西安电子科技大学
XIDIAN UNIVERSITY

博弈论大作业三

十字路口智能交通控制算法

任俊杰
1702052
17170120015

智能控制 智能交通控制算法

一、实验内容

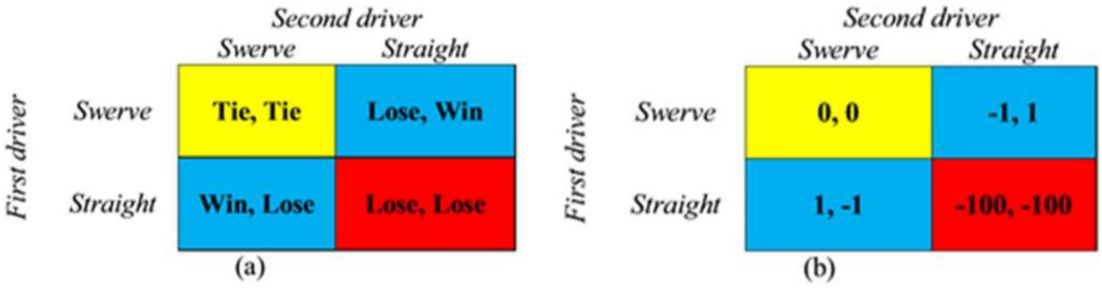
车联网环境下一个基于博弈论的十字路口交通控制算法，即该应用的核心技术之一。

在车联网环境下，车辆可以与交叉路口的中央管理中心通信，以提供其瞬时速度，位置和方向。在中央管理中心融合所有的信息之后，决定每辆车的行动（加速，减速或保持当前速度），以避免碰撞并为每辆车提供最低延迟。

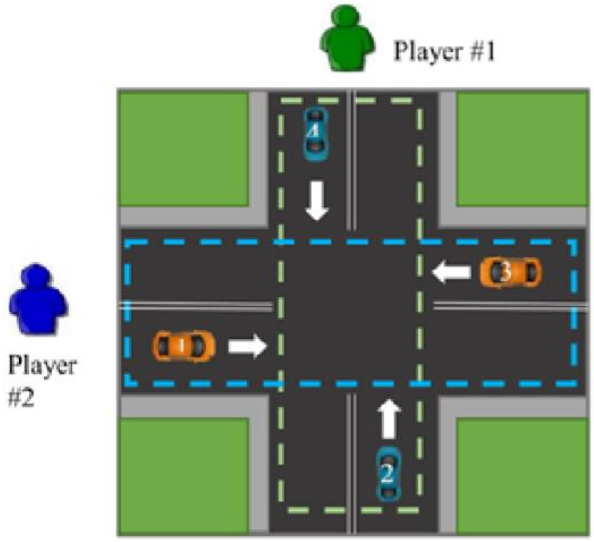
二、实验相关知识背景说明

2.1 胆小鬼博弈

网胆小鬼博弈（英文：The game of chicken），又译懦夫博弈，是博弈论中一个影响深远的模型，逻辑就是“不要命的最大”。模型中，两名车手相对驱车而行，谁最先转弯的一方被耻笑为“胆小鬼”（chicken），让另一方胜出，因此这博弈模型在英文 中 称 为 The Game of Chicken（懦夫游戏），但如果两人拒绝转弯，任由两车相撞，最终谁都无法受益。其收益矩阵如图所示。本方法受胆小鬼博弈启发。



2.2 玩家



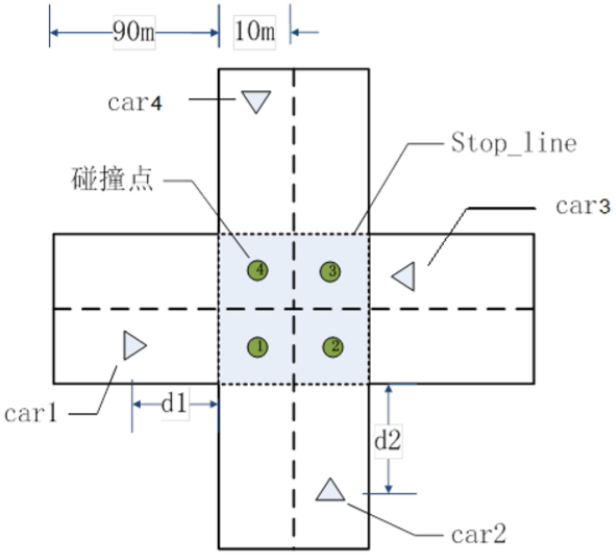
该博弈只有两名玩家。$Player\#1$ 决定车辆 2 和 4 的动作，而$Player\#2$决定车辆 1 和 3 的动作，如图 2 所示。每个玩家想要以这样的方式控制车辆以最小化它们在交叉路口的延迟并保证它们安全地（未发生碰撞）穿过十字路口。

2.3 玩家动作

每个玩家最三种可能的动作：加速（表示为 1），减速（表示为−1）或以当前速度继续行驶（表示为 0）。由于每个玩家有两辆车并且每辆车都有动作组，因此玩家的动作是车辆动作的笛卡尔乘积。 每辆汽车可以采取的动作是{1, 0, −1}，则 每个玩家可以采取的动作有 9 种，即{ 1 , 0 , −1 }×{ 1 , 0 , −1 } = {(1,1), (1,0), (0,1), (1,−1), (−1,1), (0,0), (0,−1), (−1,0), (−1,−1)}，例如 $player\#1$ 的动作为(1,0)代表他命令 2 号车加速，3 号车保持当前速度行驶。

三、问题模型介绍

3.1 道路结构：



3.2 数据说明

道路结构如上图所示。每次模拟中随机生成 4 辆车的数据，每辆车的数据的形式为(d,v)，如图所示，d为车辆距离停车线 stop _line 的距离，范围为[0,90]，v为车辆的当前速度，初始化范围为[20,40]，规则是加速后速度不能超过 40，减速下限无限制。

3.3 根据车辆动作速度更新

为简化计算过程，实验中加速（action = 1）直接将原速度提高 40%，减速（action = −1）直接将原速度减少 40%，保持当前速度则速度不变。速度的更新公式为：

$$v\_new = v \times (1 + action \times 0.4)$$

3.4 碰撞判定

以图 3 中 4 号碰撞点为例，计算 3 号车与 4 号车到达碰撞点 4 的时间差，如时间差小于 1s 则判定为碰撞，否则判定为通过。

3.5 博弈时间（通行时间）

一次模拟中 4 个碰撞点中有一个发生了碰撞则为博弈中双输的情况，该次博弈损失为无穷大（inf）。 如未发生碰撞，则博弈损失为 4 辆车通过该十字路口的平均时间。一辆车(d,v)的通行时间为：

$$t=\begin{cases}\dfrac{90-d}{v}+\dfrac{110+d}{v_{new}}, & if\ 未发生碰撞\\ inf, & if\ 发生碰撞\end{cases}$$

此处假设车辆经过一次速度调整后，在驶出路口前不再调整速度。

四、实验要求

4.1 计算博弈损失矩阵

四辆车的数据分别为

$$car1=(45,23)$$
$$car2=(71,24)$$
$$car3=(47,28)$$
$$car4=(76,28)$$

遍历所有的决策组合，计算出博弈损失矩阵（通行时间），并给出最快通过时的玩家的决策：

4.2 与传统方法对比

传统方法：先截断一个方向的车流，让另一个方向的车全部通行后在开流。例如先让车 2，4 通行，后让 1，3 通行。通行时间为

$$t=\max\left(\frac{200}{v_1},\frac{200}{v_3}\right)+\max\left(\frac{200}{v_2},\frac{200}{v_4}\right)$$

每次实验中，随机生成 4 辆车的数据，计算该数据在传统方法下的通行时间，再计算 4.1 中的矩阵，以矩阵中的最小值为博弈方法的通行时间。

进行 10 次实验，分别记录 10 次实验中博弈方法和传统方法的结果，

五、算法简介

1、生成四辆车，并随机初始化车的初始信息，包括初始速度与初始位置。

2、对于实验要求 1，需要指定数据，在此对每辆车的位置和速度进行赋初值，对于实验要求 2，则跳过该步骤。

3、生成每个玩家的所有的策略组合。

4、遍历玩家 1 和玩家 2 所有的策略组合：

1）根据每位玩家的决策，计算所对应车辆的新速度。

2）判断对应车辆是否撞击，若撞击，则置通过时间为无穷，否则按照公式计算车辆通过时间。

3）将通过时间保存至二维列表。

5、使用传统方式对应的通过时间公式计算通行时间。

6、返回博弈方法的通行时间矩阵、传统方法的通行时间。

7、搜索博弈损失矩阵中的最短通行时间，并找出对应玩家 1 和玩家 2 的策略，与传统方式进行比较。

六、算法具体实现（Python）

6.0 用到的库和自定义类

使用了 random 和 numpy 连个基础库，用于随机初始化和相关数学计算。自定义的 CAR 类存储车辆的各种信息和计算函数，详解如下：

__init__()：初始化车辆时，随机在（20，40）之间确定车辆的初始速度，随即在（0，90）之间去顶车辆的初始位置，并初始化一些之后用到的数据。

action()：根据玩家的决策来更新速度（依据 3.3 中公式），新速度单独保

存。

Caclu_time()：计算该车辆在博弈方法下的通行时间（依据 3.5 中公式）。

print()：打印车辆信息，便于观察。

```
import random
import numpy as np

class CAR:
    def __init__(self,i): # d:[0,90] v:[20,40]
        super().__init__()
        self.name = i
        self.v = random.randint(20,40)
        self.v_new = 0
        self.d = random.randint(0,90)
        self.cross_time = 0

    def action(self,act):
        self.v_new = (1 + act*0.4)*self.v

    def caclu_time(self):
        self.cross_time = (90-self.d)/self.v+(110 + self.d)/self.v_new
        return self.cross_time

    def print(self):
        print("car_%d:(%.2f,%.2f)"%(self.name,self.d,self.v),"cross_ti
me:",self.cross_time)   #(d,v)
```

6.1 更新初值、生成玩家策略

博弈的主程序位于 traffic_game()下，下面是前半部分：

1）生成四辆车，车辆初始信息在类初始化中自动确定。

2）根据题目 1 要求，指定所有车辆的数据信息，做题目 2 时注释掉掠过即可。

3）根据{ 1 , 0 , −1 }×{ 1 , 0 , −1 }，使用两个循环得到每个玩家的所有策略组合，存储在 all_action 中。

```
def traffic_game():
    # 随机初始化车辆信息
```

```
cars = [CAR(i+1) for i in range(4)]
# 问题 1，指定数据
cars[0].d,cars[0].v = 45,23
cars[1].d,cars[1].v = 71,24
cars[2].d,cars[2].v = 47,28
cars[3].d,cars[3].v = 76,28

for car in cars:
    car.print()

# 所有决策
all_actions = []
for i in [1,0,-1]:
    for j in[1,0,-1]:
        all_actions.append([i,j])
# print(all_actions)
```

6.2 遍历所有策略组合并计算传统时间

1）对于每个玩家的策略，控制对应车辆进行决策（更新速度），这里直接调用了类中定义好的函数即可。（player_1 对应 2、4，但由于列表索引从 0 开始，程序中为 car[1]、car[3]，player_2 同理）

2）调用 judge_crash()，判断相对应车辆是否撞击，撞击则为 inf，没有撞击则调用类中函数，计算通过时间，并保存数据至 loss_array 二维列表中。（判断撞击函数在下面给出）

3）根据 4.2 中公式计算传统方法通过时间，返回博弈方法的通行时间矩阵、传统方法的通行时间。

```
# 开始遍历所有的决策组合
loss_array = [[0 for j in range(9)] for i in range(9)]
i = j = 0
for player_1 in all_actions:
    for player_2 in all_actions:
        # player_1 决策,控制 2，4
        cars[1].action(player_1[0])
        cars[3].action(player_1[1])
        # player_2 决策，控制 1，3
        cars[0].action(player_2[0])
```

```
            cars[2].action(player_2[1])

            # 判断是否撞击
            if judge_crash(cars):
                time_loss = float('inf')
            else:
                time_loss = 0
                for car in cars:
                    time_loss += car.caclu_time()

            loss_array[i][j] = time_loss/4

            j += 1
        i += 1
        j = 0

    # 传统方式的通过时间
    t_old = max(200/cars[0].v,200/cars[2].v) + max(200/cars[1].v,200/c
ars[3].v)
    return all_actions,loss_array,t_old
```

判断车辆撞击函数:只有下面的四种组合可能发生撞击,对于每种组合中,分别计算每辆车到达碰撞点（路径交汇中心）的时间，如时间差小于 1s 则判定为碰撞。返回值为 1 说明撞击，否则不撞击。

```
def judge_crash(cars):
    crash = 0
    # car_1 与 car_2
    t_1 = (cars[0].d+15)/cars[0].v_new
    t_2 = (cars[1].d+5 )/cars[1].v_new
    if(abs(t_1-t_2)<1):
        crash = 1
    # car_1 与 car_4
    t_1 = (cars[0].d+5 )/cars[0].v_new
    t_4 = (cars[3].d+15)/cars[3].v_new
    if(abs(t_1-t_4)<1):
        crash = 1
    # car_3 与 car_2
    t_3 = (cars[2].d+5 )/cars[2].v_new
    t_2 = (cars[1].d+15)/cars[1].v_new
    if(abs(t_3-t_2)<1):
        crash = 1
    # car_3 与 car_4
```

```
    t_3 = (cars[2].d+15)/cars[2].v_new
    t_4 = (cars[3].d+5 )/cars[3].v_new
    if(abs(t_3-t_4)<1):
        crash = 1

    return crash
```

6.3 找出最短通行时间并迭代 10 次

主函数中，先调用 6.2 中函数，然后搜索博弈损失矩阵中的最短通行时间，并找出对应玩家 1 和玩家 2 的策略，与传统方式进行比较。按照实验 2 要求，要重复 10 次。最终结果打印显示。

```
if __name__ == "__main__":
    for i in range(10):
        print("*****************",i,"********************")
        actions,loss,time_old = traffic_game()

        # 找出最小损失决策组合
        min_loss = float('inf')
        player1_action = player2_action = []
        print("损失矩阵：")
        for i in range(len(loss)):
            for j in range(len(loss[i])):
                if min_loss > loss[i][j]:
                    min_loss = loss[i][j]
                    player1_action = actions[i]
                    player2_action = actions[j]
                print("%6.2f"%loss[i][j],end=' ')
            print('')

        print("博弈方法：min = %6.2f \nplayer1 决策：
"%min_loss,player1_action," player2 决策：",player2_action)
        print("\n 传统方法：min_old = %6.2f"%time_old)
```

七、结果展示与分析

7.1 实验 1

指定四辆车数据遍历所有的决策组合，计算出博弈损失矩阵(通行时间)，

并给出最快通过时的玩家的决策：

```
car_1:(45.00,23.00) cross_time: 0
car_2:(71.00,24.00) cross_time: 0
car_3:(47.00,28.00) cross_time: 0
car_4:(76.00,28.00) cross_time: 0
损失矩阵:
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf   9.12    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
  6.95    inf    inf    inf    inf    inf    inf    inf    inf
  8.05   8.45    inf    inf    inf    inf   9.66  10.06    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
  8.20    inf    inf   8.69    inf    inf    inf    inf    inf
  9.31   9.71  10.65   9.79  10.19  11.13    inf    inf    inf
博弈方法: min =   6.95
player1决策:  [0, 0]  player2决策:  [1, 1]
```

上述结果可以看出当玩家一决策为[0,0]、玩家二决策为[1,1]时，博弈方法同行时间最短，为 6.95。损失矩阵也如上图所示。

7.2 实验 2

每次实验中，随机生成 4 辆车的数据，计算该数据在传统方法下的通行时间，再计算 1 中的矩阵，以矩阵中的最小值为博弈方法的通行时间。 进行 10 次实验，分别记录 10 次实验中博弈方法和传统方法的结果。

十次随机结果如下（具体数据太多，故只展示了后两次）：

次数	传统方法	博弈方法
0	16.93	7.11
1	12.70	6.34
2	16.00	8.91
3	15.84	7.27

4	17.42	7.05
5	14.90	8.22
6	15.99	6.36
7	16.67	7.34
8	15.77	8.27
9	15.15	8.13

```
****************** 8 ********************
car_1:(41.00,37.00) cross_time: 0
car_2:(43.00,21.00) cross_time: 0
car_3:(80.00,32.00) cross_time: 0
car_4:(41.00,22.00) cross_time: 0
损失矩阵:
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf   8.27    inf    inf    inf    inf    inf    inf
   inf    inf   9.41    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
  9.21    inf    inf   9.50    inf    inf  10.18    inf    inf
博弈方法: min =   8.27
player1决策:  [0, 0]  player2决策:  [1, -1]

传统方法: min_old =  15.77
****************** 9 ********************
car_1:(87.00,39.00) cross_time: 0
car_2:(44.00,32.00) cross_time: 0
car_3:(28.00,31.00) cross_time: 0
car_4:(24.00,23.00) cross_time: 0
损失矩阵:
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf    inf
   inf    inf    inf    inf    inf    inf   8.13    inf    inf
   inf    inf    inf    inf    inf    inf    inf    inf   8.60
   inf    inf    inf    inf    inf    inf    inf    inf   9.02
   inf    inf    inf    inf    inf    inf   8.93    inf    inf
博弈方法: min =   8.13
player1决策:  [0, -1]  player2决策:  [-1, 1]

传统方法: min_old =  15.15
```

可以看出虽然大部分决策中时撞击的情况，但最终博弈方法的结果比传统方法好了两倍左右，而且十次实验均如此。可见博弈方法比传统方法大大改善了车辆的通行时间。但不足的时，这种遍历所有决策的方法需要的计算量和存储量均很大，效率很低，后续可通过剪枝等方法进行优化。

八、心得体会

这次实验主要是特定场景下博弈方法的应用，虽然具体的算法流程和相关公式老师均已给出，但是在编程实现上我还是遇到了不少的困难，尤其是对 Python 中类的使用很不熟悉导致了一些 bug，解决时耗费了一些时间。但这样在一定程度上提升了自己的编程水平。另外，对于这个实验结果，虽然博弈方法有很大计算量和存储量上的弊端，但他的优化效果还是很好的，值得继续学习，考虑推广到别的领域。