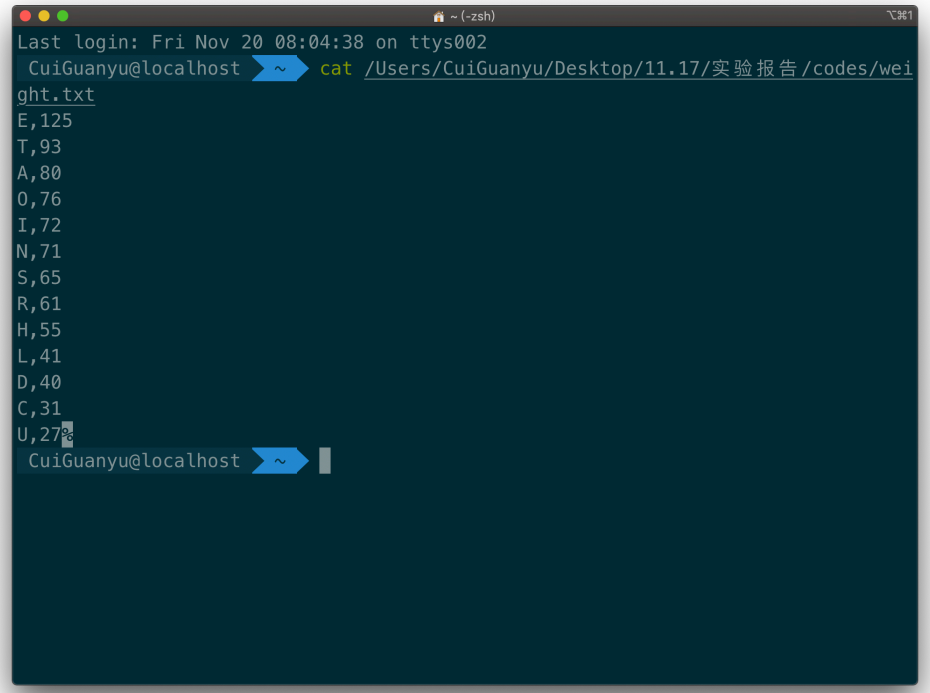

图 1: 权值文件

编译运行 huffman.cpp，将权值文件输入做测试：

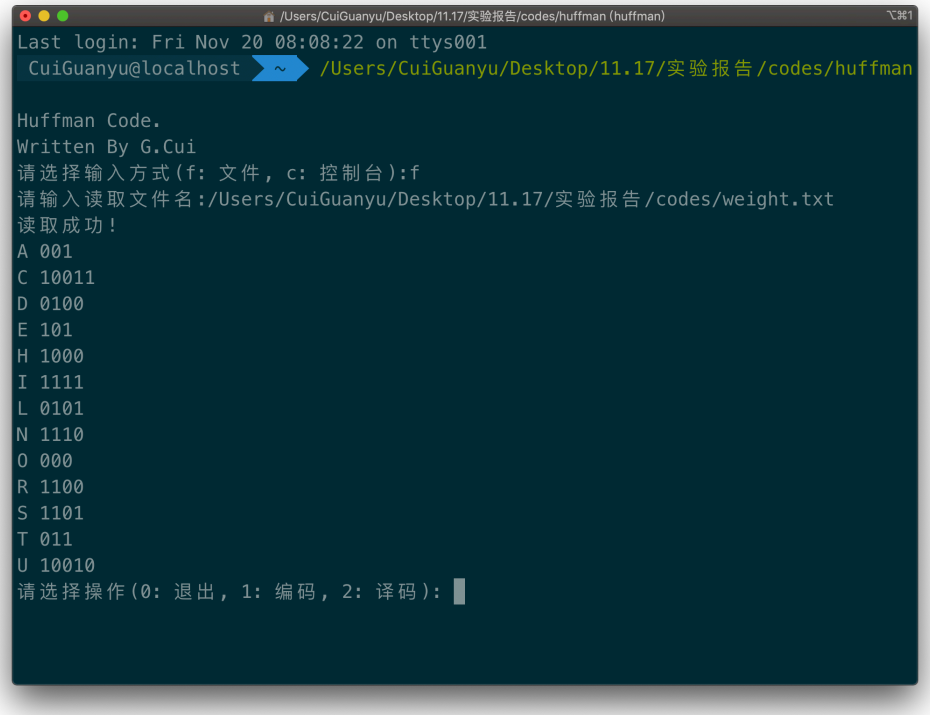

图 2: Huffman 编码测试

观察可见程序运行正确（由于老师的课件上取出两个节点后左右节点的大小关系并没有指定，所以与程序输出

略有不同，但都是最优的）。此外，该程序还支持 Huffman 编码和译码，此处不再展示。

上机题 2 实现 Dijkstra 算法。

一、问题描述

利用 Dijkstra 算法，根据给定的图的邻接矩阵和源，计算单源最短路径长度以及各最短路径。

1. 输入：图的邻接矩阵 **G**，源 s。
2. 输出：数组 $dist$（其中 $dist[j]$ 表示源 s 到顶点 j 的最短路径长度），数组 $from$（其中 $from[j]$ 表示源 s 到顶点 j 的最短路径上 j 之前的节点）。

二、算法基本思路

在课堂上，我们已经证明过了 Dijkstra 算法（具体证明此处不再赘述），所以可以用贪心算法得到单源最短路径。

下面给出 Dijkstra 算法的伪代码：

```
// Dijkstra 单源最短路径
// 参数:
//       G[][]: 图带权邻接矩阵
//       s: 源
// 返回值:
//       dist[]: 最短距离数组
//       from[]: 最短路径上前一个顶点数组
Dijkstra(G, s):
    // 顶点数
    n = |V(G)|
    // 初始化
    // 是否找到最短路的数组
    found[1..i..n] = [false]
    // 路径长
    dist[1..j..n] = [G[s][j]]
    // 上一个顶点数组
    from[1..k..n] = [s if G[s][j] < +INF else 0]
    // 标记源点
    dist[s] = 0
    found[s] = true
    // 进行 n 轮
    for i = 1 to n:
```

```
        // 找寻最小的 dist
        minDist = +INF
        u = s
        for j = 1 to n:
            if not found[j] and dist[j] < minDist:
                u = j
                minDist = dist[j]

        found[u] = true
        // 更新距离
        for j = 1 to n:
            if not found[j] and G[u][j] < +INF:
                newDist = dist[u] + G[u][j]
                if newDist < dist[j]:
                    dist[j] = newDist
                    from[j] = u
    return dist, from
```

还可以利用 *from* 数组构建源到各节点的最短路径:

```
// 递归打印路径
// 参数:
//      from: Dijkstra 算法返回的 from 数组
//      src: 源
//      dest: 终点
// 输出:
//      一条从源到终点的最短路径
printPath(from, src, dest):
    // 已经到源
    if dest == src:
        print(src)
        return
    // 递归前面的
    printPath(from, src, from[dest])
    // 本节点
    print(" -> ", dest)
// 递归打印路径
// 参数:
```

```
//       from: Dijkstra 算法返回的 from 数组
//       src: 源
// 输出:
//       对于所有终点，一条从源到终点的最短路径
printAllPath(from, src):
    for i = 1 to n:
        printPath(from, src, i)
```

三、算法复杂性分析

给定有向图 $G = (V, E)$，问题规模是 $|V|$ 以及 $|E|$。

先分析 `Dijkstra` 的复杂度：

1. 时间复杂度：
   算法有两层循环，每层循环都进行了 $|V|$ 次，所以在给定图的邻接矩阵时，Dijkstra 算法的时间复杂度为 $T(|V|, |E|) = O(|V|^2)$。
2. 空间复杂度：
   算法除去邻接矩阵外，还使用了三个长为 $|V|$ 的数组，所以算法的空间复杂度为 $S(|V|, |E|) = O(|V|)$。

再来分析 `printPath` 和 `printAllPath` 的时间复杂度：

`printPath` 算法的基本操作是打印一个节点。由于在单源最短路径的语境下，从源到终点的最短路最多只会经过所有 $|V|$ 个顶点，因此该算法的时间复杂度 $T(|V|, |E|) = O(|V|)$。

`printAllPath` 算法是以所有节点作为终点运行 `printPath`，所以总时间复杂度 $T(|V|, |E|) = O(|V|^2)$。

四、程序源代码

Dijkstra 单源最短路径: **dijkstra.cpp**:

```
#include <iostream>
#include <limits>
#include <utility>
#include <vector>

// Dijkstra 单源最短路径
// 参数:
//       G[][]: 图的邻接矩阵（G_{i,j}）表示边 i -> j 的权值，+INF 表示不可达
//       s: 源节点下标
// 返回值:
```

```cpp
//      dist[]: 表示从源s到各点的最短距离
//      from[]: 表示从源s到各点的最短路径中，前一个节点
auto Dijkstra(const std::vector<std::vector<double>> & G, size_t s)
{
    // 用来记录是否已经找到最短路径
    std::vector<bool> found;
    // 路径长度
    std::vector<double> dist;
    // 用来回溯路径
    std::vector<size_t> from;

    // 初始化
    for(size_t i = 0; i < G.size(); i++)
    {
        found.push_back(false);
        dist.push_back(G[s][i]);
        // 如果从源不可达
        if(G[s][i] == std::numeric_limits<double>::infinity())
        {
            // 前一个认为不存在
            from.push_back(G.size());
        }
        else
        {
            // 前一个是源
            from.push_back(s);
        }
    }

    // 源初始化
    dist[s] = 0;
    found[s] = true;

    // n 个节点
    for(size_t i = 0; i < G.size(); i++)
    {
        double minDist = std::numeric_limits<double>::infinity();
```

```
48          size_t u = s;
49          // 找dist最小的
50          for(size_t j = 0; j < G.size(); j++)
51          {
52              if( !found[j] && dist[j] < minDist)
53              {
54                  u = j;
55                  minDist = dist[j];
56              }
57          }
58          // 加入集合
59          found[u] = true;
60          // 更新距离
61          for(size_t j = 0; j < G.size(); j++)
62          {
63              if(!found[j] && (G[u][j] < std::numeric_limits<double>::infinity()))
64              {
65                  double newDist = dist[u] + G[u][j];
66                  if(newDist < dist[j])
67                  {
68                      dist[j] = newDist;
69                      from[j] = u;
70                  }
71              }
72          }
73      }
74      return std::make_pair(dist, from);
75  }
76
77  // 利用 from 递归打印从 src 到 dest 的路径
78  // 参数:
79  //       from: Dijkstra 算法得出的from数组
80  //       src: 源
81  //       dest: 终点
82  void printPath(const std::vector<size_t> & from, size_t src, size_t dest)
83  {
84      // 到达源
```

```cpp
    if(dest == src)
    {
        std::cout << src + 1;
        return;
    }
    // 递归打印之前的路径
    printPath(from, src, from[dest]);
    // 打印本节点
    std::cout << " -> " << dest + 1;
}

int main(int argc, char *argv[])
{

#define INF std::numeric_limits<double>::infinity()

    // 课件上的例子
    std::vector<std::vector<double>> G = {
        {0, 10, INF, 30, 100},
        {INF, 0, 50, INF, INF},
        {INF, INF, 0, INF, 10},
        {INF, INF, 20, 0, 60},
        {INF, INF, INF, INF, 0}
    };
    auto res = Dijkstra(G, 0);
    // 打印路长
    std::cout << "(以1为源)" << std::endl;
    std::cout << "最短路径长度:" << std::endl;
    for(size_t i = 0; i < G.size(); i++)
    {
        std::cout << res.first[i] << " ";
    }
    std::cout << std::endl << std::endl;
    // 打印路径
    std::cout << "各最短路径:" << std::endl;
    for(size_t i = 0; i < G.size(); i++)
    {
```

```
122             printPath(res.second, 0, i);
123             std::cout << std::endl;
124         }
125         return 0;
126 }
```

五、运行结果截图

编译运行 dijkstra.cpp，程序以课件中的例子做测试：

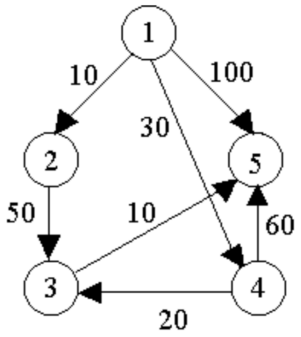

图 3: 测试例子

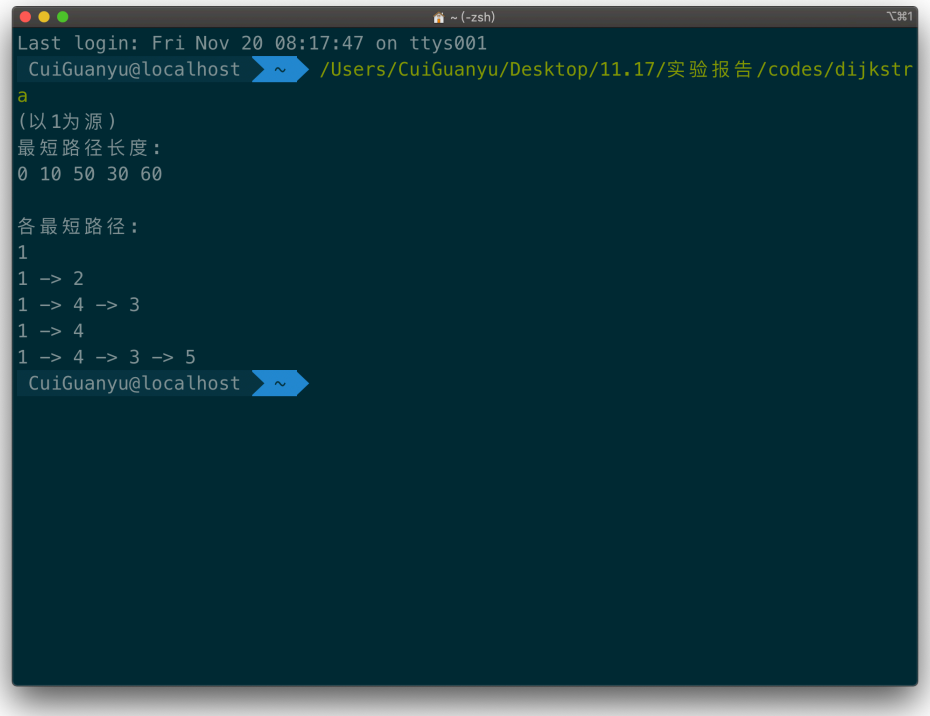

图 4: Dijkstra 测试

观察可见程序运行正确。