北京師範大學
BEIJING NORMAL UNIVERSITY

《数据结构》上机实验报告

第 5 次上机

学号：202011140104
姓名：李馨
学院：物理学系
专业：物理学
教师：郑新
日期：2022.11.16

1 实验过程

1.1 实验内容

1.1.1 问题描述

设计、实现一个全国大城市间的交通咨询程序，为旅客提供四种最优决策方案：（1）飞行时间最短（2）总用时最短（3）费用最小（4）中转次数最少。

1.1.2 实验要求

1. 选取合适的数据结构存储带权路线图

2. 实现单源最短路径算法

1.1.3 数据

机号	出发地	到达地	出发时间	到达时间	费用
6320	北京 上海	上海 北京	16：20 18：00	17：25 19：05	680 元
2104	北京 乌鲁木齐	乌鲁木齐 北京	8：00 10：45	9：55 11：40	1150 元
201	北京 西安	西安 北京	15：25 12：35	17：00 14：15	930 元
2323	西安 广州	广州 西安	7：15 10：15	9：35 11：35	1320 元
173	拉萨 昆明	昆明 拉萨	10：20 12：35	11：45 14：00	830 元
3304	拉萨 武汉	武汉 拉萨	14：15 16：25	15：45 17：55	890 元
82	乌鲁木齐 昆明	昆明 乌鲁木齐	9：30 13：05	12：15 15：50	1480 元
4723	武汉 广州	广州 武汉	7：05 11：25	8：45 13 ： 05	810 元

1.2 实验步骤

航班中转时，到达时间应早于要中转航班的出发时间，这一点很麻烦，不能直接使用单源最短路径的 Dijkstra 算法。不管怎样，我还是先以城市为顶点，分别以飞行时间、费用、中转次数为权值**建立有向图**。

然后，**基于 Dijkstra 算法**，对其具体实现进行统一修改：在中转时**判断是否可以中转**再决定是否记录该路径。

比较特殊的是“总用时”。可以基于飞行时间的有向图，再基于 Dijkstra 算法，在算法中进行不同于其他三种指标的修改：在中转时**判断是否可以中转**，并将**权值加上中转时间**得到新权值。

1.3 实验过程

1.3.1 从航班表到图（time24.h，flight.h，Graph.h，Graph.cpp，FlightGraph.cpp）

首先是输入数据。我们输入的是航班表。如何把航班表转化为图?恶补了一下 C++ 面向对象程序设计相关的知识，试试投入实践。分别对航班和图进行类定义，具体可见 **flight.h** 、**Graph.h** 和 **Graph.cpp**，同时其中用到 24 小时制时间的各种计算换算，又对之进行了类定义，在 **time24.h** 中。

将关键定义和成员函数摘录如下：

```
class flight {
private:
    string _num;          // 航班号
    string _pos[2];       // 出发地、目的地
    time24 _time[2];      // 出发时间、到达时间
    int _cost;            // 价格
public:
    flight() : _cost(-1) {};
    flight(string num, string pos[2], time24 t[2], int cost);

    string getNum();
    string getPos(int i);
    int getCost();
    // 返回从出发时间或到达时间（输入 0 或 1）
    time24 getTime(int i);
    /* ****** 一些计算时间的方法 ****** */
    // 返回出发和到达时间间隔
    int getInterval();
    // 若该航班之后，可以换乘 other 航班，计算出包括换乘时间的 other 航班到达目的地后总用时。否则返回
        maxWeight
    int getIntervalWith(flight& other);
    /* ****** 一些比大小的方法 ****** */
    // 比较该航班的到达时间是否在另一航班出发时间之前，即在该航班之后是否可以换乘 other 航班
    bool isBefore(flight& other);
};
```

```
// 邻接矩阵表示的图
template<class T, class W>
class MGraph : public Graph<T, W> {
private:
    vector<T> VerticesList;                 // 顶点表
    W Edge[maxVertices][maxVertices];       // 邻接矩阵
public:
    // 构造函数
```

```
    MGraph(vector<T>& v, T ed[][2], vector<W>& c, int n, int e, int d);

    int getVertexPos(T v) const;
    T getValue(int v) const;
    W getWeight(int v, int w) const;
    int firstNeighbor(int v) const;
    int nextNeighbor(int v, int w) const;
    void printMGraph(int d);
};
```

读取航班列表 **vector<flight> fls** 得到图的函数代码在 **FlightGraph.cpp** 中。依次读取数据并存入图中边、权值、顶点列表中即可。代码中的注释较为详尽。

同时为了在之后求最短路径时方便，还写了 **findFlight** 函数。比如北京顶点号为 0，上海顶点号为 1，输入后得到出发地为北京、目的地为上海的航班所对应的 **flight** 对象。

```
#include "flight.h"
#include "Graph.cpp"
// 由航班列表，不考虑时间先后，形成以城市为顶点、以飞行用时为权值的有向图
MGraph<string, int> FlyingTimeGraph(vector<flight>& fls) {…
// 由航班列表，不考虑时间先后，形成以城市为顶点、以价格为权值的有向图
MGraph<string, int> CostsGraph(vector<flight>& fls) {…
// 由航班列表，不考虑时间先后，形成以城市为顶点、权值均为 1 的有向图
MGraph<string, int> TransGraph(vector<flight>& fls) {…
// 输入由上面函数建立的图及对应的航班列表，以及在图中的边的两个顶点号，找到对应的航班
flight findFlight(MGraph<string, int>& G, vector<flight>& fls, int i, int j) {…
```

图 1: FlightGraph.cpp

由航班表建立带权有向图的具体实现，以 **FlyingTimeGraph** 函数为例，其代码如下（由于换页，分为两张图片）：

```
// 由航班列表，不考虑时间先后，形成以城市为顶点、以飞行用时为权值的有向图
MGraph<string, int> FlyingTimeGraph(vector<flight>& fls) {
    int i, j, n = fls.size();
    int numOfV, numOfE;
    vector<string> v; // 储存顶点
    string ed[n][2]; // 储存边
    vector<int> c;  // 储存权值
    for (i = 0; i < n; i++) {

        // 正向和反向边及其权值
        ed[i][0] = fls[i].getPos(0);
        ed[i][1] = fls[i].getPos(1);
        c.push_back(fls[i].getInterval());

```

图 2: 函数 FlyingTimeGraph-1

```
17          // 顶点是否已经被存，一条航班有两个地点，都要进行判断
18          for (j = 0; ; j++) {
19              if (j == v.size()) {
20                  v.push_back(fls[i].getPos(0));
21                  break;
22              }
23              if (v.at(j) == fls[i].getPos(0)) {
24                  break;
25              }
26          }
27          for (j = 0; ; j++) {
28              if (j == v.size()) {
29                  v.push_back(fls[i].getPos(1));
30                  break;
31              }
32              if (v.at(j) == fls[i].getPos(1)) {
33                  break;
34              }
35          }
36      }
37      numOfV = v.size();
38      numOfE = n;
39      return MGraph<string, int>(v, ed, c, numOfV, numOfE, 1);    // 构造函数建立图
40  }
```

图 3: 函数 FlyingTimeGraph-2

1.3.2 最短路径算法的实现（ShortestPathForFlight.cpp）

基于 Dijkstra 算法，按路径长度的递增次序，逐步产生最短路径。

1. 初始化：源点到源点的最短路径显然已找到，标记源点，以之为出发点，进行后续操作。
2. 每次从**未标记的顶点中**，选择**距源点路径长度最短**、且**允许中转**的顶点，进行标记，即该顶点已找到最短路径。
3. **经由刚找到的最短路径**，计算其后续邻接顶点（不包含已被标记的顶点）距源点的路径长度，如果该长度**比已知的**到该邻接顶点最短路径更优，更新记录该长度和最短路径。
4. 重复 2、3 共 $n-1$ 次，n 是顶点个数。由于每次必标记 1 个新顶点，故 $n-1$ 次后所有路径的最短路径都已找到（或者没有，路径长度是最大权值）。

基于 Dijkstra 算法的具体解决，针对飞行时间、费用、中转次数分别最优的三种要求，可见 **ShortestPathForFlight.cpp** 的 **ShortestPath** 函数；对于总用时最优的要求，见其中的 **ShortestTotalTime** 函数。

函数 **printFlightsPath** 用于输出路径方案。

```
#include "FlightGraph.cpp"



void ShortestPath(MGraph<string, int>& G, int v, vector<flight>& fls, vector<int>& dist, vector<int>& path) {…
58
59
void ShortestTotalTime(MGraph<string, int>& G, int v, vector<flight>& fls, vector<int>& dist, vector<int>& path) {…
120
// 输入起点 A 和 终点 B，以"A -> B -> C, 123"的形式输出路径
void printFlightsPath(MGraph<string, int>& G, vector<int>& dist, vector<int>& path, string A, string B) {…
```

图 4: ShortestPathForFlight.cpp

ShortestPath 函数中，主要是在上述算法第 3 步将要修改路径时，判定前后两条航班是否允许中转。因此需要一个前趋 $pre = \text{path}[u]$，其中 u 是刚被标记找到最短路径的顶点号。

用前述 **findFlight** 函数查询 pre 到 u 的航班，以及 u 到其邻接顶点 k 的航班，判断前者的到达时间是否在后者的出发时间之前，若在之前，则能赶上 u 到 k 的航班，允许中转。

```
        /* 下面的循环相比一般的 Dijkstra 算法加入了起飞时间是否允许中转的判定 */
        // 修改经过 u 到其他顶点的路径长度
        k = G.firstNeighbor(u);
        while (k != -1) {
            w = G.getWeight(u, k);
            pre = path[u];
            if (!S[k] && w < maxWeight && dist[u] + w < dist[k]) {
                // 邻接顶点 k 未加入 S，且经过 u 到 k 比已知路径更短
                // 判定是否可中转
                ukFlight = findFlight(G, fls, u, k);    // u 到 k 的航班
                if (pre == -1 || (k != u && findFlight(G, fls, pre, u).isBefore(ukFlight)) ) {
                    // 刚从源点 v 出发，或者 pre 到 u 的航班时间是在从 u 到 k 的航班前面的
                    dist[k] = dist[u] + w;
                    path[k] = u;
                }
            }
            k = G.nextNeighbor(u, k);
        }
    }
}
```

图 5: 函数 ShortestPath 片段

ShortestTotalTime 函数中，除了判定是否允许中转，还要对权值进行处理，加上中转时间。对 w 简单重新计算和赋值即可。

```
89          /* 下面的循环相比一般的 Dijkstra 算法加入了起飞时间是否允许中转的判定 */
90          /* * 再相比 ShortestPath 函数，w 变成了权值加上换乘时间 * */
91          // 修改经过 u 到其他顶点的路径长度
92          k = G.firstNeighbor(u);
93          while (k != -1) {
94              pre = path[u];
95              ukFlight = findFlight(G, fls, u, k);    // u 到 k 的航班
96              // 权值计算改变
97              if (pre == -1) {
98                  w = G.getWeight(u, k);
99              }
100             else if (findFlight(G, fls, pre, u).getIntervalWith(ukFlight) > 0) {
101                 w = findFlight(G, fls, pre, u).getIntervalWith(ukFlight);
102             }
103             else w = G.getWeight(u, k);
104
105             if (!S[k] && w < maxWeight && dist[u] + w < dist[k]) {
106                 // 邻接顶点 k 未加入 S，且经过 u 到 k 比已知路径更短
107                 // 判定是否可中转
108                 if (pre == -1 || (k != u && findFlight(G, fls, pre, u).isBefore(ukFlight)) ) {
109                     // 刚从源点 v 出发，或者 pre 到 u 的航班时间是在从 u 到  k 的航班前面的
110                     dist[k] = dist[u] + w;
111                     path[k] = u;
112                 }
113             }
114             k = G.nextNeighbor(u, k);
115         }
```

图 6: 函数 ShortestTotalTime 片段

1.3.3 数据输入和运行（DatasAndFinalFunction.cpp）

首先是输入数据，通过 **flight** 类的构造函数进行实例化，这些实例列成一个 **vector<flight> datas**；然后运用 1.3.1 中所述函数由 **vector<flight> datas** 建立 **Graph** G；再而运用 1.3.2 中最短路径函数得到路径及其长度；最后将路径和长度用函数 **printFlightsPath** 输出方案。

表 1 中航班表数据的输入和上述一系列函数的整合调用，在 **DatasAndFinalFunction.cpp** 文件中。

```
1   #include "ShortestPathForFlight.cpp"
2
3   // 数据输入
4   vector<flight> data() {…
57
58  void minFlyingTime(string A, string B) {…
66
67  void minTotalTime(string A, string B) {…
75
76  void minCost(string A, string B) {…
84
85  void minTrans(string A, string B) {…
```

图 7: DatasAndFinalFunction.cpp

以函数 **minFlyingTime** 为例，展示上述一系列函数的调用。

```
58  void minFlyingTime(string A, string B) {
59      vector<flight> datas = data();
60      MGraph<string, int> G = FlyingTimeGraph(datas);
61      vector<int> path;
62      vector<int> dist;
63      ShortestPath(G, G.getVertexPos(A), datas, dist, path);
64      printFlightsPath(G, dist, path, A, B);
65  }
```

图 8: 函数 minFlyingTime

1.4 运行结果

1.4.1 三种权值的图的输出（myTestMain.cpp）

```
顶点数 = 8, 边数 = 16
顶点数据为
0, 北京
1, 上海
2, 乌鲁木齐
3, 西安
4, 广州
5, 拉萨
6, 昆明
7, 武汉
输出边, 形式为 (i -> j), w:
(0 -> 1), 65
(0 -> 2), 115
(0 -> 3), 95
(1 -> 0), 65
(2 -> 0), 55
(2 -> 6), 165
(3 -> 0), 100
(3 -> 4), 140
(4 -> 3), 80
(4 -> 7), 100
(5 -> 6), 85
(5 -> 7), 90
(6 -> 2), 165
(6 -> 5), 85
(7 -> 4), 100
(7 -> 5), 90
```

(a)

```
顶点数 = 8, 边数 = 16
顶点数据为
0, 北京
1, 上海
2, 乌鲁木齐
3, 西安
4, 广州
5, 拉萨
6, 昆明
7, 武汉
输出边, 形式为 (i -> j), w:
(0 -> 1), 680
(0 -> 2), 1150
(0 -> 3), 930
(1 -> 0), 680
(2 -> 0), 1150
(2 -> 6), 1480
(3 -> 0), 930
(3 -> 4), 1320
(4 -> 3), 1320
(4 -> 7), 810
(5 -> 6), 830
(5 -> 7), 890
(6 -> 2), 1480
(6 -> 5), 830
(7 -> 4), 810
(7 -> 5), 890
```

(b)

```
顶点数 = 8, 边数 = 16
顶点数据为
0, 北京
1, 上海
2, 乌鲁木齐
3, 西安
4, 广州
5, 拉萨
6, 昆明
7, 武汉
输出边, 形式为 (i -> j), w:
(0 -> 1), 1
(0 -> 2), 1
(0 -> 3), 1
(1 -> 0), 1
(2 -> 0), 1
(2 -> 6), 1
(3 -> 0), 1
(3 -> 4), 1
(4 -> 3), 1
(4 -> 7), 1
(5 -> 6), 1
(5 -> 7), 1
(6 -> 2), 1
(6 -> 5), 1
(7 -> 4), 1
(7 -> 5), 1
```

(c)

图 9: 飞行时间、费用、中转次数权值图

1.4.2 方案的输出（main.cpp）

运行 main.cpp ，输入两个城市（中间以空格隔开），得到四种方案。随机尝试了几个城市的结果如下所示：

```
clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures/projects/project05remake$ g++ main.cpp
clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures/projects/project05remake$ ./a.out
请输入出发地和到达地，中间以空格隔开
乌鲁木齐 上海
从 乌鲁木齐 到 上海 有如下最优方案及相应花费：
飞行时间最短：
乌鲁木齐 -> 北京 -> 上海，120
总用时最短：
乌鲁木齐 -> 北京 -> 上海，400
费用最低：
乌鲁木齐 -> 北京 -> 上海，1830
中转次数最少：
乌鲁木齐 -> 北京 -> 上海，2
clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures/projects/project05remake$ g++ main.cpp
clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures/projects/project05remake$ ./a.out
请输入出发地和到达地，中间以空格隔开
拉萨 乌鲁木齐
从 拉萨 到 乌鲁木齐 有如下最优方案及相应花费：
飞行时间最短：
拉萨 -> 昆明 -> 乌鲁木齐，250
总用时最短：
拉萨 -> 昆明 -> 乌鲁木齐，330
费用最低：
拉萨 -> 昆明 -> 乌鲁木齐，2310
中转次数最少：
拉萨 -> 昆明 -> 乌鲁木齐，2
clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures/projects/project05remake$ ./a.out
请输入出发地和到达地，中间以空格隔开
西安 拉萨
从 西安 到 拉萨 有如下最优方案及相应花费：
飞行时间最短：
西安 -> 广州 -> 武汉 -> 拉萨，330
总用时最短：
西安 -> 广州 -> 武汉 -> 拉萨，640
费用最低：
西安 -> 广州 -> 武汉 -> 拉萨，3020
中转次数最少：
西安 -> 广州 -> 武汉 -> 拉萨，3
```

图 10: 方案结果-1

```
clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures/projects/project05remake$ ./a.out
请输入出发地和到达地，中间以空格隔开
西安 昆明
从 西安 到 昆明 有如下最优方案及相应花费：
飞行时间最短：
从 西安 到 昆明 在一天内没有可飞行的航班方案！
总用时最短：
从 西安 到 昆明 在一天内没有可飞行的航班方案！
费用最低：
从 西安 到 昆明 在一天内没有可飞行的航班方案！
中转次数最少：
从 西安 到 昆明 在一天内没有可飞行的航班方案！
clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures/projects/project05remake$ ./a.out
请输入出发地和到达地，中间以空格隔开
西安 乌鲁木齐
从 西安 到 乌鲁木齐 有如下最优方案及相应花费：
飞行时间最短：
从 西安 到 乌鲁木齐 在一天内没有可飞行的航班方案！
总用时最短：
从 西安 到 乌鲁木齐 在一天内没有可飞行的航班方案！
费用最低：
从 西安 到 乌鲁木齐 在一天内没有可飞行的航班方案！
中转次数最少：
从 西安 到 乌鲁木齐 在一天内没有可飞行的航班方案！
clem@Connor:/mnt/c/Users/12879/Desktop/dataStructures/projects/project05remake$ ./a.out
请输入出发地和到达地，中间以空格隔开
武汉 北京
从 武汉 到 北京 有如下最优方案及相应花费：
飞行时间最短：
武汉 -> 广州 -> 西安 -> 北京，280
总用时最短：
武汉 -> 广州 -> 西安 -> 北京，430
费用最低：
武汉 -> 广州 -> 西安 -> 北京，3060
中转次数最少：
武汉 -> 广州 -> 西安 -> 北京，3
```

图 11: 方案结果-2

2 总结

实验中遇到的最大的问题就是要考虑中转是否可行，这使得这个问题不是简单的给定权值的单源最短路径问题，而多了一些限制条件。另外，“总时间”这一最优方案的要求也使得权值变得不那么单纯。不过总体仍可以基于 Dijkstra 算法等单源最短路径算法。

综合来看，“中转是否可行”和“总时间”这两种要求，都使得路径中除第一条边之外，每条边的权值受**上一条边**影响。可以认为前者使权值保持原状或成为无穷大，后者让旧权值加上中转时间成为新权值。

还有较麻烦的问题是航班表的存储结构及其实现。尤其考虑到要把航班表转化为图，如何达到这一转化能更有效率、更简洁、更清晰，也是值得思考和优化的问题。这次为了方便用了诸如 vector、string 这类 STL 库中的东西。还尝试了面向对象程序设计，用了很长时间进行学习和调试。总体来说是很有收获，比较有成就感的。