2023-2024 春季学期 数据结构实验报告

小组成员：王泽黎 姚永舟 张家玮

实验项目：

1.实习 5.8 全国交通咨询模拟

2.实习 6.7 多关键字排序

人员分工：实习 1 由姚永舟和张家玮完成，实习 2 由王泽黎完成

目录

实验项目 1：全国交通咨询模拟

1. 基本要求：

1.1 提供对城市信息进行编辑（如：添加或删除）的功能。

1.2 城市之间有两种交通工具：火车和飞机。提供对列车时刻表和飞机航班进行编辑（增设或删除）的功能。

1.3 提供两种最优决策：最快到达或最省钱到达。全程只考虑一种交通工具。

1.4 旅途中耗费的总时间应该包括中转站的等候时间。

1.5 咨询以用户和计算机的对话方式进行。由用户输入起始站、终点站、最优决策原则和交通工具，输出信息：最快需要多长时间才能到达或者最少需要多少旅费才能到达，并详细说明依次于何时乘坐哪一趟列车或哪一次班机到何地。

2. 数据结构概要

2.1 记录交通信息（以下两个结构体均用来记录航班/列车信息，包括航班/车次号、起点城市、终点城市、出发时间、到达时间、旅费价格）

```
struct arc
{
    char id[MAX_TRANSPORT_ID_LEN];
    char StartCity[MAX_CITY_NAME_LEN];
    char EndCity[MAX_CITY_NAME_LEN];
    int BeginTime[2];
    int ArriveTime[2];
    float price;
}a[MAX_ARC_SIZE]; //临时存放飞机或列车信息，用于初始化交通系统

typedef struct
{
    char number[MAX_TRANSPORT_ID_LEN];
    float expenditure;
    int begintime[2];
    int arrivetime[2];
}Vehide; //记录每一个航班或车次的信息
```

2.2 有关图的数据结构（每条弧上从出发点到结束点最多几种交通方式、弧节点、邻接表、图、满足用户要求的路径）

```
typedef struct
{
    Vehide stata[MAX_ROUTE_NUM];
    int last;
}infolist; //记录每一个弧的信息

typedef struct ArcNode
{
    int adjvex;
    struct ArcNode *nextarc;
    infolist info;
}ArcNode; //弧结点

typedef struct VNode
{
    char cityname[MAX_CITY_NAME_LEN];
    ArcNode *planefirstarc, *trainfirstarc;
}VNode,AdjList[MAX_VERTEX_NUM]; //邻接表

typedef struct
{
    AdjList vertices;
    int vexnum, planearcnum, trainarcnum;
}ALGraph;  //存放交通图

typedef struct Node
{
    int adjvex;
    int route;
    struct Node *next;
}Node; //记录满足用户要求的路径
```

2.3 广度优先搜索使用的队列（结点和头尾指针）

```
typedef struct QNode
{
    int adjvex;
    struct QNode *next;
}QNode; //链队列结点

typedef struct
{
    QNode *front;
    QNode *rear;
}LinkQueue; //链队列的头尾指针
```

3.详细设计

3.1 采用Dijkstra算法求解最短路径和最少花费

3.1.1 Dijkstra算法求解最短花费时间路线

（1）初始化：将所有节点标记为未访问，设置起点到自身的最短时间为0，到其他所有点的最短时间为无穷大。

（2）选择节点：从未访问的节点中选择一个距离起点最短时间的节点作为当前节点。

（3）更新邻居：对当前节点的每一个邻居，计算通过当前节点到达它的时间，如果这个时间比已知的最短时间短，更新这个邻居的最短时间。

（4）标记访问：将当前节点标记为已访问。

（5）重复：如果有未访问的节点，回到步骤2。

（6）完成：当所有节点都被访问后，算法结束。从终点节点回溯到起点，就可以找到最短花费时间的路线。

3.1.2 Dijkstra算法求解最少旅费花费路线

这个过程与求解最短花费时间的路线类似，不同之处在于评估标准不是时间，而是旅费。

3.2 采用BFS求解最少中转次数

使用广度优先搜索（BFS）算法求解中转次数最少的路线的过程如下：

（1）初始化：创建一个队列用于存储待访问的节点，将起点加入

队列。为每个节点设置一个中转次数计数器，起点的中转次数为0，其他所有点的中转次数为无穷大。

（2）遍历队列：只要队列不为空，就从队列中取出一个节点作为当前节点。

（3）访问邻居：对于当前节点的每一个邻居，检查是否已经访问过（即中转次数是否为无穷大）。如果邻居节点未被访问过，将其加入队列，并更新该邻居节点的中转次数为当前节点的中转次数加1。

（4）重复步骤2和3：继续从队列中取出节点并访问其未被访问过的邻居，直到队列为空。

（5）找到目标节点：当目标节点从队列中被取出来访问时，其中转次数即为从起点到达该节点的最少中转次数。

（6）完成：算法结束时，所有节点的中转次数都已经计算。可以通过回溯从终点到起点的路径来找到中转次数最少的路线。

BFS算法按层次遍历图，确保在访问较远的节点之前先访问所有较近的节点。一旦找到目标节点，就可以确信找到的路径是中转次数最少的路径。

3.3 核心代码展示（由于代码太长，故展示伪代码，具体代码可在源码中查看）

下面代码是求最少旅费的Dijkstra算法：

```
函数 ExpenditureDispose(起始城市, 目标城市, 图):
    初始化路径和费用信息
    对每个城市:
        初始化最小费用为无穷大，除了起始城市为0
        找到最小费用未处理的城市
        如果该城市是目标城市:
            输出路径和费用
            退出
        否则:
            更新邻接城市的最小费用和路径
    清理资源

函数 MinExpenditure(直达路径信息):
    初始化最小费用为第一个路径的费用
    遍历所有路径:
        如果找到更小的费用:
            更新最小费用和路径索引
```

下面代码是求最少中转次数使用的队列：

```
function TransferDispose(k, arcs, G, v0, v1):
    初始化 visited 为 G.vexnum 长度的数组，所有值为 0
    初始化队列 Q
    标记 v0 为已访问
    将 v0 加入队列 Q
    while Q 不为空:
        从 Q 中删除顶点 v
        选择 t 为火车或飞机的邻接表，基于 k 的值
        while t 不为空:
            w = t 的邻接顶点
            if w 未被访问:
                标记 w 为已访问
                记录从 v 到 w 的路径
                if w 是 v1:
                    打印路径和最少中转次数
                    返回
                将 w 加入队列 Q
            t = t 的下一个邻接点
    打印不存在路径的信息
```

4. 测试结果与调试分析

4.1 编辑城市信息、列车车次信息和航班信息

4.1.1 初始信息展示(初始信息以文档方式读入，后续添加删除以键盘输入方式读入)

图 1：初始信息展示

4.1.2 添加操作展示

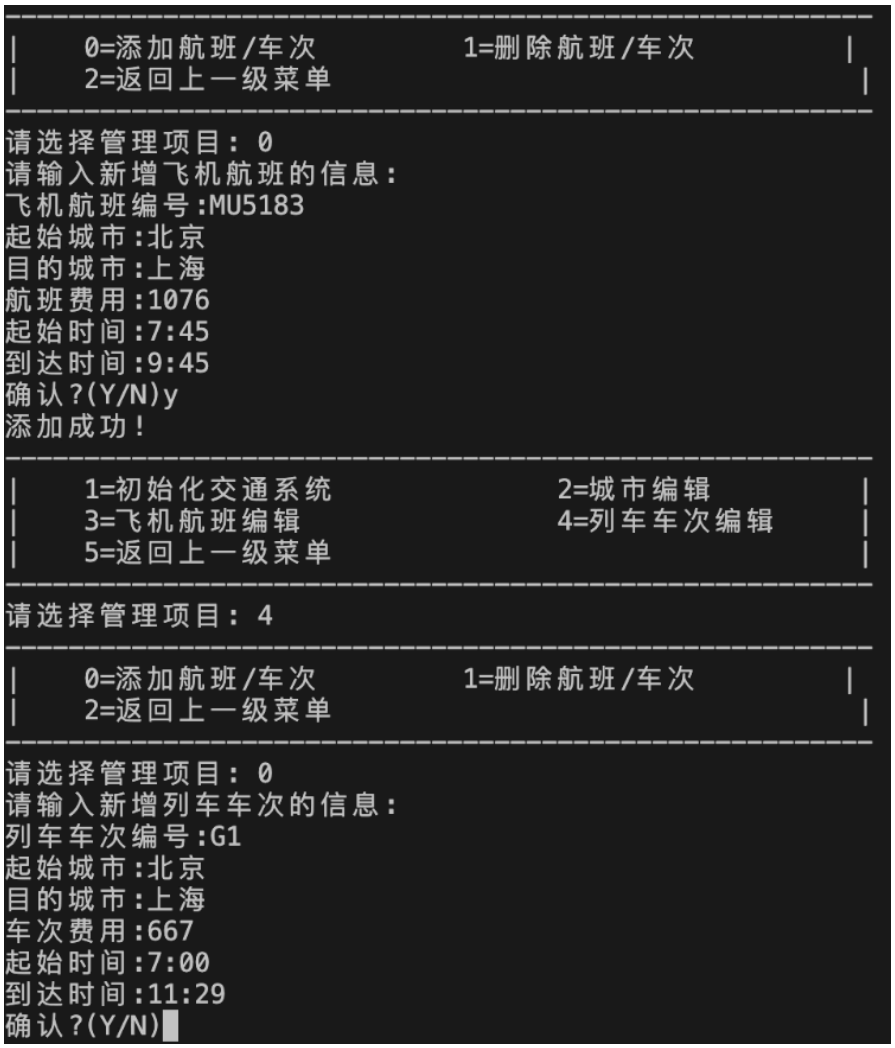


图 2：添加车次和航班信息

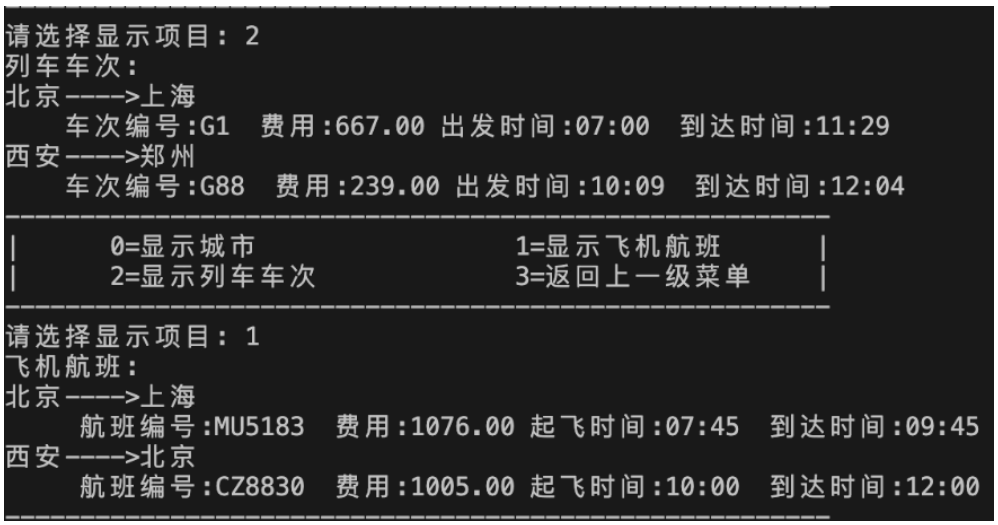


图 3：添加车次和航班后的结果

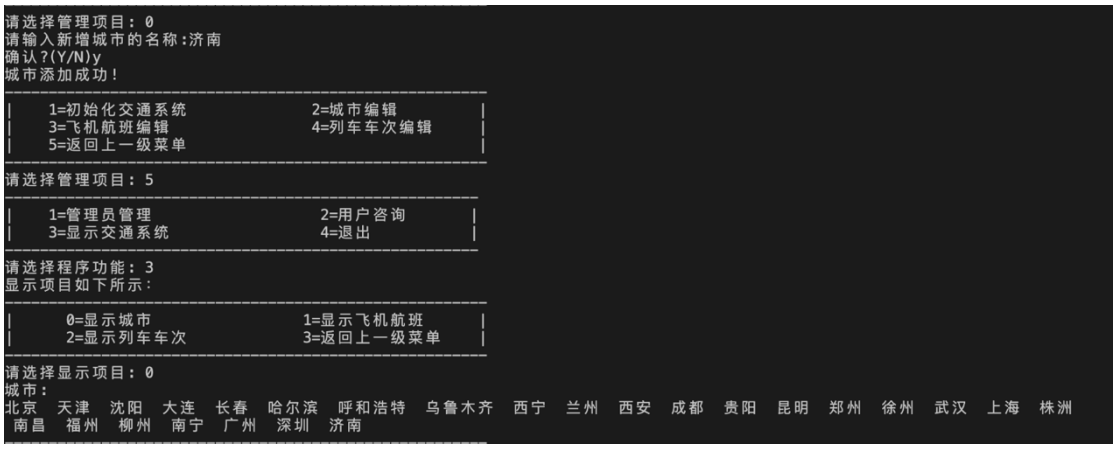


图 4：添加城市及添加后的结果

4.1.3 删除操作展示

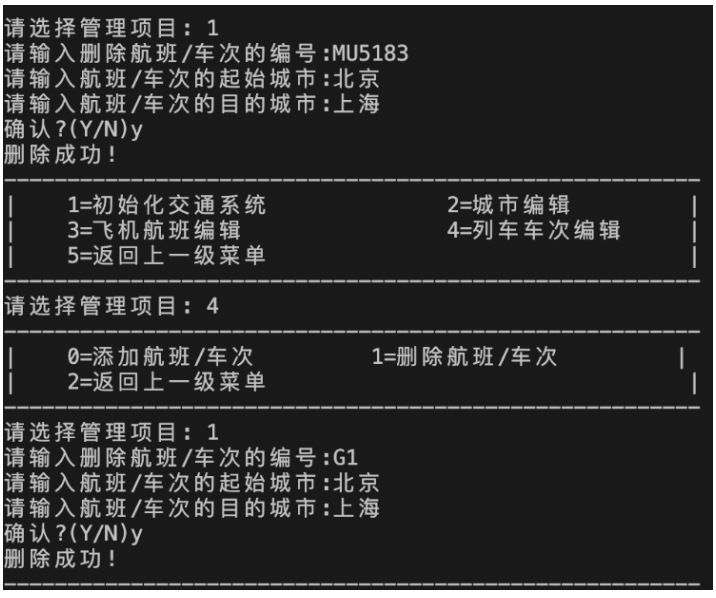


图 5：删除列车车次与航班操作

```
显示项目如下所示：
------------------------------------------------------------
|      0=显示城市                    1=显示飞机航班       |
|      2=显示列车车次                3=返回上一级菜单     |
------------------------------------------------------------
请选择显示项目： 1
飞机航班：
西安----->北京
     航班编号:CZ8830  费用:1005.00 起飞时间:10:00  到达时间:12:00
------------------------------------------------------------
|      0=显示城市                    1=显示飞机航班       |
|      2=显示列车车次                3=返回上一级菜单     |
------------------------------------------------------------
请选择显示项目： 2
列车车次：
西安----->郑州
    车次编号:G88  费用:239.00 出发时间:10:09  到达时间:12:04
------------------------------------------------------------
```

图 6：删除列车车次与航班后的结果

4.2 最少旅费花费咨询展示

三种咨询策略的展示，初始化城市、车次和航班信息均如下图所示

```
|      0=显示城市                    1=显示飞机航班       |
|      2=显示列车车次                3=返回上一级菜单     |
------------------------------------------------------------
请选择显示项目： 0
城市：
北京  天津  呼和浩特  徐州  郑州  西安  兰州
------------------------------------------------------------
|      0=显示城市                    1=显示飞机航班       |
|      2=显示列车车次                3=返回上一级菜单     |
------------------------------------------------------------
请选择显示项目： 1
飞机航班：
呼和浩特----->北京
     航班编号:CA1112  费用:520.00 起飞时间:17:30  到达时间:18:55
郑州----->北京
     航班编号:CA1916  费用:765.00 起飞时间:09:10  到达时间:10:40
     航班编号:CA1914  费用:820.00 起飞时间:15:55  到达时间:17:35
西安----->天津
     航班编号:MU6923  费用:352.00 起飞时间:08:55  到达时间:11:00
西安----->北京
     航班编号:CZ8830  费用:1005.00 起飞时间:10:00  到达时间:12:00
     航班编号:CA1206  费用:1100.00 起飞时间:12:55  到达时间:10:40
兰州----->呼和浩特
     航班编号:MU2347  费用:1010.00 起飞时间:01:55  到达时间:16:50
兰州----->西安
     航班编号:MU6879  费用:440.00 起飞时间:08:35  到达时间:09:50
兰州----->北京
     航班编号:HU7198  费用:900.00 起飞时间:10:25  到达时间:12:25
兰州----->郑州
     航班编号:UQ3562  费用:287.00 起飞时间:12:55  到达时间:14:50
```

图 7：初始化城市与航班信息

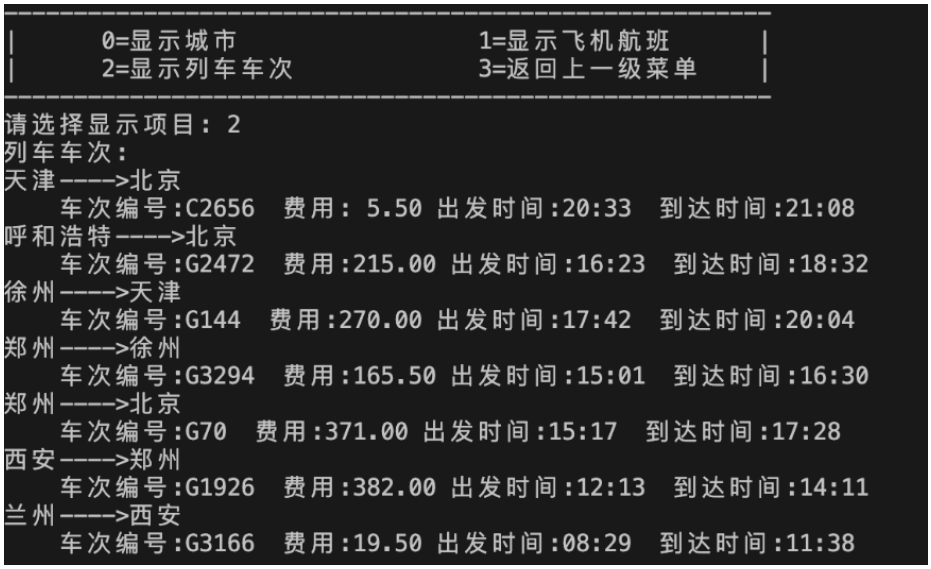


图 8：初始化列车车次信息

最少旅费花费如下：

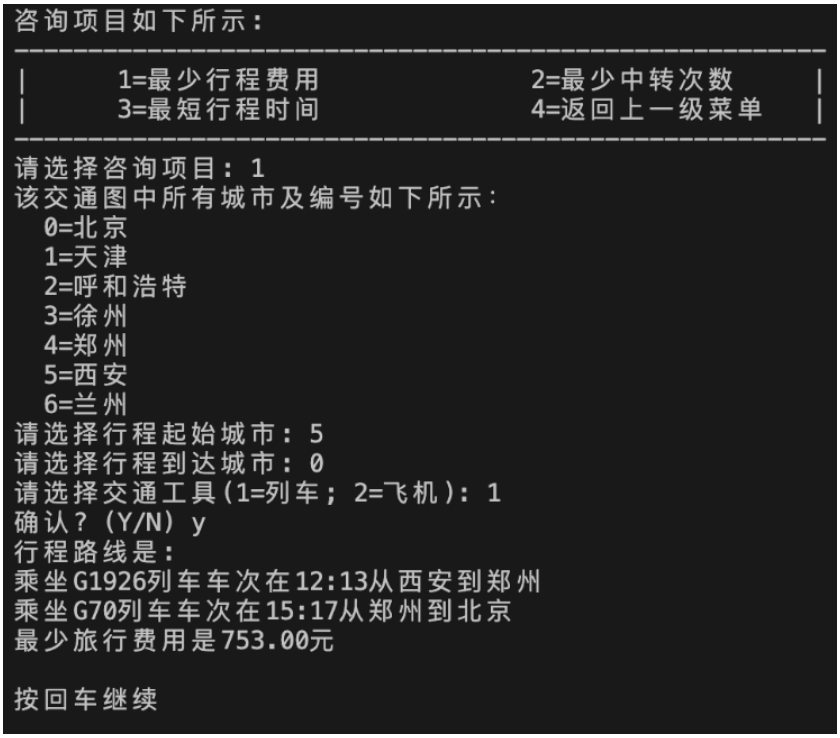


图 9：最少旅行费用咨询

4.3 最短时间花费咨询演示

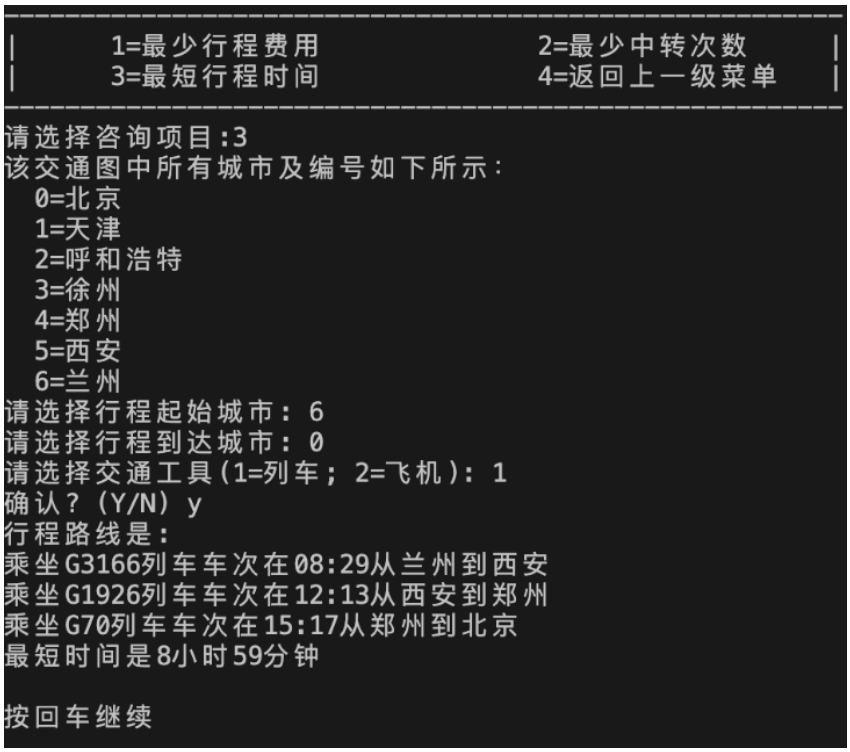


图 10：最短旅行时间咨询

4.4 最少中转次数咨询演示

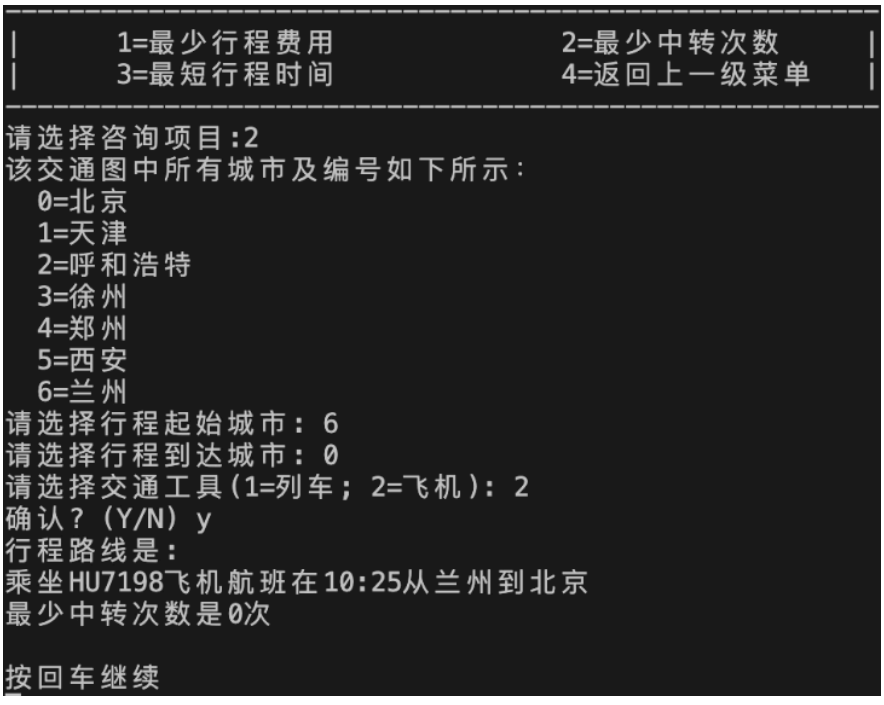


图 11：最少中转次数咨询

实验项目 2：多关键字排序

1. 基本要求：

1.1 按用户给定的进行排序的关键字优先关系进行多关键字排序，并输出排序结果

1.2 待排序的记录数与关键词数由用户给出，其中记录数不超过 1000，关键词数不超过 5，各个关键字的范围均为 0 至 100

1. 使用 LSD 法和 MSD 法进行多关键字排序，每种排序方法分别采用两种策略：其一是利用稳定的内部排序法，其二是利用“分配”和“收集”的方法

1.4 分析比较各种方法策略的排序时间

2. 数据结构概要

“分配”和“收集”使用的抽象数据类型定义

```
// 桶元素结点
typedef struct Node *PtrToNode;
struct Node
{
    int data[MAXKEY]; // 关键字内容
    PtrToNode next; // 桶中下一个记录的位置
};

// 桶头结点
typedef struct HeadNode
{
    PtrToNode head; // 桶头结点
    PtrToNode tail; // 桶尾结点
} HeadNode;
```

3. 详细设计

3.1 LSD 和 MSD 策略的实现

LSD 排序根据优先级从低到高依次调用对应关键字的排序函数，全部调用完即可。

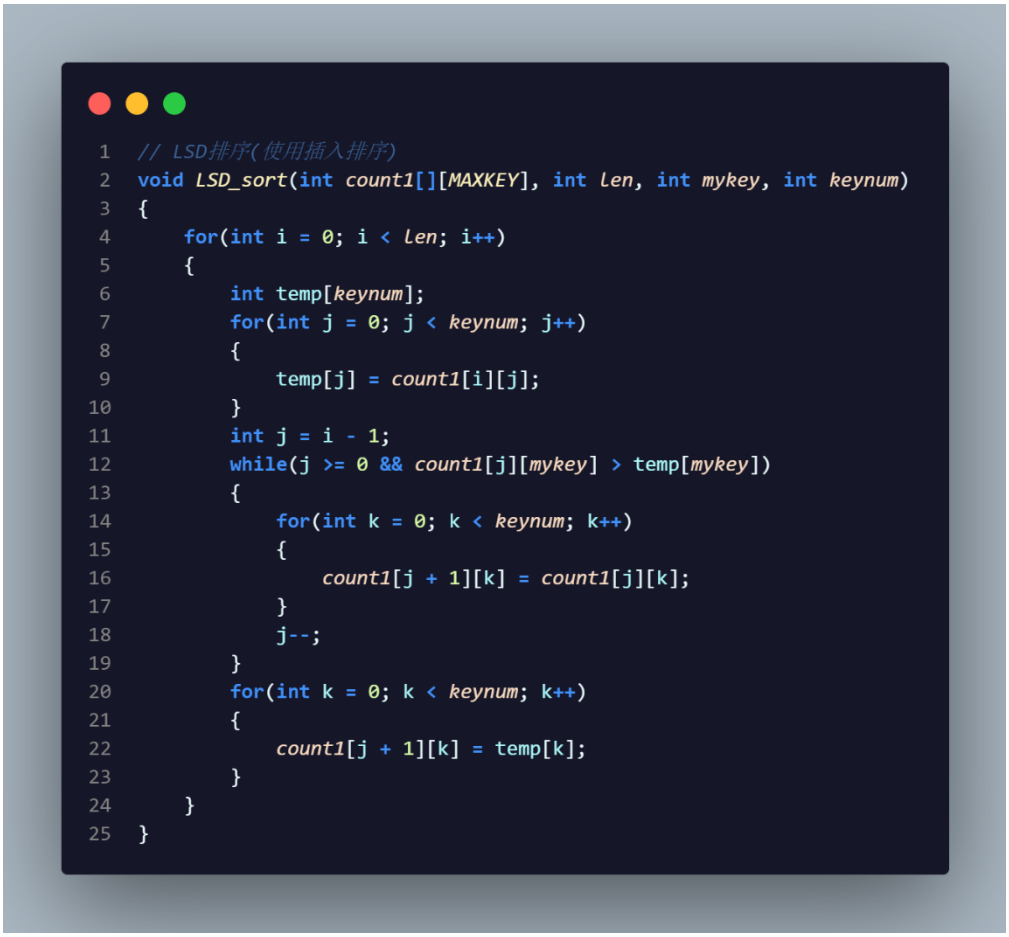
MSD 排序优先依据高关键字进行排序，在高一级值相等时，递归在对应范围内进行低一级关键字进行排序。

3.2 稳定的内部排序法和“分配”“收集”法

稳定的内部排序法选择使用插入排序，“分配”“收集”法即使用基数排序（桶排序）

3.3 核心代码段展示

LSD 法插入排序



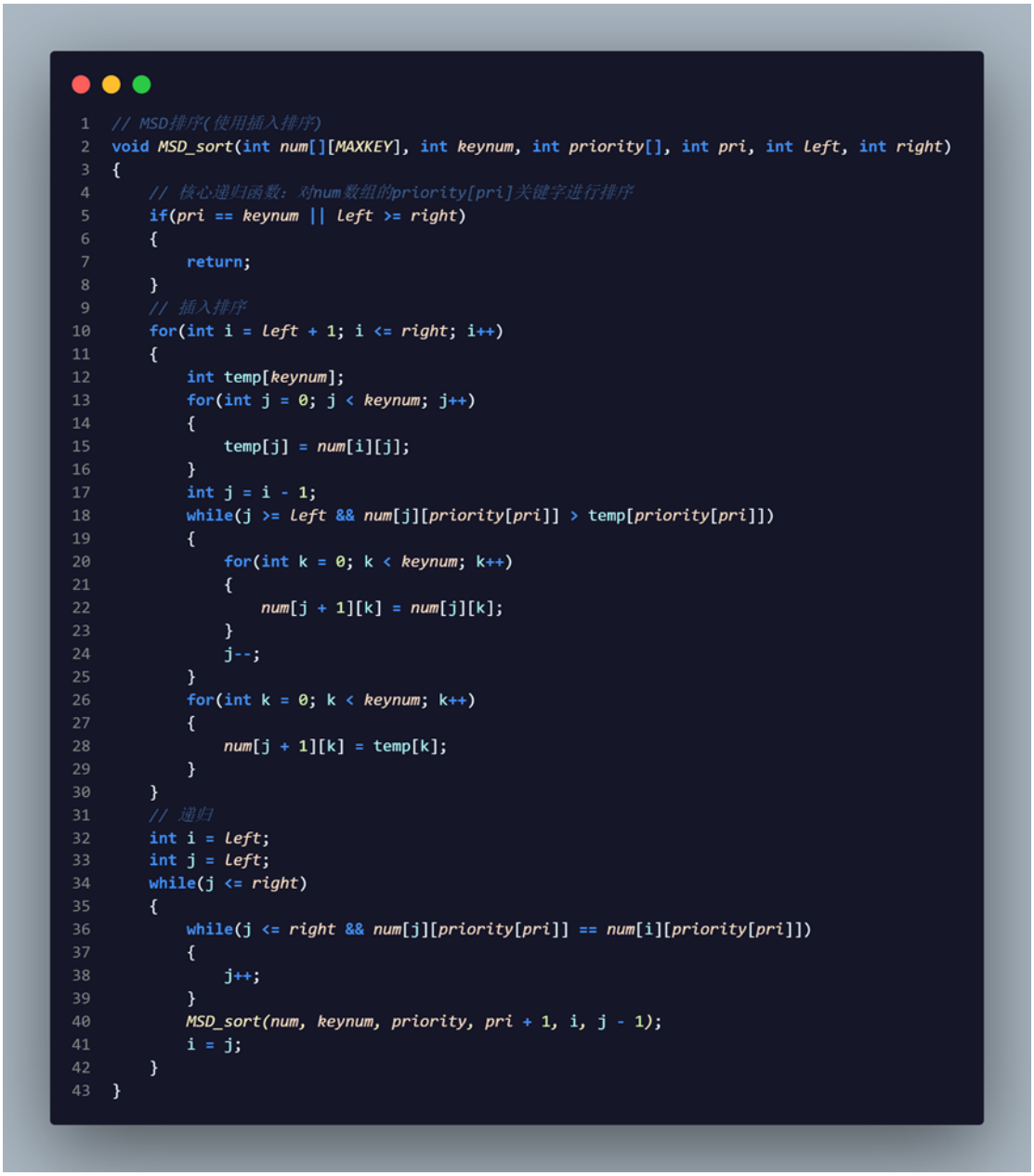
```
// LSD排序(使用插入排序)
void LSD_sort(int count1[][MAXKEY], int Len, int mykey, int keynum)
{
    for(int i = 0; i < Len; i++)
    {
        int temp[keynum];
        for(int j = 0; j < keynum; j++)
        {
            temp[j] = count1[i][j];
        }
        int j = i - 1;
        while(j >= 0 && count1[j][mykey] > temp[mykey])
        {
            for(int k = 0; k < keynum; k++)
            {
                count1[j + 1][k] = count1[j][k];
            }
            j--;
        }
        for(int k = 0; k < keynum; k++)
        {
            count1[j + 1][k] = temp[k];
        }
    }
}
```

LSD 法基数排序

```c
// LSD排序（使用“分配”和“收集”方法）
void LSD_Radix_Sort(int num[][MAXKEY], int size, int key, int priority[])
{
    int number = 0;
    Bucket bucket;
    PtrToNode temp, p, list = NULL;
    // 初始化桶
    for (int i = 0; i < BUCKETNUM; i++)
    {
        bucket[i].head = bucket[i].tail = NULL;
    }
    // 将待排序记录逆序存入初始链表list
    for(int i = 0; i < size; i++)
    {
        temp = (PtrToNode)malloc(sizeof(struct Node));
        for(int j = 0; j < key; j++)
        {
            temp->data[j] = num[i][j];
        }
        temp->next = list;
        list = temp;
    }
    // LSD排序
    for(int i = key - 1; i >= 0; i--)
    {
        // 分配
        p = list;
        while(p)
        {
            // 获取关键字
            number = p->data[priority[i]];
            // 将结点从list中摘除
            temp = p;
            p = p->next;
            // 将结点插入到桶中
            temp->next = NULL;
            if(bucket[number].head == NULL)
            {
                bucket[number].head = bucket[number].tail = temp;
            }
            else
            {
                bucket[number].tail->next = temp;
                bucket[number].tail = temp;
            }
        }
        // 收集
        list = NULL;
        for(int j = BUCKETNUM - 1; j >= 0; j--)
        {
            // 将桶中结点逆序存入list
            if(bucket[j].head) // 桶不为空
            {
                // 将整个桶插入到list
                if(list == NULL)
                {
                    list = bucket[j].head;
                }
                else
                {
                    bucket[j].tail->next = list;
                    list = bucket[j].head;
                }
                // 清空桶
                bucket[j].head = bucket[j].tail = NULL;
            }
        }
    }

    // 将排序后的记录存入num，并释放空间
    for(int i = 0; i < size; i++)
    {
        temp = list;
        for(int j = 0; j < key; j++)
        {
            num[i][j] = temp->data[j];
        }
        list = list->next;
        free(temp);
    }
}
```

MSD 法插入排序



```c
// MSD排序(使用插入排序)
void MSD_sort(int num[][MAXKEY], int keynum, int priority[], int pri, int left, int right)
{
    // 核心递归函数：对num数组的priority[pri]关键字进行排序
    if(pri == keynum || left >= right)
    {
        return;
    }
    // 插入排序
    for(int i = left + 1; i <= right; i++)
    {
        int temp[keynum];
        for(int j = 0; j < keynum; j++)
        {
            temp[j] = num[i][j];
        }
        int j = i - 1;
        while(j >= left && num[j][priority[pri]] > temp[priority[pri]])
        {
            for(int k = 0; k < keynum; k++)
            {
                num[j + 1][k] = num[j][k];
            }
            j--;
        }
        for(int k = 0; k < keynum; k++)
        {
            num[j + 1][k] = temp[k];
        }
    }
    // 递归
    int i = left;
    int j = left;
    while(j <= right)
    {
        while(j <= right && num[j][priority[pri]] == num[i][priority[pri]])
        {
            j++;
        }
        MSD_sort(num, keynum, priority, pri + 1, i, j - 1);
        i = j;
    }
}
```

MSD 法基数排序

```c
// MSD排序（使用“分配”和“收集”方法）
void MSD_Radix_Sort(int num[][MAXKEY], int size, int key, int priority[], int pri, int left, int right)
{
    // 核心递归函数：对num数组的priority[pri]关键字进行排序
    int number = 0, i = 0, j = 0;
    Bucket bucket;
    PtrToNode temp, p, list = NULL;
    // 递归终止条件
    if(pri == key || left >= right)
    {
        return;
    }
    // 初始化桶
    for (int i = 0; i < BUCKETNUM; i++)
    {
        bucket[i].head = bucket[i].tail = NULL;
    }
    // 将待排序记录逆序存入初始链表list
    for(int m = left; m <= right; m++)
    {
        temp = (PtrToNode)malloc(sizeof(struct Node));
        for(int n = 0; n < key; n++)
        {
            temp->data[n] = num[m][n];
        }
        temp->next = list;
        list = temp;
    }
    // MSD排序
    // 分配
    p = list;
    while(p)
    {
        // 获取关键字
        number = p->data[priority[pri]];
        // 将结点从list中摘除
        temp = p;
        p = p->next;
        // 将结点插入到桶中
        if(bucket[number].head == NULL)
        {
            bucket[number].head = bucket[number].tail = temp;
            bucket[number].tail->next = NULL;
        }
        else
        {
            bucket[number].tail->next = temp;
            bucket[number].tail = temp;
            bucket[number].tail->next = NULL;
        }
    }
    // 收集
    // i和j记录当前处理的num数组的左右端下标
    i = j = left;
    for(int m = 0; m < BUCKETNUM; m++)
    {
        // 将非空的桶中的元素收集到num数组中，递归排序
        if(bucket[m].head)
        {
            p = bucket[m].head;
            while(p)
            {
                for(int n = 0; n < key; n++)
                {
                    num[j][n] = p->data[n];
                }
                j++;
                temp = p;
                p = p->next;
                free(temp);
            }
            // 递归排序
            MSD_Radix_Sort(num, size, key, priority, pri + 1, i, j - 1);
            // 为下一个桶对应num数组的左端下标赋值
            i = j;
        }
    }
}
```

4. 测试结果与调试分析

4.1 LSD 法插入排序（横轴为记录数，纵轴为关键字数，数据为耗时，单位为秒）

LSD插入	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1	0	0	0	0.019	0.016	0.023	0.038	0.038	0.045	0.063
2	0.001	0.016	0.023	0.027	0.049	0.061	0.065	0.106	0.123	0.16
3	0	0.012	0.037	0.059	0.081	0.128	0.161	0.206	0.253	0.318
4	0.001	0.036	0.054	0.092	0.131	0.203	0.276	0.35	0.436	0.526
5	0.019	0.037	0.079	0.132	0.218	0.322	0.433	0.555	0.686	0.858

4.2 LSD 法基数排序

LSD桶	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1	0	0	0	0	0	0	0	0	0	0.001
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0.012
4	0	0	0	0	0	0	0	0	0	0.016
5	0	0	0	0	0	0	0	0	0	0

4.3 MSD 法插入排序

MSD插入	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1	0	0	0	0.01	0.017	0.019	0.031	0.028	0.04	0.048
2	0	0.007	0.015	0.019	0.017	0.039	0.048	0.047	0.078	0.095
3	0	0	0.012	0.016	0.031	0.047	0.063	0.08	0.095	0.109
4	0	0.007	0.006	0.032	0.047	0.054	0.064	0.094	0.126	0.141
5	0.02	0.011	0.016	0.032	0.031	0.063	0.079	0.109	0.195	0.189

4.4 MSD 法基数排序

MSD桶	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0.016
4	0	0	0	0	0	0	0	0	0	0.016
5	0	0	0	0	0	0	0	0	0	0

综合上表数据可知，对于 LSD 法和 MSD 法，“分配”“收集”法的性能表现均明显优于稳定的内部排序法的性能表现。此外，

根据两种方法在稳定的内部排序法中的性能表现对比，可推断MSD法减少了递归的消耗，进而性能表现更优。