

# 北京邮电大学课程设计报告

课程 设计 名称	数 据 结 构		学 院	计 算 机	指导教师	张海旻
班 级	班内序号	学 号		学生姓名	成 绩	
2018211308	16	2018211334		凌杰		
课 程 设 计 内 容	设计一个 COVID-19 疫情环境下低风险旅行模拟系统。考虑在当前 COVID-19 疫情环境下，各个城市的风险程度不一，分低分险、中风险、高风险。系统需要根据对各线路的风险评估之结果，为旅客设计一条符合旅行策略的旅行线路并输出。同时，系统能够查询当前时刻旅客所处的地点和状态。					
学 生 课程 设计 报 告  (附页)	见后文					
课 程 设 计 成 绩 评 定	<p>遵照实践教学大纲并根据以下四方面综合评定成绩：</p> <p>1、课程设计目的任务明确，选题符合教学要求，份量及难易程度</p> <p>2、团队分工是否恰当与合理</p> <p>3、综合运用所学知识，提高分析问题、解决问题及实践动手能力的效果</p> <p>4、是否认真、独立完成属于自己的课程设计内容，课程设计报告是否思路清晰、文字通顺、书写规范</p> <p>评语：</p>          <p>成绩：</p>          <p style="text-align: right;">指导教师签名：_____</p> <p style="text-align: right;">年   月   日</p>					

注：评语要体现每个学生的工作情况，可以加页。

# 目录

1. 功能需求分析.....	3
1.1 总体功能分析.....	3
1.2 时刻表与算法要求分析.....	3
1.3 系统时间推进与用户状态更新要求分析.....	4
1.4 用户状态查询分析.....	4
1.5 日志处理要求分析.....	4
2. 总体方案设计.....	5
2.1 软件开发环境.....	5
2.2 总体结构设计与模块划分.....	5
3. 数据结构.....	7
3.1 宏定义与全局变量.....	7
3.2 时刻表与图的数据结构.....	7
3.3 路径栈的数据结构.....	8
3.4 用户信息的数据结构.....	9
4. 算法设计与分析.....	9
4.1 主模块计时算法.....	9
4.2 最优路径计算的算法设计.....	10
4.2.1 深度优先搜索算法.....	10
4.2.2 限时风险最低路径算法.....	11
4.2.3 无限时风险最小路径算法.....	12
5. 测试样例.....	13
6. 评价与改进.....	14

# 1. 功能需求分析

## 1.1 总体功能分析

本系统需要完成的功能主要分三个部分，一为替汽车火车飞机的时刻表设计合适的数据结构进行存储，替用户计算出一条符合用户所设条件的最优路径（风险、时间等）；二为设置一个时钟源，系统时间按整小时向前推进，并根据各用户的旅行路线实时更新用户状态；三为开放实时查询用户旅行状态的接口，方便用户在系统运行的任意时刻查询任意用户的旅行状态；四为将用户与程序进程间的交互信息和返回信息以日志文件的形式进行记录。

## 1.2 时刻表与算法要求分析

要求城市的总数不得少于 10 个，且要为不同城市分配合适的风险值（0.5, 0.2, 0.9），各城市风险值的分配需均匀，个数均不小于 3 个。建立的时刻表假设各种交通工具均为起点到终点的直达，无经停（即不考虑停等时间），亦不考虑换乘的时间，即假设一下飞机可直接乘坐火车。且需要考虑在某城市停留的风险。各城市间班次的设置需要较为合理，不可导致任何两城市间的最优路径都是直达路径。

各交通工具的时刻表信息要用合适的数据结构进行存储以及关联，方便之后进行搜索以及路径的构建，单一班次信息可以用统一的结构体进行定义，结构体可考虑以顺序表、n 元组、线性链表、邻接表等数据结构进行存储。而时刻表数据文件在本地也要以合适的格式进行存储，方便程序在运行时读取。

计算两城市间的最优路径本质上是计算带权有向图的最短路径。图的顶点可设为城市，而边即为连接不同城市的班次。当时刻表存储妥当之后，需要以合适的数据结构描述图结构，主要考虑邻接表或邻接矩阵。权值的计算方面既要考虑在城市停留的风险，与搭乘某一交通工具的风险，且要考虑到每一条边对应的风险权值会随着时间的推移而变化，因而贪心算法不一定能够起效，若要使用经典的 Dijkstra 算法，需做不小修改，且当图中存在多条符合最优条件的路径时，如何用 Dijkstra 算法输出其他的“次优路径”需要好好考虑。其他的算法，如深度优先、广度优先、回溯法等需要较大的开销且时间复杂度较高，需要做好剪枝。

同时，若使用深度优先等算法进行路径搜索时，输出的路径是未经排序的，需要用合适的算法为所有搜索到的路径进行排序。可以考虑插入排序，这样可以一边搜索一边排序，大大降低时间复杂度；同时，选择排序也是个好的选择，因为我们不需要对所有路径都进行完整的排序，知道时间限制后，只需要用选择排序搜索出最优的几个符合条件的路径后即可结束排序。

## 1.3 系统时间推进与用户状态更新要求分析

要求旅行系统的时间以固定间隔向前推进，且旅客状态要根据系统时间与预先规划的路线实时更新以模拟旅行的过程。为实现此，可设置一全局变量作为系统时间，利用 `clock` 或 `time` 函数计时，当满足固定间隔，系统时间加 1。

同时，要求用户在输入命令时时钟源暂停更新。从这一功能上而言，若把计时模块作为主模块，用户的输入以及查询动作就像是对主模块的中断。因而当然可以把作为时钟源的主模块与用户输入模块分离成两个进程，同时监测键盘输入；但从中断原理的角度考虑，亦可以在主模块中直接监测键盘中断，路径计算与状态查询函数就作为中断程序。同时记录进入中断前、中断处理完成后的时间，将处理中断的时间排除在系统时间的更新之外。

最后，用户状态需要在每一次系统时间更新时同步进行更新，仅需要在系统时间更新时读取某一用户预先规划的路径中各个班次的时间节点，再根据旅客先前所处状态（是否完成旅行或旅行是否开始，即该条路径是否在生效）即可推断出旅客当下所处的状态。要求返回的状态要包含其所在城市和所搭乘的班次。

## 1.4 用户状态查询分析

要求实现用户输入查询指令即可查询指定用户所处状态的旅行状态。因而不同用户的旅行状态需要分开存储，因而可以考虑以文件的形式存储的本地，但这样以来将大大降低程序运行效率。还可以考虑以全局结构体数组的结构存储用户的旅行状态信息。

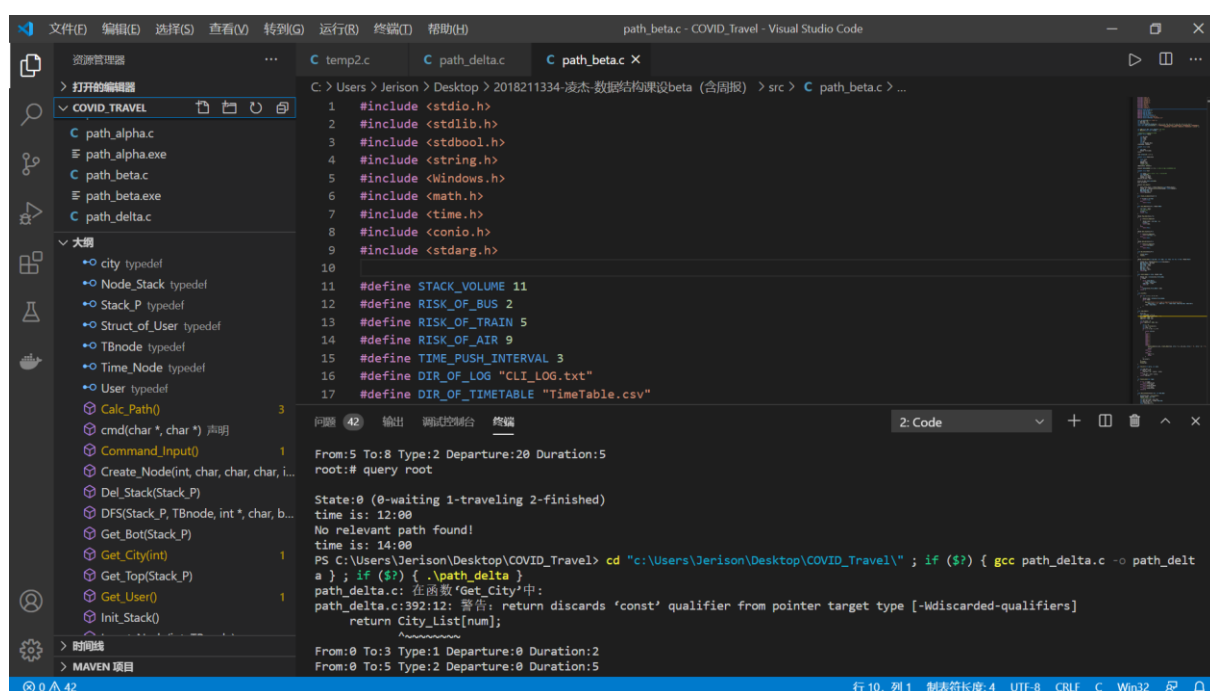
## 1.5 日志处理要求分析

要求记录所有的用户请求信息以及程序返回信息，可以考虑同步调用 `printf` 和 `fprintf`、`scanf` 和 `fscanf` 函数，为了实现复用以减少代码量，可以按照 `printf/scanf` 函数原型对两个函数进行重写，在调用时调用新写的函数。而日志信息不可或缺的一个元素便是时间戳，为打印时间戳，可以利用 `time.h` 提供的 `time_t` 类型。并通过调用 `localtime` 函数将 `time_t` 表示的时间转换为 UTC 时间，存储在一个 `struct tm` 结构体中。

## 2. 总体方案设计

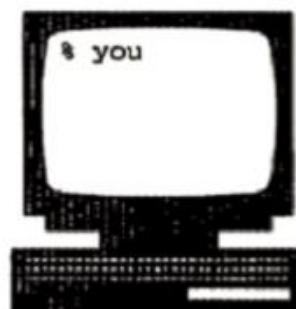
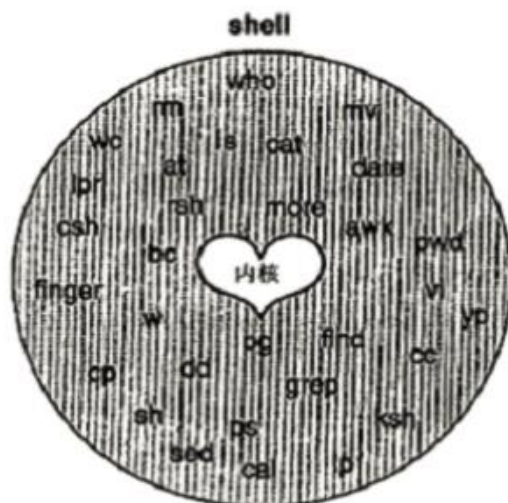
### 2.1 软件开发环境

本项目全部采用 C 语言，在 Visual Studio Code + Gcc 环境下完成编写与 debug，采用的所有库皆为 C 语言标准库。



### 2.2 总体结构设计与模块划分

受 shell 的启发，整个程序参照 shell 的思想与工作原理进行编写。

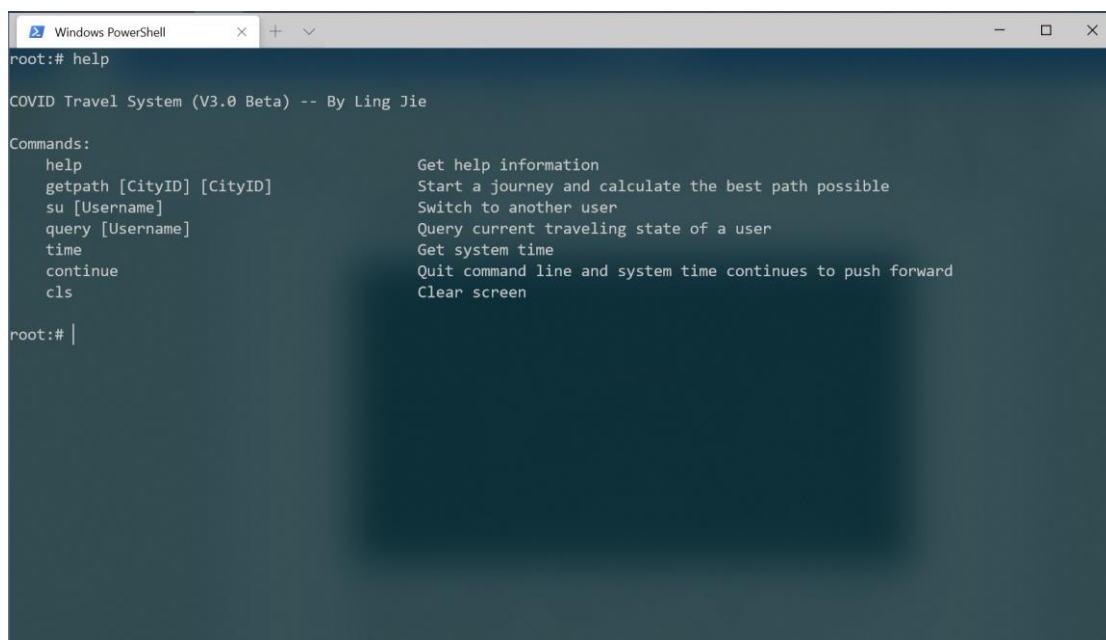


[https://blog.csdn.net/Miss\\_Monster](https://blog.csdn.net/Miss_Monster)

shell 是用户和 Linux 内核之间的接口程序，其与操作系统之间的交互为：等待用户输入，向操作系统解释用户的输入，并且处理各种各样的操作系统的输出结果。shell 提供了用户与操作系统之间通讯的方式。

而在本程序中，我也设计了一个类 shell 的模块，其负责接收用户的指定命令，将命令翻译成需要调用的接口（子函数），并向这些接口去请求数据，将返回的数据翻译后返回给用户。

而考虑到存在指令输入时需要暂停系统计时的功能需求，shell 在本程序中并非以一个独立的线程存在，而是作为主模块计时程序的中断程序存在，当监测到键盘中断（键入了“:”，参考 Vim）时，调用该模块，其接收用户指令并执行指令对应操作。目前，该 shell 模块支持以下命令：



```
Windows PowerShell
root:# help

COVID Travel System (V3.0 Beta) -- By Ling Jie

Commands:
  help                Get help information
  getpath [CityID] [CityID] Start a journey and calculate the best path possible
  su [Username]        Switch to another user
  query [Username]     Query current traveling state of a user
  time                Get system time
  continue            Quit command line and system time continues to push forward
  cls                 Clear screen

root:# |
```

不同命令对应不同模块：getpath 对应最优路径的计算与存储模块，该模块负责搜索两城市间的可能符合条件的路径，并将路径进行排序、存储；su 和 query 对应用户信息管理模块，该模块负责用户的切换、创建，用户旅行信息的查询与随时间的实时更新等等。除此之外，还有时刻表导入与管理模块与日志处理模块，前者从本地导入时刻表数据，将其转换为合适的数据结构以构造一个带权有向图，方便后续的路径搜索与计算，后者负责将程序的所有输入输出实时导出到指定日志文件。

最后，主模块负责系统时间的推进，其只与两个模块存在调用关系：当检测到键盘中断“:”时调用 shell 模块进入命令行模式，以及每当系统时间向前推进时，调用用户管理模块更新用户旅行状态。即是说，用户与程序的所有交互都在命令行模式下以输入命令的方式完成，从而避免了多层菜单间的切换，使得软件交互扁平化，且提高了软件的运行效率。

## 3. 数据结构

### 3.1 宏定义与全局变量

```
/******宏定义******/
#define STACK_VOLUME 10 //路径栈大小的最大值，由于每个城市最多只经过 1 次，最大值为 10

#define RISK_OF_BUS 2 //巴士单位时间风险值
#define RISK_OF_TRAIN 5 //火车单位时间风险值
#define RISK_OF_AIR 9 //飞机单位时间风险值
#define TIME_PUSH_INTERVAL 3 //系统时间推进间隔
#define DIR_OF_LOG "CLI_LOG.txt" //日志文件路径
#define DIR_OF_TIMETABLE "TimeTable.csv" //时刻表数据文件路径

/******全局变量******/
char Cur_User[50] = { "root" }; //当前用户名
int User_Num = 0; //用户总量
int Sys_Time = 12; //当前系统时间，默认从 12 点开始
const float RISK_OF_CITIES[10] = { 0.2, 0.5, 0.5, 0.2, 0.9, 0.9, 0.2, 0.9, 0.5, 0.2 }; //每一城市的单位时间风险值
```

### 3.2 时刻表与图的数据结构

本程序中，时刻表以 csv 格式存储在本地，如下：

```
1 Table(format:Destination-Type-Duration),,,,,,,,,,,,,,
2 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23
3 312,525,638,134,237,335,439,535,634,739,839,937,612,133,913,636,738,937,836,535,825,738,835,938,634,733
4 033,038,037,414,835,934,633,839,937,738,535,537,734,838,538,612,637,234,433,438,325,735,925,237
5 636,537,438,937,938,834,735,636,111,538,012,535,334,337,435,937,723,837,633,738,837,936,635,138
6 ,524,,,,,515,,,,,624,,,,,427,,,824,
7 ,528,,326,,,,,924,,,,,626
8 ,,,,,,,,,,825,,
9 ,111,,,,,712,,,727,,,824,,,,
10 ,922,,,,,624,,,,,
11 ,,,423,,227,,,723,,114,,,,,226,,
12 ,,,222,,,,,624,,,,,
```

从第三行起为数据正文，班次以一个三位数的方式被记录，每行对应一个城市，时刻以逗号的方式隔开。班次码的最高位表示目的地，第二位表示交通工具，最低位表示该班次从出发到到达所需时间。通过在运行程序时导入上表，我们可以构建一个用来描述城市间交通关系图的邻接表，图的顶点即城市，在邻接表中顶点节点以顺序表的方式存储，其结构如下：

```

//图的顶点节点，即城市
typedef struct city
{
    char num;
    TNode First_Node;
}city;

city Cities[10] = { 0 };    //图的顶点节点以顺序表方式存储

```

图的边界点即为班次，以链表的方式与顶点节点相连，从而构成完整的邻接表。其结构定义如下所示：

```

//图的边界点，即班次，除了 type，都以纯数字方式存储
typedef struct TNode
{
    int dp_time;           //出发时间
    char type;             //表示交通工具：1-飞机，2-火车，3-巴士
    char from;             //出发城市 ID
    char to;               //到达城市 ID
    int dur;               //该班次从出发到到达所需时间
    struct TNode* Next;    //邻接表中该节点的 next 节点
}Time_Node, *TNode;

```

### 3.3 路径栈的数据结构

本程序中，所有路径皆以栈的结构存储，以此可以方便地在进行深度优先搜索同时将每一个途径的结点压入栈，从而栈底到栈顶就构成了一条完整的路径。栈的数据结构定义如下：

```

//路径栈，用以存储路径
typedef struct Node_Stack
{
    float risk;            //该条路径的风险值
    int time;              //该条路径花费的总时间
    TNode* bot;            //栈底元素，即路径的起点
    TNode* top;            //栈顶元素+1
    //char Cur_Num;
}Node_Stack, *Stack_P;

Stack_P Path[100000] = { 0 }; //路径栈，设最多有 100000 条路径

```



## 3.4 用户信息的数据结构

本程序需要维护并根据系统时间更新每一个用户的状态信息，以供用户进行查询。用户的信息在本程序中用一下结构存储：

```
typedef struct User
{
    int state;                //0=未出发, 1 = 旅行中, 2=已完成
    char Name[50];           //用户名
    Stack_P Cur_Path;         //当前用户所设置的旅行路径
    TBNODE* Cur_Node;         //当前用户正在/即将搭乘的班次
}Struct_of_User, *User;
```

Struct\_of\_User User\_List[100]; //设最多有 100 个用户，用户信息的指针以顺序表的结构存储，方便查询

User Cur\_UserP; //当前用户（指针）

## 4. 算法设计与分析

### 4.1 主模块计时算法

主模块负责系统时间的推进，其只与两个模块存在调用关系：当检测到键盘中断“:”时调用 shell 模块进入命令行模式，以及每当系统时间向前推进时，调用用户管理模块更新用户旅行状态。同时，利用 time.h 提供的 time 函数进行程序计时，有两个 time\_t 类的变量，Last\_Time 与 Cur\_Time。前者代表上一个整点时操作系统的真实时间和当下操作系统的时间，当二者相差为固定值时（我默认设为了 3s），旅行系统时间加一。且当发生中断，从中断程序中出来后，要及时更新 Last\_Time。

```
640     time_t Last_Time = time(NULL);
641     time_t Cur_Time = Last_Time;
642     Mprintf("\r\n");
643     Mprintf("\rtime is: %d:00 ", Sys_Time);
644     Update_User();
645     while (1)
646     {
647         if (kbhit())//检测键盘中断
648         {
649             char command = getch();
650             if (command == ':')//检测到键盘中断“:”，调用类shell模块进入命令行模式
651             {
652                 Mprintf("\r\n");
653                 Mprintf("\r%s:# ", Cur_User);
654                 Command_Input();
655                 Last_Time = time(NULL);
656             }
657         }
658         Cur_Time = time(NULL);
659         if (Cur_Time - Last_Time == TIME_PUSH_INTERVAL)
660         {
661             Sys_Time = (Sys_Time + 1) % 24;
662             Mprintf("\rtime is: %d:00 ", Sys_Time);
663             Last_Time = Cur_Time;
664             Update_User();//每个整点调用Update_User函数更新用户旅行信息
665         }
666     }
```



## 4.2 最优路径计算的算法设计

### 4.2.1 深度优先搜索算法

计算两城市间的最优路径本质上是计算带权有向图的最短路径。图的顶点在本程序中设为城市，而边即为连接不同城市的班次。当时刻表存储妥当之后，需要以合适的数据结构描述图结构，本程序采取邻接表。权值的计算方面既要考虑在城市停留的风险，又要考虑搭乘某一交通工具的风险（具体公式为：等候时间\*城市单位时间风险+出发城市单位时间风险\*交通工具单位时间风险\*班次旅行时间），且要考虑到每一条边对应的风险权值会随着时间的推移而变化，因而贪心算法不一定能够起效，若要使用经典的 Dijkstra 算法，需做不小修改，且当图中存在多条符合最优条件的路径时，要利用 Dijkstra 算法输出其他的“次优路径”难度较大。因而本程序采用深度优先算法配合剪枝。

```
void DFS(Stack_P Cur_Path, TNode Cur_Node, int* Path_Num, char to, bool* flag)
{
    if (Cur_Node == NULL);
    else if (Cur_Node->to == to)
    {
        Push_Node(Cur_Path, Cur_Node);
        Record_Path(Cur_Path, *Path_Num);
        *Path_Num += 1;
        Pop_Node(Cur_Path);
        DFS(Cur_Path, Cur_Node->Next, Path_Num, to, flag); //向右搜索
    }
    else
    {
        if (flag[Cur_Node->to])
            DFS(Cur_Path, Cur_Node->Next, Path_Num, to, flag); //向右搜索
        else
        {
            Push_Node(Cur_Path, Cur_Node);
            flag[Cur_Node->from] = true;
            DFS(Cur_Path, Cities[Cur_Node->to].First_Node, Path_Num, to, flag); //向下搜索
            Pop_Node(Cur_Path);
            flag[Cur_Node->from] = false;
            DFS(Cur_Path, Cur_Node->Next, Path_Num, to, flag); //向右搜索
        }
    }
}
```

算法步骤如上所示，其中 flag 是一个长度为 10 的数组，用以标记在搜索过程中某一城市是否被途径（防止环路的产生）。该算法的主要思路是，从邻接表中起点结点的第一个边界点开始进行搜索，若该节点非空且该节点在之前未被途径，则将该节点加入当前路径（即压入栈），并且继续先向下（即走到该边结点依附的另一顶点结点处，搜索与新顶点结点相依附的所有边界点）搜索，而后将本结点弹出栈（即从路径中删除，途径该结点的所有可能至此已经搜索完毕），向右（即搜索从当前顶点节点出发的下一条边）搜索，而若当前结点所指向的另一顶点为先前经过的顶点，则不向下搜索，直接向右搜索，这是因为所有有回路的路径都不可能是最优

路径。最后，若是搜到了能到达目的结点的边界点，则将当前栈中的所有结点进行记录。由此该条路径亦被完整记录下来。重复操作，可以搜索出两城市间所有的无环路径。

**算法性能分析：**DFS 算法是一个递归算法，其需要借助一个递归路径栈，空间复杂度是  $O(|V|)$ 。而遍历图的过程实际上是对每个顶点查找其邻接点的过程，当以邻接表表示时，查找所有顶点的邻接点所需时间为  $O(|E|)$ ，访问顶点所需时间为  $O(|V|)$ ，故此，总的时间复杂度为  $O(|V|+|E|)$ 。

#### 4.2.2 限时风险最低路径算法

当 DFS 搜索并存储完毕两城市间所有路径时，各路径间是未经排序的，每一个路径栈的指针都被以顺序表的结构存储在 `Path[100000]` 中。要为这些路径进行排序，可选的一个排序算法为插入排序，好处是可以在路径生成的同时对路径进行排序，且空间复杂度极低，仅为  $O(1)$ 。但本程序选择使用的是改良后的选择排序，因为用户仅需要符合最优条件的那一条或者那几条路径，故而对所有路径进行排序是一件吃力不讨好的事，由于选择排序能够在最初的几次排序就确定最优的路径，故在此情境下选择排序更为划算。本程序改良了选择排序，即搜索完毕所有符合条件的最优路径后当即停止排序。由于有限时的要求，故在排序前从路径表的第一条路径开始，向右搜索第一个符合限时条件的路径，将其作为初始的 `Best_i`（当前最好路径的角标），之后排序时仅需从角标大于 `Best_i` 的元素开始，而若搜索后发觉无满足限时条件的路径，则直接返回 0（返回值是符合条件的最优路径数量）。

```
if(Time_Limit)//限时风险最小策略
{
    int i = 0;
    int Best_i = 0xFFFFFFFF;
    int cnt = 0;
    for(;i < Path_Num; i++)
    {
        if(Path[i]->time <= Time_Limit)
        {
            Best_i = i;
            break;
        }
    }
    if(Best_i == 0xFFFFFFFF)
        return 0;
```

而后从 `Best_i` 开始进行选择排序，倘若发现本次搜索到的符合限时条件的风险最小路径风险值不如上一次，则说明所有符合条件的限时最优风险路径都已经被排序好（放在了路径数组的最左侧），故没必要继续排序。

```

Swap_Path(Best_i, 0);
bool found = 1;
int ini = Best_i + 1;
Best_i = 0;
while(found == 1)
{
    found = 0;
    for (int j = ini; j < Path_Num; j++)
    {
        if (Path[j]->time <= Time_Limit && Path[j]->risk < Path[Best_i]->risk)
        {
            Best_i = j;
            found = 1;
        }
        else if (Path[j]->time <= Time_Limit && Path[j]->risk == Path[Best_i]->risk)
        {
            if (Path[j]->time <= Path[Best_i]->time)
            {
                Best_i = j;
                found = 1;
            }
        }
    }
    if(found)
    {
        Swap_Path(Best_i, cnt);
        Best_i = cnt;
        cnt++;
    }
}

```

**算法性能分析：**选择排序空间复杂度为  $O(1)$ ，而一般的选择排序时间复杂度为  $O(n^2)$ ，但在我改良后的选择排序中，由于最终符合条件的最优路径往往只有 1 到 2 条，本程序所采用的排序算法时间复杂度将会非常接近  $O(n)$ 。

### 4.2.3 无限时风险最小路径算法

当没有时间约束时，对路径进行排序的方法更加直接。同样采用选择排序的改良版，思路大致同上，不过省去了搜索初始 Best\_i 的操作。直接一次次遍历整个数组取得最小风险值，当本次遍历取得的最小风险值大于上一次遍历的结果时，当即停止选择排序，说明风险最小的所有路径在此时已经被筛选出。

**算法性能分析：**空间复杂度为  $O(1)$ ，时间复杂度在实际应用中接近  $O(n)$ 。

## 5.测试样例

```
root:# getpath 9 2
When would you like to start your trip? (-1 = default, 0 = now, 1 = 1 hour later...)
-1
sorting...
What's your time limit ? (unit:hour, 0 == No Limit)
0

Path1: Risk:2.6 Time:5 hrs
0:00- 3:00: Wait for 3 hours;
3:00- 5:00: Take a train from LASA to SHANGHAI;

root:# getpath 9 3
When would you like to start your trip? (-1 = default, 0 = now, 1 = 1 hour later...)
5
sorting...
What's your time limit ? (unit:hour, 0 == No Limit)
30

Path1: Risk:11.5 Time:23 hrs
17:00- 3:00: Wait for 10 hours;
3:00- 5:00: Take a train from LASA to SHANGHAI;
5:00-12:00: Wait for 7 hours;
12:00-16:00: Take a bus from SHANGHAI to GUANGZHOU;

root:# getpath 8 2
When would you like to start your trip? (-1 = default, 0 = now, 1 = 1 hour later...)
7
sorting...
What's your time limit ? (unit:hour, 0 == No Limit)
0

Path1: Risk:21.0 Time:14 hrs
0:00- 7:00: Wait for 7 hours;
7:00-14:00: Take a train from CHANGSHA to SHANGHAI;

root:# getpath 1 5
When would you like to start your trip? (-1 = default, 0 = now, 1 = 1 hour later...)
15
sorting...
What's your time limit ? (unit:hour, 0 == No Limit)
0

Path1: Risk:6.0 Time:7 hrs
8:00-10:00: Wait for 2 hours;
10:00-15:00: Take a bus from BEIJING to CHENGDU;

root:# getpath 9 7
When would you like to start your trip? (-1 = default, 0 = now, 1 = 1 hour later...)
-1
sorting...
What's your time limit ? (unit:hour, 0 == No Limit)
0

Path1: Risk:8.1 Time:11 hrs
0:00- 3:00: Wait for 3 hours;
3:00- 5:00: Take a train from LASA to SHANGHAI;
5:00- 6:00: Wait for 1 hours;
6:00-11:00: Take a bus from SHANGHAI to WUHAN;
```

---

以上是系统在不同时间点，在不同限制条件下输出的不同最优路径。在时刻表中，从编号靠后的城市出发至编号考前的城市的班次较少，因而多数情况下输出的路径都为非直达的路径。

## 6. 评价与改进

本程序完成了所有的基本功能，并完成了两项选做功能，即考虑了交通工具风险参与边权值的计算，同时支持用户开始旅行后修改旅行计划重新规划路线。本程序采用 DFS 算法与栈结构，在确保效率的同时保证精确无误地记录下每一条路径。排序时本程序改良了传统的选择排序，使得时间复杂度大大降低。在程序的交互方式上，用户与程序的所有交互都在命令行模式下以输入命令的方式完成，从而避免了多层菜单间的切换，使得软件交互扁平化，且提高了软件的使用效率。

仍可改进的方面在于，可以绘制地图来更直观地展现用户行进的过程。由于 C 语言的 EGE 库缺乏生态支撑，可考虑采用 qt 或者 pyqt 来构筑图形界面，用 folium、echarts、Pyecharts 等可视化工具完成地图的绘制。