Decentralized Greedy Strategy for Large-Scale Traffic Signal Control

Yin Gu[*]
Yuqiang Zhou[*]
1205085045@qq.com
zyq1998@mail.ustc.edu.cn
University of Science and Technology of China
Hefei, Anhui, China

Yanqing An
University of Science and Technology of China
Hefei, China
anyq@mail.ustc.edu.cn

Linan Yue
University of Science and Technology of China
Hefei, China
lnyue@mail.ustc.edu.cn

ABSTRACT

Traffic jams plague people all over the world. How to control traffic lights to reduce traffic congestion is a difficult problem. Especially when the scale of the problem is extended to the city level, where thousands of signal lights are involved, the problem will be challenging. In this competition, we were provided with a city-sized map network, and we needed to control thousands of signal lights at the same time to optimize the vehicle delay index to serve more vehicles. We adopted a decentralized greedy strategy to control actions of signal lights in the intersections. Our method is effective and easy to implement. The results on competition environment indicates the effectiveness of our method, and our team ranks 8 in the final phrase.

CCS CONCEPTS

• **Applied computing → Decision analysis**.

KEYWORDS

traffic signal control

ACM Reference Format:
Yin Gu, Yuqiang Zhou, Yanqing An, and Linan Yue. 2021. Decentralized Greedy Strategy for Large-Scale Traffic Signal Control. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '21), August 14–18, 2021, Virtual Event, Singapore.* ACM, New York, NY, USA, 4 pages.

1 INTRODUCTION

Traffic jams plague people all over the world. How to control traffic lights to reduce traffic congestion is a difficult problem. Especially when the scale of the problem is extended to the city level, where thousands of signal lights are involved. To the best of our knowledge, the existing models for the signal lights control can be grouped into two categories: modeling global traffic flow of the whole map and modeling local traffic flow of single intersection[1, 2]. Considering The scale of this problem is large, we choose to design our solution from the latter perspective of modeling the local traffic flow.

In practical, there are some rules in the traffic flow due to the regularity of people's daily life. So we analyzed the traffic flow in the competition environment (simulator) in order to get some traffic rules. It is worth noting that the environments(simulator) for the qualification phrase and the final phrase are somewhat different. In the following sections, unless otherwise specified, the default environment is the simulator of final phrase. Meanwhile, agent refers to intersections with traffic lights, and non-agent refers to intersections without traffic lights.

After constant testing, we have discovered some stable rules of the traffic flow and the environment. The understanding of these following rules is very important to the implement of our algorithm:

1 The route of a vehicle is fixed, and we don't need to implement a policy to control the route of the vehicle.
2 A flow generates a car every fixed time, and the routes of these cars are the same.
3 The route and full trip travel time at free-flow speed of the vehicles are not provided, but in the qualifying competition, these two are provided.
4 A road has two opposite directions, therefore, a road has two road segments.
5 We observe the road net data and find that each road segment has three lanes, namely lane 00, lane 01, and lane 02. Lane 00 is only allowed to turn left, lane 01 is only allowed to go straight, and lane 02 is only allowed to turn right.
6 When a car is driving to an agent, if the car is going to turn left next, he will stay in lane 00; if the car is going straight next, he will stay in lane 01; if the car is going to turn right next, he will stay in lane 02.
7 When a car is driving towards a non-agent, the situation is more complicated. The car will not follow the above rule 6. It has nothing to do with which the direction it will turn.
8 An agent can perform 8 actions, and one action corresponds to 2 non-conflicting movements. We checked the data and found that the agents are all three-way intersections or four-way intersections. In other words, an agent has at least 3 incident road segments, and an agent has at least 3 exit road segments. Certain movements of a part of the agent are invalid.

[*]Both authors contributed equally to this research.

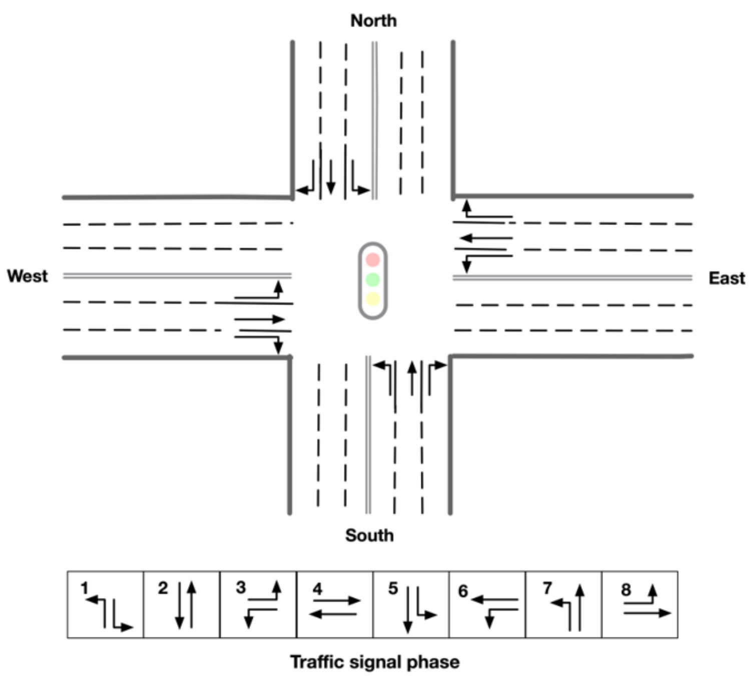

Figure 1: Overview of intersection

9 We cannot control the non-agent, and the non-agent cannot block any vehicles. Even if the movements of two non-agent vehicles are in conflict, they can still pass through the non-agent at the same time.
10 In the final stage, regardless of whether a car is driving to an agent or a non-agent, if its downstream road segment is blocked, then it will not be able to pass the intersection.
11 When a car is generated on the map, it will appear in lane 00 of the road segment. Then move to the corresponding lane according to his next route.
12 The speed when the car is generated is 0, and the acceleration of the car is 2.

Besides, the delay index is calculated as following:

$$Delay_i = \frac{TT_i + TT_i^T}{Tff_i}, \tag{1}$$

$$Delay_{all} = \sum_{i=1}^{N} Delay_i, \tag{2}$$

where $Delay_i$ and $Delay_{all}$ are the delay index of car i and all cars respectively, TT_i is the time car i have been ran, TT_i^T is the left time of car i to finish its route at free-flow speed and Tff_i is the full trip travel time of car i finishing its route at free-flow speed. So the importance of specific car i is related to the value of Tff_i, which could be calculated with car i's route information. Along this line, we proposed a purely tree-based algorithm to predict the car's routes and adopted a Decentralized Greedy Strategy (DGS) to control actions of signal lights in the intersections. Finally, the results on competition test platform indicates the effectiveness of our method.

2 METHOD

In our method, all agents are independent of each other, and we do not consider the interaction between agents. In each step, we first traverse all agents, take an action according to their input road segments and output road segments, and then return actions of all agents to the environment.

As mentioned before, an agent has 8 actions, and each action corresponds to 2 non-conflicting movements (one movement corresponds to one input lane). Similarly, we treat each movement of agent dependently. First, we calculate a weight for each movement, then calculate the weight of each action (the sum of the weights of the two movements), and finally choose the action with the largest weight.

The calculation of the movement weight adheres to the following principles:

1 The current action has a greater weight;
2 If there is no car in an input lane, the weight of the corresponding movement is 0;
3 If a downstream lane of a movement is blocked (in this case, the car cannot drive to the downstream lane), then its corresponding movement weight is 0;
4 For a three-way intersection, some of its movements are invalid, and the corresponding weight is 0.

In the following section, we will introduce the process of predicting the car's route and the details of action weight calculation based on our DGS method, respectively.

2.1 Tff Time Prediction

According to the evaluation metrics of the competition, we know that the car with the smaller Tff has the greater impact on the delay index. Therefore, the shorter route has the greater priority. However, in the final, the route of the vehicle is unknown, and we need to predict the next road of the vehicle.

After analyzing the flow, we find that cars with the same birth road segment are supposed to have the same route. So, it's easy to get the idea of storing the cars' routes and infer the route and Tff time of cars by their current tracks. Generally, we build a tree for all cars' routes where a leaf refers to a full route and the road lanes in a route are the nodes in different depth. And we also store the car's track in a python dict. For each car, we use its track to search in the route tree and get the latest route as the predicted route. If we cannot find a route that contains the car's track, we update the tree's nodes and give a default value for the car's Tff time (e.g., 700).

To be specific, we first store the appearing cars, the disappearing cars, the running cars in each step with the help of the observations. Then for disappearing cars with a new route, we set their last road lane nodes in the tree as leaf nodes and update the nodes in the route with the new route time. For appearing cars and running cars, we update their track python dict with the current road lane and search their route in the tree based on their track.

If we get more than one route containing the same track, we choose the latest route as the car's route. If we could not get a route containing their track, we also assign the car with a default Tff value of 700 and update the track to the tree.

2.2 Greedy Action Control

We design a greedy strategy to get the best action for each single agent. It is worth noting that we only take the roads directly concerned to the agent into consideration for each agent. So it is a greedy strategy for the local traffic flow and not considering the global flow pressure. Generally, for each agent, we first calculate the weight for each input lane connected to the agent. Then, we get the weights for each action based on the weights of action-related lane. Finally, considering the action change would lead to 5 seconds delay, we encourage the current on-going action by a weight larger than 1(e.g., 1.6). To be specific, for each lane l, we define the weights R_l as following:

$$R_l = \sum_{i=1}^{m} \frac{t_i}{Tff_i}, l \in Roadlane, \tag{3}$$

where m is the num of cars on the lane, i is the i-th car on the lane and t_i is the left time after car i arrive the agent. It is worth noting that the time used to approaching the agent does no matter with the action of agent.

In practice, we calculate t_i as :

$$t_i = MAX(\frac{D_i^{step} - D_i^{agent}}{\overline{v_i}}, 0), \tag{4}$$

where D_i^{step}, D_i^{agent}, $\overline{v_i}$ are the distance car i can run in one step, car i's distance to agent and car i's average speed after passing the agent, respectively. And we model the car's motion as an uniformly accelerated motion. So D_i^{step} is defined as:

$$D_i^{step} i = MAX(\frac{v_i^{start} - v_i^{end}}{2} * t_{acc} + v_i^{end} * (t_{step} - t_{acc}), \tag{5}$$

$$v_i^{end} \leq v_{limit}^{l}, \tag{6}$$

$$t_{acc} \leq t_{step}, \tag{7}$$

where v_i^{start}, v_i^{end}, t_{acc} t_{step},v_{limit}^{l} are the original speed of i, the final speed of i, the accelerated time of i, the time of one step (e.g., 10 seconds) and the lane l's limit speed respectively. Then for each action, the weight is calculated as:

$$R_{action}^0 = \sum_{j=1}^{n} R_{l_j} * valid_{l_j}, \tag{8}$$

where n is the num of lanes for a agent(i.e., 8), j is the j-th lane of the agent, R_{l_j} is the weight of lane l_j, $valid_{l_j}$ is whether lane l_j exists. Finally, the same action with last action will be encouraged as:

$$R_{action} = \begin{cases} k * R_{action}^0, & keepphrase \\ R_{action}^0, & else \end{cases}, \tag{9}$$

Algorithm 1 Get Car Tff Time

```
Input: $Obs, RouteTree$
Output: $CarTff$
1: function CarInfoUpdateStep($Obs_{step}, RouteTree$)
2:     $CurrentCarSet, LastCarSet, CarTrackSet, CarRoadLane \leftarrow Obs_{step}$
3:     $FinishCarSet \leftarrow LastCarSet - CurrentCarSet$
4:     $AppearCarSet \leftarrow CurrentCarSet - LastCarSet$
5:     $RunningCarSet \leftarrow CurrentCarSet - AppearCarSet$
6:     for $car$ in $FinishCarSet$ do
7:         $CarTrackSet[car]$.update($CarRoadLane[car]$)
8:         $Node \leftarrow RouteTree$.search($CarTrackSet[car]$)
9:         if $Node$ is not Leaf then
10:            $RouteTree$.SetLeaf($Node$)
11:        end if
12:    end for
13:    for $car$ in $AppearCarSet$+$RunningCarSet$ do
14:        $CarTrackSet[car]$.update($CarRoadLane[car]$)
15:        $Node \leftarrow RouteTree$.Search($CarTrackSet[car]$)
16:        if $Node.HaveFullRoute$ then
17:            $CarTff[car] \leftarrow Node.LastestTff$
18:        else
19:            $RouteTree$.AddChildren($Node$,$CarRoadLane[car]$)
20:            $CarTff[car] \leftarrow 700.0$
21:        end if
22:    end forreturn $CarTff$
23: end function
24:
25: function RouteTreeUpdate($Obs$,$RouteTree$)
26:    for $each$ $step$ do
27:        $Obs_{step} \leftarrow Obs$
28:        $CarTff \leftarrow$CarInfoUpdateStep($Obs_{step}$,$RouteTree$)
29:    end forreturn $CarTff$
30: end function
```

Competition phrase	Total Served Vehicle	Delay	Team rank
Qualification	126572	1.517	17
Final	316718	1.403	8

Table 1: Result

where k is the award for action of keeping phrase (e.g., 1.6), R_{action} is the final action weight.

2.3 Results

The results of our team is displayed in Table1. It is worth noting that the environments in qualification phrase and final phrase are different. And we didn't use the tree-based algorithm for Tff prediction in qualification phrase.

3 CONCLUSION

In this report, we introduced solution of our team *alphabeta* to the city brain challenge task. Generally, we proposed a purely rule-based algorithm to predict the car's routes and adopted a decentralized greedy strategy to control actions of signal lights in the intersections. Finally, our team ranks 8 in the final phrase of competition.

REFERENCES

[1] Chacha Chen, Hua Wei, Nan Xu, Guanjie Zheng, Ming Yang, Yuanhao Xiong, Kai Xu, and Zhenhui Li. 2020. Toward a thousand lights: Decentralized deep reinforcement learning for large-scale traffic signal control. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 3414–3421.

[2] Guanjie Zheng, Yuanhao Xiong, Xinshi Zang, Jie Feng, Hua Wei, Huichu Zhang, Yong Li, Kai Xu, and Zhenhui Li. 2019. Learning phase competition for traffic signal control. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 1963–1972.