HUNAN UNIVERSITY

数据结构与算法分析

实验报告

学生姓名/学号　梅炳寅　202108010206

专业班级　计科 2102

指导老师　夏艳

2022 年 5 月 17 日

目录

1.问题分析.....3

1.1 处理的对象（数据）.....3

1.2 实现的功能.....3

1.3 处理后的结果如何显示.....3

1.4 请用题目中样例，详细给出样例求解过程。.....3

2.数据结构和算法设计.....4

2.1 抽象数据类型设计.....4

2.2 物理数据对象设计（不用给出基本操作的实现）.....5

2.3 算法思想的设计.....8

2.4 关键功能的算法步骤（不能用源码）.....8

3. 算法性能分析.....8

3.1 时间复杂度.....8

3.2 空间复杂度.....9

4.不足与反思.....

1.问题分析

在 n 个人中，某些人的银行账号之间可以互相转账。这些人之间转账的手续费各不相同。给定这些人之间转账时需要从转账金额里扣除百分之几的手续费，请问 A 最少需要多少钱使得转账后 B 收到 100 元。

1.1 处理的对象（数据）

第一行两个正整数 n,m，分别表示**总人数**和可以互相转账的人的**对数**。

以下 m 行每行输入三个正整数 **x,y,z**，表示**标号为 x 的人**和**标号为 y 的人**之间互相转账需要扣除 **z%的手续费**(z<100)。

最后一行**两个正整数 A,B**。数据保证 A 与 B 之间可以直接或间接地转账。

注意： 2<=N<=20，1<=M<=20，1<=Q<=100

1.2 实现的功能

A 最少需要多少钱使得转账后 B 收到 100 元。

本质是求 A 与 B 之间的重定义（根据题目要求做出重载的）距离。

1.3 处理后的结果如何显示

一个浮点数，输出 A 使得 B 到账 100 元最少需要的总费用。

精确到小数点后 8 位。

1.4 请用题目中样例，详细给出样例求解过程。

【输入样例】

3 3

1 2 1

2 3 2

1 3 4

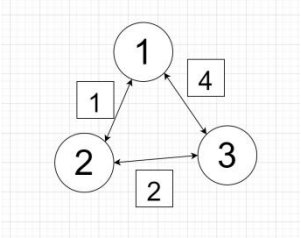
1 3

【输出样例】

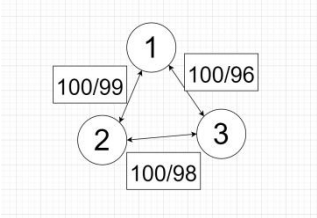
103.07153164

【分析】

1、根据题意，构建一张图形。（存储入图结构）



2、从起始点开始，按照重定义的距离进行类似于 Dijkstra 算法求到各点的最短路径（重定义后边权值如下，后面算法分析中会给出重定义的规则）



3、第一步，标记 1，从起始点开始，发现 1->3 的距离为 100/96，约为 1.04；而 1->2 的距离为（100/99），约为 1.01，显然后者较小，故选择 2 为下一节点，此时重定义的距离数组为{1，100/99，100/96}

4、第二步，标记 2，从当前节点 2 开始，发现 2->3 的距离为 100/98，那么此时 1->2->3 的距离为（100/99）*（100/98）约为 1.03；而 1->3 的距离为 100/96，为 1.04。需要更新距离。故选择 2 为下一节点，此时重定义的距离数组为{1，100/99，10000/(98*99)},并选择 3 为下一节点。

5、第三步，标记 3，发现此时已经全部被标记，返回 D[end]即 10000/(98*99)。

2.数据结构和算法设计

2.1 抽象数据类型设计

（1）图 ADT ——Graph.h

```
class Graph {
private:
   void operator =(const Graph&) {}          // Protect assignment
   Graph(const Graph&) {}                    // Protect copy constructor
public:
   Graph() {}                    // Default constructor
   virtual ~Graph() {} // Base destructor
  //初始化 n 节点图
   virtual void Init(int n) =0;
  //  返回图的节点数与边数
   virtual int n() =0;
   virtual int e() =0;
  //  返回 v 的首个邻节点
   virtual int first(int v) =0;
  //  返回 v 在 w 后的下一个邻节点
   virtual int next(int v, int w) =0;
  //  为边设置权值
  // i, j: The vertices// wgt: Edge weight
   virtual void setEdge(int v1, int v2, int wght) =0;
  //删除边// i, j: The vertices
   virtual void delEdge(int v1, int v2) =0;
  //  判定该边是否存在邻边// i, j: The vertices
    // Return: true if edge i,j has non-zero weight
   virtual bool isEdge(int i, int j) =0;
```

```
  // 返回边的权值// i, j: The vertices
    // Return: The weight of edge i,j, or zero
    virtual int weight(int v1, int v2) =0;
  // 获取并设置边的权值    // v: The vertex
    // val: The value to set
  virtual int getMark(int v) =0;
  virtual void setMark(int v, int val) =0;
};
```

（2）图的底层存储 ADT（使用链表结构）list.h

```
template <typename E> class List { // List ADT
private:
    void operator =(const List&) {}          // Protect assignment
    List(const List&) {}                     // Protect copy constructor
public:
    List() {}                  // 默认构造函数
    virtual ~List() {} // 基本的析构函数
// 从列表中清除内容,让它空着
    virtual void clear() = 0;
// 在当前位置插入一个元素// item: 要插入的元素
    virtual void insert(const E& item) = 0;
// 在列表的最后添加一个元素 // item: 要添加的元素
    virtual void append(const E& item) = 0;
// 删除和返回当前元素 // Return: 要删除的元素
    virtual E remove() = 0;
// 将当前位置设置为列表的开始
    virtual void moveToStart() = 0;
// 将当前位置设置为列表的末尾
    virtual void moveToEnd() = 0;
// 将当前位置左移一步，如果当前位置在首位就不变
    virtual void prev() = 0;
// 将当前位置右移一步，如果当前位置在末尾就不变
    virtual void next() = 0;
// 返回列表当前元素个数
    virtual int length() const = 0;
// 返回当前位置
    virtual int currPos() const = 0;
// 设置当前位置 // pos: 要设置的当前位置
    virtual void moveToPos(int pos) = 0;
// Return: 当前位置的元素
    virtual const E& getValue() const = 0;
};
```

2.2 物理数据对象设计（不用给出基本操作的实现）

（1）图的算法实现 Graph_test.h

```
class option
{
private:
	Graph *G;
public:
	option(Graph *g);
	long double Dijkstra1(long double* D, int start, int end);
	//使用类 Dijkstra 算法求最短重定义距离
	int minVertex(long double* D);	// 找到最短代价点
};
```

（2）图的存储操作底层实现 grlist.h

```
class Edge
{
	int vert,wt;
public:
	Edge();
	Edge(int v, int w);
	int vertex();
	int weight();
};
//以上主要是处理将内部量向外部传递，更好实现封装性

class Graphl : public Graph
{
private:
	List<Edge>** vertex;		// List headers
	int numVertex, numEdge;		// Number of vertices, edges
	int *mark;			// Pointer to mark array
public:
	Graphl(int numVert)//构造函数
	~Graphl()		// 析构函数
	void Init(int n)	//返回边数与点数
	int n()			//返回点数
	int e()			//返回边数
	int first(int v)	// 返回 v 的首个邻节点
	int next(int v, int w)	// 返回 v 在 w 后的下一个邻节点
	void delEdge(int i, int j)	// 删除边(i, j)
	bool isEdge(int i, int j)	// 判定(i, j)是否有边
	int weight(int i, int j)	// 返回(i, j)的边权值
	int getMark(int v)
	void setMark(int v, int val)
};
```

（3）链表的底层实现 llist.h

```
template <typename E> class LList: public List<E> {
private:
   Link<E>* head;            // 指向链表头结点
   Link<E>* tail;            // 指向链表最后一个结点
   Link<E>* curr;            // 指向当前元素
   int cnt;                  // 当前列表大小
   void init() {             // 初始化
      curr = tail = head = new Link<E>;
      cnt = 0;
   }
   void removeall() {      // Return link nodes to free store
      while(head != NULL) {
         curr = head;
         head = head->next;
         delete curr;
      }
   }

public:
   LList(int size=100)               // 构造函数
   ~LList() ;                        // 析构函数
   void print() const;               // 打印列表内容
   void clear()                      // 清空列表
   // 在当前位置插入“it”
   void insert(const E& it);
   void append(const E& it);
   // 删除并返回当前元素
   E remove();
   void moveToStart() ;// 将 curr 设置在列表头部
   void moveToEnd() ;   // 将 curr 设置在列表尾部
   void prev();
   // 将 curr 指针往前（左）移一步；如果已经指向头部了就不需要改变
   void next();
   // 将 curr 指针往后（右）移一步；如果已经指向尾部了就不需要改变
   int length() const; // 返回当前列表大小
   int currPos() const; // 返回当前元素的位置
   void moveToPos(int pos);   // 向下移动到列表“pos”位置
   const E& getValue() const    //返回当前元素
};
```

（4）链接文件 link.h

```
template <typename E> class Link {
public:
```

```
    E element;          // 结点值
    Link *next;             // 结点指针：在链表中指向下一结点
    // 构造函数
    Link(const E& elemval, Link* nextval =NULL);
        Link(Link* nextval =NULL);
};
```

2.3 算法思想的设计

1、第一步，标记起始点，从起始点开始，比较与起始点相连的点的转化率，选择最短边权相连的为下一节点。更新所有与初始节点相邻点的转化率。

2、第二步，标记该节点，从当前节点开始，比较与当前节点相连的点的距离，选择最短边权相连的为下一节点。更新所有与当前节点相邻点的转化率，方法如下：比较所有相邻点与初始点的转化率（即 D[w]）和初始点经自己再到相邻节点的转化率（即 D[v]*G->weight(v,w)）是否符合三角形法则,若不符合，则更新。

3、第三步，重复 2 步骤 n 次，发现已经全部被标记时，返回 D[end]。

2.4 关键功能的算法步骤（不能用源码）

1、高准确度存取节点与边的位置信息

将节点信息准确、高效地存储在图中是十分重要的，也是完成整个问题的基础，本题我将采取链表的结构来存储。

2、重定义距离（转化率） and 类 Dijkstra 算法

本题最重要的解题点是理解重定义距离（转化率）。

在普通 Dijkstra 算法中，两点的距离表示从起始点到重点的代价。而本题的两点之间的代价有所不同。我作如下分析。

假设两人 A、B，假定手续费为 z%，假设 A 给 B 了 m 元，那么 B 收到 m(1-z%)元，要使 B 收到 100 元，解方程 m(1-z%)=100 有 m=100/(1-z%)，其中 1/(1-z%)称为 A 到 B 的重定义距离，我们也可以形象地理解为“转化率”。

在多人模式中，假定由 A 经过 BCD 传至 E，有初始值 m=100/【(1-z(A,B)%)*(1-z(B,C)%)*(1-z(C,D)%)*(1-z(D,E)%)】。

如果说在原 Dijkstra 算法中，距离是经过点的边权值相加，那么在本题中，类 Dijkstra 算法就是经过点的边权值相乘。类 Dijkstra 本质是求解一条最优转化路径，使得总转化率最低。

理解了以上两个概念，就基本能够理解本题。

3、D[start]的含义

起始点 start 所对应的 D 值为 1，特别注意不是 0。因为自己转给自己不需要手续费。

3.算法性能分析

3.1 时间复杂度

该算法的时间复杂度是 O（n^2），下面是具体分析。由于代码较长，我们主要采取功能的方法来分析。

1、图的存储，非嵌套调用：

建立图结构、存储图结构为 O（n^2），一些取值的结构为 O（n）

2、类 Dijkstra 算法求最优转化效率

使用 Dijkstra 算法，时间复杂度为 O（n^2）

综上所述，时间复杂度为

$$T = O(n^2)$$

3.2 空间复杂度

由于使用链表结构图，并构造辅助数组 D[n],所以大致在 O（n）。

4.不足与反思

在算法上本题应该已经是比较优化的了。但仍然可以更加优化，包括用堆优化 Dijkstra 算法，可使时间复杂度降到 O((m+n)log n)。