# 电梯调度算法

**王绍文 张午潇 张子洋 景瑞**

## 一、现代电梯使用背景

电梯作为现代建筑中不可或缺的垂直交通工具，已经深深融入人们的日常生活中。随着城市化进程的加速和高层建筑的不断增加，电梯的需求也日益增长。现代电梯不仅在商业大厦、住宅楼、医院和购物中心中广泛使用，还在机场、地铁站、火车站等公共交通枢纽中发挥着重要作用。电梯的使用不仅提高了人们的生活质量，还大大改善了建筑的空间利用效率。

## 二、发展历程

电梯的发展可以追溯到古代，但现代电梯的雏形出现在19世纪。以下是电梯发展的几个重要阶段：

#### （1）早期

公元前236年，古希腊科学家阿基米德发明了原始的卷扬机，这被认为是最早的电梯雏形。

在中世纪，欧洲的修道院和城堡中使用了类似于绞盘的设备来提升货物和人。

#### （2）工业革命与现代电梯的诞生

19世纪中期，工业革命推动了城市化进程和高层建筑的发展，对电梯的需求骤增。

1853年，美国发明家伊莱莎·奥的斯（Elisha Otis）发明了安全电梯。他在纽约水晶宫的博览会上展示了电梯的安全装置，即使绳索断裂，电梯也不会坠落，这一创新大大提高了电梯的安全性和普及度。

#### （3）电动电梯的出现

1880年，德国工程师维尔纳·冯·西门子（Werner von Siemens）发明了世界上第一台电动电梯，这标志着电梯技术进入了电气化时代。

随着电动电梯的普及，高层建筑的建设得以快速发展，纽约、芝加哥等城市的摩天大楼如雨后春笋般涌现。

#### （4）现代电梯技术的发展

20世纪中期，自动控制技术的发展使电梯实现了无人操作，进一步提高了使用效率和安全性。

20世纪后期，微电子技术和计算机技术的发展，使电梯控制系统更加智能化，能够自动调度电梯、优化运行路径，提高了电梯的运行效率和乘客的满意度。

#### （5）21世纪的智能电梯

进入21世纪，物联网、人工智能、大数据等新兴技术被应用到电梯系统中，电梯变得更加智能、安全和高效。

现代智能电梯能够实时监控运行状态，预判故障并提前维护，减少了停机时间，提高了安全性和可靠性。

电梯公司还致力于开发节能环保的电梯，通过使用再生制动、节能照明等技术，减少能耗，降低对环境的影响。

## 三、常见的电梯算法实现

电梯调度的目标是优化电梯的运行，使乘客等待时间和电梯运行时间最小化。这需要考虑多种因素，包括：

乘客需求：不同楼层的乘客请求上行或下行。

电梯位置：电梯当前所在的楼层。

电梯状态：电梯的运行方向（上行或下行）和载荷情况（空载或满载）。

优先级：某些请求可能具有更高的优先级，如紧急情况。

以下是一些常见的电梯调度算法及其特点：

**FCFS（First Come First Serve）算法：**

特点：按照请求到达的顺序依次处理。

优点：实现简单，公平。

缺点：可能导致电梯运行效率低，乘客等待时间长。

**SSTF（Shortest Seek Time First）算法：**

特点：优先处理与当前楼层距离最近的请求。

优点：减少电梯的移动距离，提高效率。

缺点：可能导致远离电梯的请求长期得不到处理。

**SCAN算法（电梯算法）：**

特点：电梯按一个方向移动，到达顶层后再反向移动，类似磁盘调度的扫描算法。

优点：能够平衡各楼层的请求，减少等待时间。

缺点：实现较为复杂，可能出现部分楼层等待时间较长的情况。

**LOOK算法：**

特点：类似SCAN算法，但只到有请求的最远楼层，然后反向。

优点：进一步减少了不必要的移动，提高效率。

缺点：实现较为复杂。

**智能调度算法：**

特点：利用人工智能、机器学习等技术，根据历史数据和实时数据进行预测和优化调度。

优点：可以显著提高电梯系统的整体性能，适应动态变化的需求。

缺点：实现复杂，需大量数据和计算资源。

## 关于电梯调度算法的调研

#### （1）实现一个电梯的调度算法

##### 代码思路

这段代码模拟了一个电梯系统，使用了类似于磁盘调度中的SCAN算法（也称为电梯算法）来处理乘客的请求。电梯会在一个方向上处理所有请求，然后改变方向，处理相反方向上的请求。

##### 算法

**1.初始化电梯状态：**电梯从指定的初始楼层开始，并且初始方向设为向上。

**2.处理请求：**电梯在当前方向上处理所有请求，包括接乘客上电梯和让乘客下电梯。

**3.检查方向：**如果当前方向上没有未处理的请求，则改变电梯方向。

**4.移动电梯：**在当前方向上还有请求时，电梯移动到下一个楼层。

**5.重复处理：**重复处理步骤，直到所有请求都被处理完。

##### 函数分析

**1. initializeElevator**

void initializeElevator(Elevator \*elevator, int startFloor) {

elevator->currentFloor = startFloor; // 初始楼层设为startFloor层

elevator->direction = 1; // 初始方向设为向上

}

作用：初始化电梯的状态。设置电梯的当前楼层和初始方向。

**2. printStatus**

void printStatus(Elevator \*elevator, Request \*requests, int numRequests) {

printf("电梯当前楼层：%d, 当前方向：%s\n", elevator->currentFloor, elevator->direction == 1 ? "向上" : "向下");

printf("未接受的请求：");

int i;

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled && !requests[i].inElevator) {

printf("(%d -> %d) ", requests[i].startFloor, requests[i].endFloor);

}

}

printf("\n");

printf("已接受但未处理完的请求：");

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled && requests[i].inElevator) {

printf("(%d -> %d) ", requests[i].startFloor, requests[i].endFloor);

}

}

printf("\n");

printf("\n");

}

作用：打印电梯的当前状态，包括电梯的当前楼层、当前方向、未接受的请求和已接受但未处理完的请求。

**3. hasPendingRequests**

int hasPendingRequests(Elevator \*elevator, Request \*requests, int numRequests) {

int i;

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled) {

if (elevator->direction == 1 &&

((!requests[i].inElevator && requests[i].startFloor >= elevator->currentFloor) ||

(requests[i].inElevator && requests[i].endFloor >= elevator->currentFloor))) {

return 1;

} else if (elevator->direction == -1 &&

((!requests[i].inElevator && requests[i].startFloor <= elevator->currentFloor) ||

(requests[i].inElevator && requests[i].endFloor <= elevator->currentFloor))) {

return 1;

}

}

}

return 0;

}

作用：检查当前方向上是否还有未处理的请求。如果有，返回1；否则返回0。

**4. elevatorAlgorithm**

void elevatorAlgorithm(Elevator \*elevator, Request \*requests, int numRequests) {

int completedRequests = 0;

while (completedRequests < numRequests) {

int requestHandled = 0;

int i;

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled) {

// 处理乘客上电梯请求

if (!requests[i].inElevator && requests[i].startFloor == elevator->currentFloor) {

if ((elevator->direction == 1 && requests[i].endFloor >= elevator->currentFloor) ||

(elevator->direction == -1 && requests[i].endFloor <= elevator->currentFloor)) {

printf("接到乘客：%d楼 -> %d楼\n", requests[i].startFloor, requests[i].endFloor);

requests[i].inElevator = 1; // 乘客进入电梯

requestHandled = 1;

}

}

// 处理乘客下电梯请求

else if (requests[i].inElevator && requests[i].endFloor == elevator->currentFloor) {

printf("乘客到达目的地：%d楼\n", requests[i].endFloor);

requests[i].handled = 1;

completedRequests++;

requestHandled = 1;

}

}

}

if (!requestHandled) {

if (!hasPendingRequests(elevator, requests, numRequests)) {

elevator->direction = -elevator->direction; // 改变方向

} else {

elevator->currentFloor += elevator->direction;

if (elevator->currentFloor > MAX\_FLOORS) {

elevator->currentFloor = MAX\_FLOORS;

elevator->direction = -1; // 到达顶层时反向

} else if (elevator->currentFloor < MIN\_FLOORS) {

elevator->currentFloor = MIN\_FLOORS;

elevator->direction = 1; // 到达底层时反向

}

}

}

printStatus(elevator, requests, numRequests);

}

}

作用：实现电梯调度算法。具体步骤如下：

**（1）初始化：**设置已完成请求计数器 completedRequests 为0。

**（2）处理请求：**

遍历所有请求，检查是否有未处理的请求。

如果电梯在当前楼层并且方向允许，将乘客接入电梯。

如果乘客在当前楼层并且到达目的地，将乘客送达并标记请求为已处理。

**（3）检查方向：**

如果当前方向上没有未处理的请求，改变电梯方向。

**（4）移动电梯：**

如果当前方向上还有请求，电梯移动到下一个楼层。

在到达顶层或底层时改变方向。

**（5）打印状态：**打印电梯的当前状态。

**5. main**

int main() {

Elevator elevator;

initializeElevator(&elevator, 1); // 初始楼层设为1楼

Request requests[] = {{3, 7, 0, 0}, {-1, 9, 0, 0}, {4, 2, 0, 0}, {8, -2, 0, 0}, {2, 6, 0, 0}}; // 示例请求

int numRequests = sizeof(requests) / sizeof(requests[0]);

printStatus(&elevator, requests, numRequests);

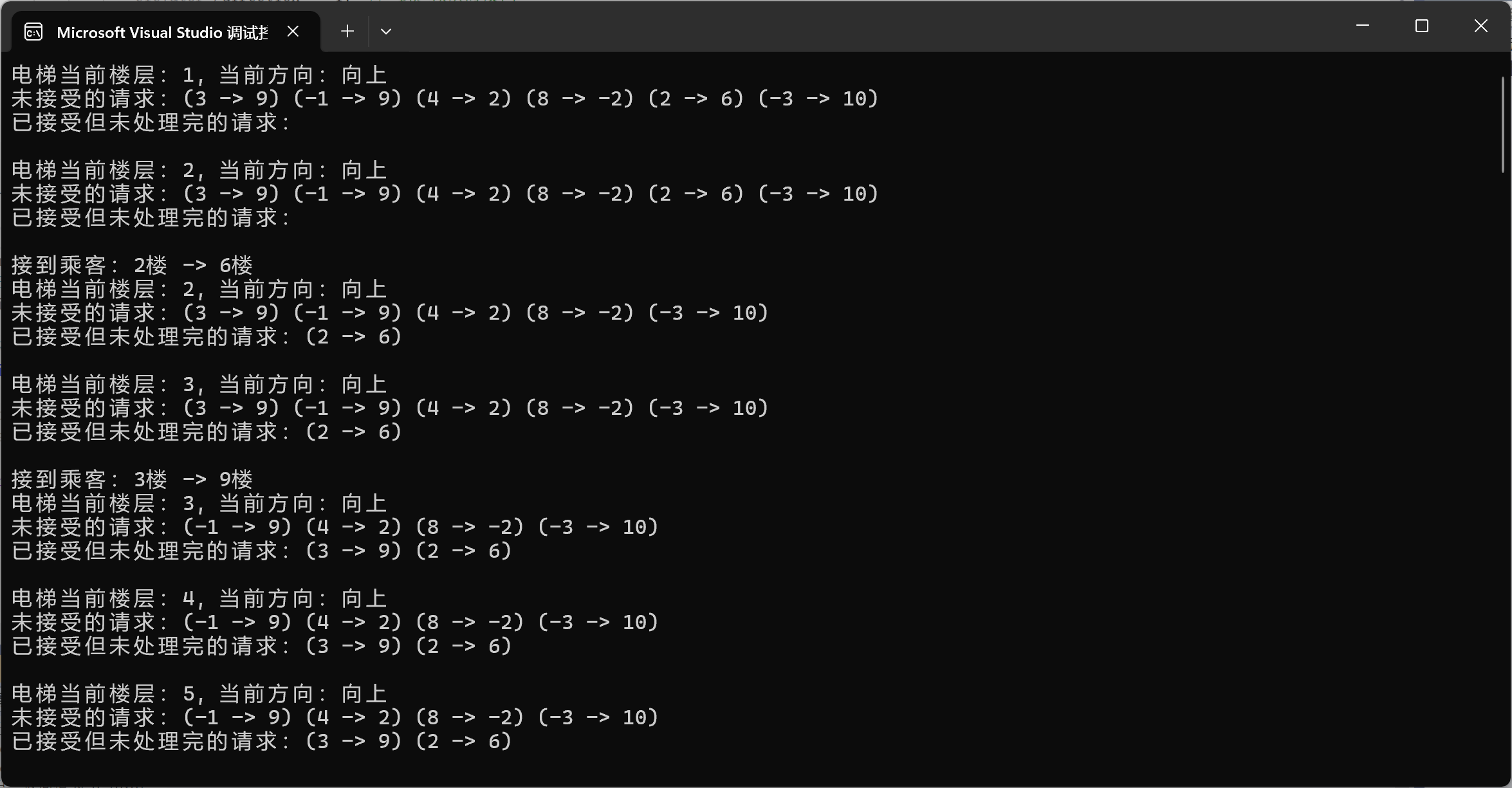
elevatorAlgorithm(&elevator, requests, numRequests);

return 0;

}

作用：程序入口。初始化电梯和请求，调用电梯调度算法，并打印电梯状态。

##### 代码运行结果



文本

描述已自动生成

#### 实现两个电梯的调度算法

代码思路

在request结构体中加入一个新的变量eleNum，表示进入的电梯号码，在电梯运行初始时给eleNum赋值为对应的电梯号。

创建两个Elevator变量，分别对其进行单个电梯的实现，要注意改变相应的控制条件，否则容易对request中的变量赋值混乱。

##### 改进部分

**改进结构体：**

// 请求状态

typedef struct {

int startFloor; // 乘客起始楼层

int endFloor; // 乘客目的楼层

int inElevator; // 是否在电梯内

int handled; // 请求是否已处理

int eleNum; // 标记上了哪个电梯

} Request;

**改进elevatorAlgorithm部分：**

// 电梯调度算法

void elevatorAlgorithm(Elevator\* elevator, Elevator\* elevator2, Request\* requests, int numRequests) {

int completedRequests = 0;

while (completedRequests < numRequests) {

int requestHandled = 0;

// 电梯1

for (int i = 0; i < numRequests; i++) {

if (!requests[i].handled && !requests[i].inElevator) {

if (requests[i].startFloor == elevator->currentFloor) {

if ((elevator->direction == 1 && requests[i].endFloor >= elevator->currentFloor) ||

(elevator->direction == -1 && requests[i].endFloor <= elevator->currentFloor)) {

printf("电梯1接到乘客：%d楼 -> %d楼\n", requests[i].startFloor, requests[i].endFloor);

requests[i].inElevator = 1;

requestHandled = 1;

requests[i].eleNum = 1;

}

}

}

else if (requests[i].inElevator && requests[i].eleNum == 1 && requests[i].endFloor == elevator->currentFloor ) {

printf("电梯1乘客到达目的地：%d楼\n", requests[i].endFloor);

requests[i].handled = 1;

completedRequests++;

requestHandled = 1;

}

}

// 电梯2

for (int i = 0; i < numRequests; i++) {

if (!requests[i].handled && !requests[i].inElevator) {

if (requests[i].startFloor == elevator2->currentFloor) {

if ((elevator2->direction == 1 && requests[i].endFloor >= elevator2->currentFloor) ||

(elevator2->direction == -1 && requests[i].endFloor <= elevator2->currentFloor)) {

printf("电梯2接到乘客：%d楼 -> %d楼\n", requests[i].startFloor, requests[i].endFloor);

requests[i].inElevator = 1;

requestHandled = 1;

requests[i].eleNum = 2;

}

}

}

else if (requests[i].inElevator && requests[i].eleNum == 2 && requests[i].endFloor == elevator2->currentFloor ) {

printf("电梯2乘客到达目的地：%d楼\n", requests[i].endFloor);

requests[i].handled = 1;

completedRequests++;

requestHandled = 1;

}

}

if (!requestHandled) {

// 如果未处理到请求，且该方向上无请求，则改变方向

if (!hasPendingRequests(elevator, requests, numRequests)) {

elevator->direction = -elevator->direction;

}

if (!hasPendingRequests(elevator2, requests, numRequests)) {

elevator2->direction = -elevator2->direction;

}

}

// 电梯各运行1层

elevator->currentFloor += elevator->direction;

elevator2->currentFloor += elevator2->direction;

// 若到达边界层，则改变方向

if (elevator->currentFloor > MAX\_FLOORS) {

elevator->currentFloor = MAX\_FLOORS;

elevator->direction = -1;

}

else if (elevator->currentFloor < MIN\_FLOORS) {

elevator->currentFloor = MIN\_FLOORS;

elevator->direction = 1;

}

if (elevator2->currentFloor > MAX\_FLOORS) {

elevator2->currentFloor = MAX\_FLOORS;

elevator2->direction = -1;

}

else if (elevator2->currentFloor < MIN\_FLOORS) {

elevator2->currentFloor = MIN\_FLOORS;

elevator2->direction = 1;

}

printStatus(elevator, elevator2, requests, numRequests);

}

}

##### 代码运行结果

屏幕上有字

描述已自动生成

手机屏幕的截图

描述已自动生成

如果要实现多个电梯的调度工作，可能需要更加复杂的数据结构和算法，如利用队列等来存储和操作电梯对象和相应的请求，比如，可以使用优先队列来存储优先级（电梯可能有突发情况需要急用等），按照优先队列中的优先级进行电梯调度等。一般情况下可以按照当前电梯与请求接入点的距离来决定调用哪个电梯，而且在实际情况中电梯调度通常不会完全与SCAN调度算法相同（因为请求有时间与先后顺序），故需要完整的实现一整个电梯的调度还需要考虑到很多更加复杂的情况。

## 心得体会

通过对电梯技术的详细了解，我们可以看到现代电梯技术的复杂性和先进性。这些技术和装置的不断创新和优化，不仅提高了电梯的运行效率和乘客的舒适性，更重要的是极大地保障了乘客的安全。电梯作为高层建筑中的重要垂直交通工具，承担了大量的日常运输任务。其技术的发展和完善，不仅仅依赖于机械结构和电气系统的进步，还涉及到计算机科学、自动化控制、材料科学等多个领域的综合应用。每一次技术革新，都是各个领域专家和工程师们共同努力的结果。现代电梯技术的不断进步，还对节能环保提出了更高的要求。通过使用再生制动系统、节能照明、智能控制等技术，现代电梯在运行过程中显著降低了能耗，对环境保护做出了积极贡献。

而通过模拟电梯调度算法，可以得出：

**算法选择的重要性：**不同的调度算法在不同场景下有不同的表现。简单的算法如FCFS虽然容易实现，但在复杂的电梯系统中效率可能较低。更复杂的算法如智能调度可以显著提高系统性能，但实现难度也较大。

**需求分析和优化：**在设计电梯调度系统时，需要充分考虑乘客的需求和建筑的特点。例如，在高峰时段需要优化电梯的载客量和运行效率，而在非高峰时段则可以更多考虑节能。

动态调整和适应：电梯调度系统需要具备动态调整能力，能够根据实时情况调整调度策略。例如，检测到某个楼层频繁请求电梯时，可以优先调度电梯到该楼层。

**安全性和可靠性：**电梯调度系统不仅需要高效，还需要保证乘客的安全。系统需要具备多重安全保护措施，如限速器、安全钳等，防止意外事故的发生。

**持续改进和技术创新：**随着科技的进步，电梯调度技术也在不断发展。人工智能和物联网技术的应用，使得电梯调度系统越来越智能化和高效化。未来，我们可以期待更多的技术创新，为电梯调度带来更大的突破。

通过不断的模拟和优化，我们可以设计出更高效、安全、智能的电梯调度系统，为现代高层建筑的垂直交通提供更优质的服务。

## 源代码

**一个电梯调度算法**

#include <stdio.h>

#include <stdlib.h>

#define MIN\_FLOORS -3 // 假设电梯服务的最低楼层数为-3

#define MAX\_FLOORS 10 // 假设电梯服务的最高楼层数为10

// 电梯状态

typedef struct {

int currentFloor; // 当前楼层

int direction; // 方向，1表示向上，-1表示向下

} Elevator;

// 请求状态

typedef struct {

int startFloor; // 乘客起始楼层

int endFloor; // 乘客目的楼层

int inElevator; // 是否在电梯内

int handled; // 请求是否已处理

} Request;

void initializeElevator(Elevator \*elevator, int startFloor) {

elevator->currentFloor = startFloor; // 初始楼层设为startFloor层

elevator->direction = 1; // 初始方向设为向上

}

void printStatus(Elevator \*elevator, Request \*requests, int numRequests) {

if (elevator->currentFloor == 0){

return;

}

printf("电梯当前楼层：%d, 当前方向：%s\n", elevator->currentFloor, elevator->direction == 1 ? "向上" : "向下");

printf("未接受的请求：");

int i;

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled && !requests[i].inElevator) {

printf("(%d -> %d) ", requests[i].startFloor, requests[i].endFloor);

}

}

printf("\n");

printf("已接受但未处理完的请求：");

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled && requests[i].inElevator) {

printf("(%d -> %d) ", requests[i].startFloor, requests[i].endFloor);

}

}

printf("\n");

printf("\n");

}

// 检查当前方向上是否还有未处理的请求

int hasPendingRequests(Elevator \*elevator, Request \*requests, int numRequests) {

int i;

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled) {

if (elevator->direction == 1 &&

((!requests[i].inElevator && requests[i].startFloor >= elevator->currentFloor) ||

(requests[i].inElevator && requests[i].endFloor >= elevator->currentFloor))) {

return 1;

} else if (elevator->direction == -1 &&

((!requests[i].inElevator && requests[i].startFloor <= elevator->currentFloor) ||

(requests[i].inElevator && requests[i].endFloor <= elevator->currentFloor))) {

return 1;

}

}

}

return 0;

}

// 电梯调度算法

void elevatorAlgorithm(Elevator \*elevator, Request \*requests, int numRequests) {

int completedRequests = 0;

while (completedRequests < numRequests) {

int requestHandled = 0;

int i;

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled) {

// 处理乘客上电梯请求

if (!requests[i].inElevator && requests[i].startFloor == elevator->currentFloor) {

if ((elevator->direction == 1 && requests[i].endFloor >= elevator->currentFloor) ||

(elevator->direction == -1 && requests[i].endFloor <= elevator->currentFloor)) {

printf("接到乘客：%d楼 -> %d楼\n", requests[i].startFloor, requests[i].endFloor);

requests[i].inElevator = 1; // 乘客进入电梯

requestHandled = 1;

}

}

// 处理乘客下电梯请求

else if (requests[i].inElevator && requests[i].endFloor == elevator->currentFloor) {

printf("乘客到达目的地：%d楼\n", requests[i].endFloor);

requests[i].handled = 1;

completedRequests++;

requestHandled = 1;

}

}

}

if (!requestHandled) {

if (!hasPendingRequests(elevator, requests, numRequests)) {

elevator->direction = -elevator->direction; // 改变方向

} else {

elevator->currentFloor += elevator->direction;

if (elevator->currentFloor > MAX\_FLOORS) {

elevator->currentFloor = MAX\_FLOORS;

elevator->direction = -1; // 到达顶层时反向

} else if (elevator->currentFloor < MIN\_FLOORS) {

elevator->currentFloor = MIN\_FLOORS;

elevator->direction = 1; // 到达底层时反向

}

}

}

printStatus(elevator, requests, numRequests);

}

}

int main() {

Elevator elevator;

initializeElevator(&elevator, 1); // 初始楼层设为1楼

Request requests[] = {{3, 9, 0, 0}, {-1, 9, 0, 0}, {4, 2, 0, 0}, {8, -2, 0, 0}, {2, 6, 0, 0}, {-3, 10, 0, 0}}; // 示例请求

int numRequests = sizeof(requests) / sizeof(requests[0]);

printStatus(&elevator, requests, numRequests);

elevatorAlgorithm(&elevator, requests, numRequests);

return 0;

}

**两个电梯调度算法**

#include <stdio.h>

#include <stdlib.h>

#define MIN\_FLOORS -3 // 假设电梯服务的最低楼层数为-3

#define MAX\_FLOORS 10 // 假设电梯服务的最高楼层数为10

// 电梯状态

typedef struct {

int currentFloor; // 当前楼层

int direction; // 方向，1表示向上，-1表示向下

} Elevator;

// 请求状态

typedef struct {

int startFloor; // 乘客起始楼层

int endFloor; // 乘客目的楼层

int inElevator; // 是否在电梯内

int handled; // 请求是否已处理

int eleNum; // 标记上了哪个电梯

} Request;

void initializeElevator(Elevator\* elevator, int startFloor, int direction) {

elevator->currentFloor = startFloor; // 初始楼层设为startFloor层

elevator->direction = direction; // 初始方向设定

}

void printStatus(Elevator\* elevator, Elevator\* elevator2, Request\* requests, int numRequests) {

printf("电梯1当前楼层：%d, 当前方向：%s\n", elevator->currentFloor, elevator->direction == 1 ? "向上" : "向下");

printf("电梯2当前楼层：%d, 当前方向：%s\n", elevator2->currentFloor, elevator2->direction == 1 ? "向上" : "向下");

printf("未接受的请求：");

int i;

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled && !requests[i].inElevator) {

printf("(%d -> %d) ", requests[i].startFloor, requests[i].endFloor);

}

}

printf("\n");

printf("已接受但未处理完的请求：");

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled && requests[i].inElevator) {

printf("(%d -> %d) ", requests[i].startFloor, requests[i].endFloor);

}

}

printf("\n\n");

}

// 检查当前方向上是否还有未处理的请求

int hasPendingRequests(Elevator\* elevator, Request\* requests, int numRequests) {

int i;

for (i = 0; i < numRequests; i++) {

if (!requests[i].handled) {

if (elevator->direction == 1 &&

((!requests[i].inElevator && requests[i].startFloor >= elevator->currentFloor) ||

(requests[i].inElevator && requests[i].endFloor >= elevator->currentFloor))) {

return 1;

}

else if (elevator->direction == -1 &&

((!requests[i].inElevator && requests[i].startFloor <= elevator->currentFloor) ||

(requests[i].inElevator && requests[i].endFloor <= elevator->currentFloor))) {

return 1;

}

}

}

return 0;

}

// 电梯调度算法

void elevatorAlgorithm(Elevator\* elevator, Elevator\* elevator2, Request\* requests, int numRequests) {

int completedRequests = 0;

while (completedRequests < numRequests) {

int requestHandled = 0;

// 电梯1

for (int i = 0; i < numRequests; i++) {

if (!requests[i].handled && !requests[i].inElevator) {

if (requests[i].startFloor == elevator->currentFloor) {

if ((elevator->direction == 1 && requests[i].endFloor >= elevator->currentFloor) ||

(elevator->direction == -1 && requests[i].endFloor <= elevator->currentFloor)) {

printf("电梯1接到乘客：%d楼 -> %d楼\n", requests[i].startFloor, requests[i].endFloor);

requests[i].inElevator = 1;

requestHandled = 1;

requests[i].eleNum = 1;

}

}

}

else if (requests[i].inElevator && requests[i].eleNum == 1 && requests[i].endFloor == elevator->currentFloor ) {

printf("电梯1乘客到达目的地：%d楼\n", requests[i].endFloor);

requests[i].handled = 1;

completedRequests++;

requestHandled = 1;

}

}

// 电梯2

for (int i = 0; i < numRequests; i++) {

if (!requests[i].handled && !requests[i].inElevator) {

if (requests[i].startFloor == elevator2->currentFloor) {

if ((elevator2->direction == 1 && requests[i].endFloor >= elevator2->currentFloor) ||

(elevator2->direction == -1 && requests[i].endFloor <= elevator2->currentFloor)) {

printf("电梯2接到乘客：%d楼 -> %d楼\n", requests[i].startFloor, requests[i].endFloor);

requests[i].inElevator = 1;

requestHandled = 1;

requests[i].eleNum = 2;

}

}

}

else if (requests[i].inElevator && requests[i].eleNum == 2 && requests[i].endFloor == elevator2->currentFloor ) {

printf("电梯2乘客到达目的地：%d楼\n", requests[i].endFloor);

requests[i].handled = 1;

completedRequests++;

requestHandled = 1;

}

}

if (!requestHandled) {

// 如果未处理到请求，且该方向上无请求，则改变方向

if (!hasPendingRequests(elevator, requests, numRequests)) {

elevator->direction = -elevator->direction;

}

if (!hasPendingRequests(elevator2, requests, numRequests)) {

elevator2->direction = -elevator2->direction;

}

}

// 电梯各运行1层

elevator->currentFloor += elevator->direction;

elevator2->currentFloor += elevator2->direction;

// 若到达边界层，则改变方向

if (elevator->currentFloor > MAX\_FLOORS) {

elevator->currentFloor = MAX\_FLOORS;

elevator->direction = -1;

}

else if (elevator->currentFloor < MIN\_FLOORS) {

elevator->currentFloor = MIN\_FLOORS;

elevator->direction = 1;

}

if (elevator2->currentFloor > MAX\_FLOORS) {

elevator2->currentFloor = MAX\_FLOORS;

elevator2->direction = -1;

}

else if (elevator2->currentFloor < MIN\_FLOORS) {

elevator2->currentFloor = MIN\_FLOORS;

elevator2->direction = 1;

}

printStatus(elevator, elevator2, requests, numRequests);

}

}

int main() {

Elevator elevator, elevator2;

initializeElevator(&elevator, 1, 1); // 初始楼层设为1楼，方向为向上

initializeElevator(&elevator2, 10, -1); // 初始楼层设为10楼，方向为向下

Request requests[] = { {3, 9, 0, 0}, {2, 8, 0, 0}, {4, 2, 0, 0}, {10, 1, 0, 0}, {2, 6, 0, 0}, {5, 10, 0, 0} }; // 示例请求

int numRequests = sizeof(requests) / sizeof(requests[0]);

printStatus(&elevator, &elevator2, requests, numRequests);

elevatorAlgorithm(&elevator, &elevator2, requests, numRequests);

return 0;

}