Airline实验报告

使用方法

对于2.1-2.4、2.7、2.8的功能，直接输入1-4、7、8即可转到相应的服务；对于2.5和2.6，输入对应整数5/6后还需输入整数1-3以明确具体需要查询的服务。
转入服务后按照文档说明要求的格式进行输入即可完成查询。
注意：输入的起飞时间上限是较晚的时间，输入的起飞时间下限是较早的时间。

算法实现

数据存储

```
enum data
{
    fid,
    dedate,
    keep,
    number,
    deport,
    arport,
    detime,
    artime,
    airid,
    airmd,
    fare
};
struct flight
{
    int fid, number, deport, arport = 0;
    string keep;
    string dedate;
    string detime;
    string artime;
    int airid, airmd, fare = 0;
};//航班
struct airports
{
    int num;
    vector<flight> flights;
};
airports all[80] = { 0 };//只使用1-79代表79个机场
```

如上述代码所示，由于需要做很多次遍历，程序主体使用邻接表存储所有航班数据，而各项具体数值用一个enum枚举类型来方便引用。

功能2.1-2.3

对于2.1和2.2，使用深度优先搜索和广度优先搜索完成即可，整体时间复杂度为O(n+e)，深搜空间只额外使用一个bool found[80]的数组，为O(n)；广搜空间还额外使用了一个队列存放机场，但因为入队的机场不重复，故也不超过O(n)，因而两者相加整体空间复杂度仍为O(n)。
对于2.3，由于最多中转一次，直接枚举即可，由于循环的最大次数取决于从起始机场不重复飞两趟所能覆盖的航班，故时间复杂度不超过O(e)，除去计数的变量和时间判断的小数组外不需要额外空间。

功能2.4

由于常规dijkstra算法主要针对单边静态图，但本功能在考虑时序后需要处理多边动态图，故该功能需要修改dijkstra算法。
首先特判两机场能否直飞，若不能则开启搜索。对于起飞机场的每条航班，每次for循环只考虑其一条（相当于图中减掉了该机场其它的航班），接着从该航班目的地（记为next）开始进行dijkstra搜索，同时维护一个int mintime[80]数组记录到每个机场的最小时间，搜索的规则是：按最短时间维护一个优先级队列（节点为node，结构附在后续代码里），首先将next的到各机场时间最小的航班入队，接着取出队首元素并弹出，如果到达目的地即更新最短时间minres，如果大于最短时间则直接剪枝continue，否则取出目的地机场并再次加入符合时序且到各机场时间最短的航班（即与mintime比较，如果小才能入队），直到栈空，此时minres若保持不变则输出-1，否则输出minres即为最短时间。
时间复杂度上，由于入队的元素为边，且边在入队时和出队时都会被剪枝，且不允许出现往返飞行，因而时间复杂度<O(e loge);空间复杂度上，栈que最多入队e条边，故复杂度为O(e)。
node节点代码：

```
struct node
{
    string start;//最初起飞时间
    string last;//当前落地时间
    int lastair;//最后落地机场
    int time = 0;//消耗的时间
    vector <int> flight;//航班路径
    int trans = 0;//中转次数
    int fare = 0;//总航费
    int found[80] = { false };//检测机场路径是否重复
};//第4题中的节点
```

功能2.5

由于只要求一条航班且中转次数不超过k，因此在2.4的整体思路上，维持一个mintime数组记录各点落地时间，只有航班落地时间更小时才能入队，因为落地时间越小可以有更多的转机选择；每次取出队首元素时若其大于当前最小落地时间或最大中转次数则直接剪枝，若到达目的地即为所求的一条航班。
三种不同的要求实际上都可转化为起飞机场删去了部分航班，理论上可以重载为一种方法；特别地，对于降落时段要求可以选择反向建立邻接表或者在到达目的地时判断即可。
时间复杂度上，由于剪枝且不允许出现往返飞行，故<O(e loge)；空间复杂度上同2.4，为O(e)。

功能2.6

当要解决在时序关系下的最短航费问题时，考虑建立时序图，本程序时序图的结构仍是用机场数组存储80个机场，每个机场内有若干个时序点，每个时序点包含该点起飞的所有航班，主要代码如下：

```
struct point//时序图点
{
    int time = 0;
    int minfare = 99999;
    int tra = 9999;
    vector <flight> pflights;
};
struct newports//时序图
{
    vector <point> points;//节点按时间顺序从前往后排列，后面排序完成此操作
};
newports newall[80];//新的图也是79个机场
priority_queue <node, vector<node>, farecmp> que1;
```

在建立好时序图后，按照2.4的方法使用一个以航费作为比较因素的优先级队列进行搜索，因而现在入队的node全部保持了时序关系,当我们到达目的地时即可得到最低航费。
时间复杂度上，由于剪枝且不允许出现往返飞行，故<O(e loge)，空间复杂度上为O(e)。

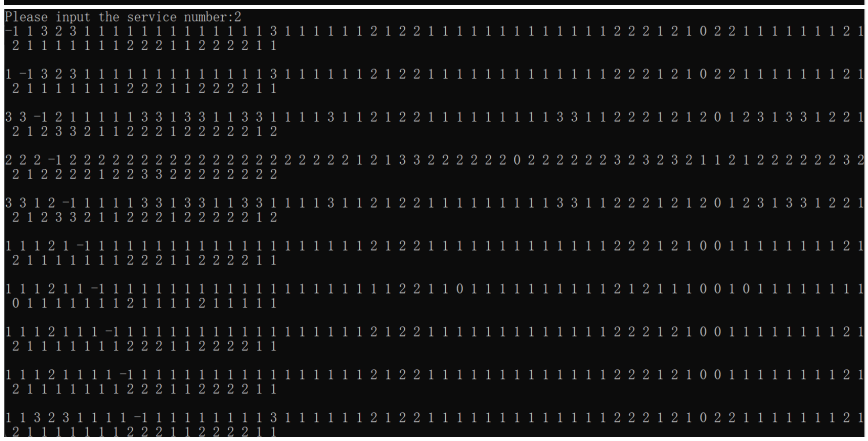
功能2.7

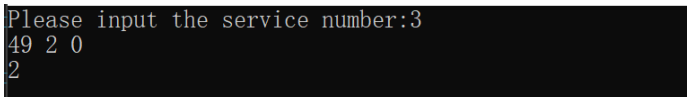
使用时序图，但优先级队列的比较元素为node的trans（中转次数），在每次拓展node时限制其中转次数和转机时间，不满足则直接丢弃次节点。注意到由于本题限制中转时间，故可以在两个机场往返飞行，其它思路整体与前几个功能一致。 时间复杂度上，最坏可达O(e² loge)，但就其平均效果而言远远不致如此低效；空间复杂度小于O(e²)。

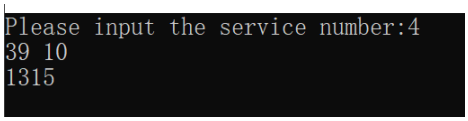
功能2.8

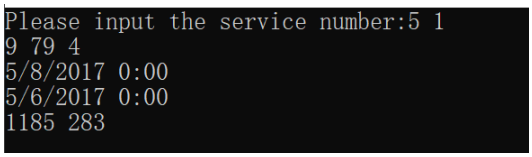
同2.7一样，由于两个条件的限制且可以往返飞行，故依然在优先级队列中将航费设为比较元素，再在node拓展后压栈前限制其转机时间即可。
由于允许往返飞行且不能限制最少费用，故最坏可达O(e² loge)，实际测试结果也远远小于该值，空间复杂度小于O(e²)，各项所花的时间和空间并不稳定，平均复杂度相对较小。

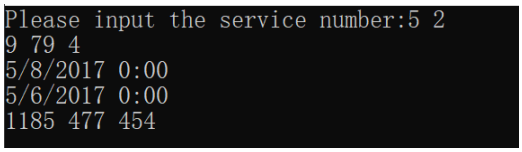
运行截图

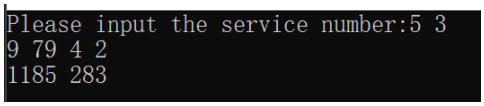


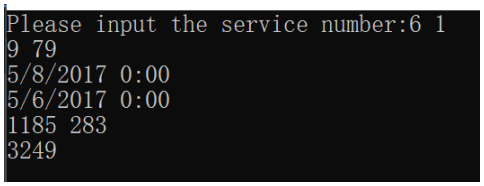


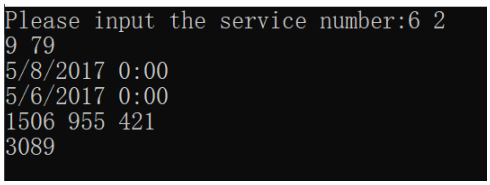


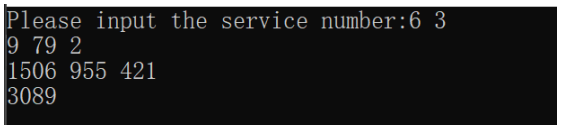


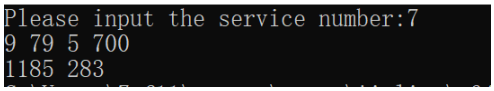


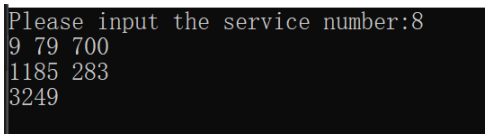












参考资料

https://blog.csdn.net/qq_36314864/article/details/112958802