Final Tracking Report

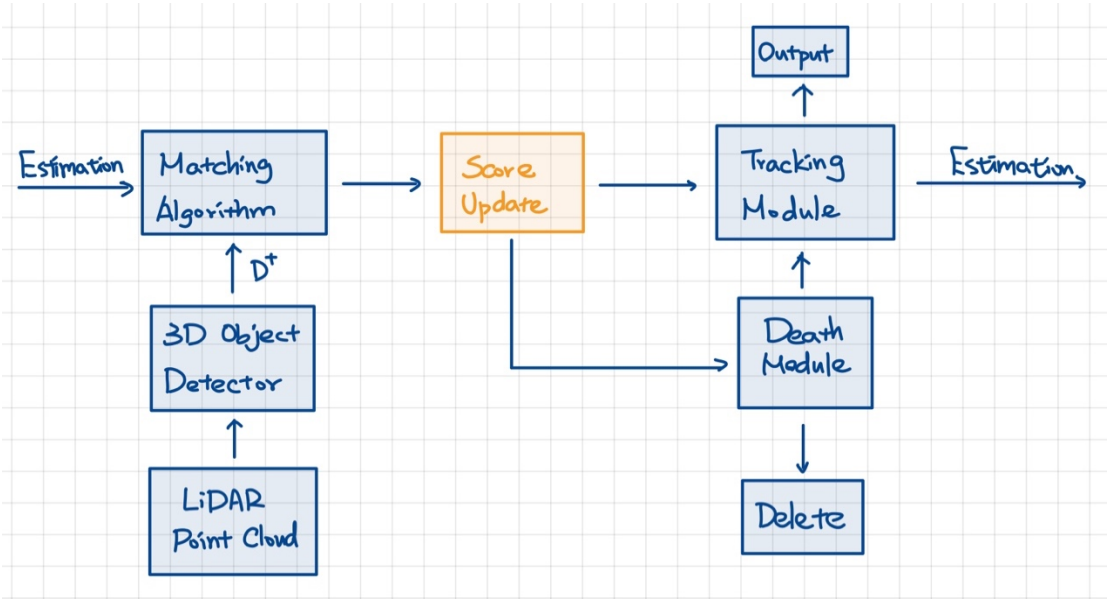
[EECN30168] Self-Driving Cars 2022

Team 1　　Member: 311605004 劉子齊

1. Introduction

For the final tracking competition, our goal is to design a brilliant tracking algorithm based on the tracking detection method and the given detected bounding boxes from the detector. In this project, I mainly focus on the design of the greedy algorithm and the tuning of the NUSCENE_CLS_VELOCITY_ERROR, aiming to obtain improvement in the tracking results through the implementation explained below.

2. Pipeline



3. Contributions

3-1. Algorithm

The main goal in designing the algorithm is to find the index of the paired object for each tracked object through the distance information between them. Therefore, the methodology we filter the most likely

correct index is the greatest challenge in this part.

Initially, I tried out the Hungarian Algorithm, which came with the original package of this project provided by the TA. Through the Hungarian Algorithm, I obtain the result of 0.616 for the AMOTA, which is way lower than the baseline.

Then I started to think about what should my greedy algorithm do to result in a considerably good result. Then I found that most of the algorithms found online will encounter the problem of which indices might be repeatedly paired in the process.

Hence, for the greedy algorithm designed, I will make the paired indices into a relatively immense value during the paring process, which was set to 1e18 here. Through the current implemented greedy algorithm, I successfully reached the same AMOTA & AMOTP result as the baseline, which is 0.680 & 0.530.

In the algorithms mentioned above, I think this algorithm feels a bit too brute force. Therefore, I also tried dynamic programming. The original method pairs and filters object from the first to the nth object sequentially. My approach reads in all objects and prioritizes the minimum from all pairs for pairing. In other words, my new approach sorts by the longest distance and pairs in order.

Interestingly, through this method, the AMOTA result is 0.672, which is indeed better than the result of using the Hungarian algorithm, but still lower than the baseline. The AMOTP is even higher at 0.798. After much thought, I believe that the cause of this result is that, through visualizing the tracking results, I found that in some scenes, there are many objects around the car that pass by the car. If only the nearest object is considered, the number of mistaken pairings will gradually accumulate over time.

Overall, compared to the Hungarian algorithm, which uses linear analysis for object pairing, the Greedy Algorithm uses a non-linear calculation method. Therefore, the Greedy Algorithm can avoid the situation where the results are not as good as the baseline due to the

fewer factors considered in linear analysis. The implementation of the final Greedy Algorithm is shown in the figure below.

```
    if dist.shape[1] == 0:
        return np.array(matched_indices, np.int32).reshape(-1, 2)

    # print(dist)

    # for i in range(dist.shape[0]):
    #     print(dist[i])

    for i in range(dist.shape[0]):
        j = dist[i].argmin()
        if dist[i][j] < 1e16:
            dist[:, j] = 1e18
            matched_indices.append([i, j])

    # raise NotImplementedError("Greedy algorithm not implemented yet!")
    ### Student implement ###
    return np.array(matched_indices, np.int32).reshape(-1, 2)
```

3-2. Tuning of the Velocity Error

As we can see in the figure below, before the program conducts the greedy algorithm, it will exclude the cases that the distance is greater than the threshold, which was defined as the max_diff by the dictionary called the NUSCENE_CLS_VELOCITY_ERROR. If the input doesn't meet the conditions above, it will all be added to 1e18 and filtered out.

```
    max_diff = np.array([self.velocity_error[box['detection_name']] for box in positions2_data], np.float32)

    if len(positions1) > 0:  # NOT FIRST FRAME
        dist = (((positions1.reshape(1, -1, 2) - positions2.reshape(-1, 1, 2)) ** 2).sum(axis=2))  # N x M
        dist = np.sqrt(dist)  # absolute distance in meter
        invalid = ((dist > max_diff.reshape(N, 1)) + (
                positions2_cat.reshape(N, 1) != positions1_cat.reshape(1, M))) > 0
        dist = dist + invalid * 1e18
```

As previously mentioned, the max_diff threshold comes from the NUSCENE_CLS_VELOCITY_ERROR dictionary, which contains the different pass thresholds of each object. During the adjustment process, as shown in the figure below, we can obtain the evaluating result of each different object at the end of the evaluation. Through the individual AMOTA and AMOTP of each object, I can adjust the speed error of the 7 objects at the same time.

```
### Final results ###

Per-class results:
             AMOTA   AMOTP   RECALL  MOTAR   GT      MOTA    MOTP    MT      ML      FAF     TP      FP      FN      IDS     FRAG    TID     LGD
bicycle      0.487   0.451   0.519   0.787   1993    0.408   0.194   60      81      14.6    1034    220     959     0       0       0.46    0.65
bus          0.872   0.501   0.836   0.940   2112    0.785   0.415   74      18      6.7     1763    105     347     2       24      0.27    0.54
car          0.848   0.342   0.850   0.861   58317   0.728   0.260   2766    718     118.8   49327   6847    8775    215     143     0.22    0.38
motorcy      0.666   0.468   0.681   0.865   1977    0.588   0.271   74      44      12.9    1343    181     631     3       6       0.54    0.82
pedestr      0.765   0.370   0.799   0.799   25423   0.632   0.274   1155    445     91.2    20086   4029    5105    232     97      0.20    0.53
trailer      0.455   0.995   0.536   0.718   2425    0.383   0.580   58      63      34.9    1294    365     1126    5       15      0.49    1.20
truck        0.696   0.601   0.728   0.772   9650    0.560   0.373   334     171     42.4    7001    1593    2628    21      28      0.37    0.70
```

The following figure on the left is the initial velocity error dictionary, and the figure on the following right is the final velocity error dictionary after tuning.

```python
NUSCENE_CLS_VELOCITY_ERROR = {
    'car': 3,
    'truck': 4,
    'bus': 5.5,
    'trailer': 2,
    'pedestrian': 1,
    'motorcycle': 4,
    'bicycle': 2.5,
    'construction_vehicle': 1,
    'barrier': 1,
    'traffic_cone': 1,
}
```

```python
NUSCENE_CLS_VELOCITY_ERROR = {
    'car': 2.125,
    'truck': 2.125,
    'bus': 6.4,
    'trailer': 5,
    'pedestrian': 2,
    'motorcycle': 3.875,
    'bicycle': 2,
    'construction_vehicle': 1,
    'barrier': 1,
    'traffic_cone': 1,
}
```

In the process of adjusting, I found that cars, trucks, and bicycles can have more significant improvement in parameter adjustment, among which trucks and bicycles have the most significant improvement, both of which have improved the accuracy by 0.007. Despite this, the overall accuracy is still very limited in terms of improvement.

In my opinion, the reason for this is that among the many objects, several objects perform poorly in tracking. For example, even after adjusting, the AMOTA for bicycles is still only 0.487, and the AMOTA for Trailers is only 0.455 after adjusting. I believe these two types of objects are the main reason for the overall AMOTA performance. Perhaps these two types of objects are naturally more difficult to track in the nuScenes environment, but if we can improve the tracking results for these two categories, the overall tracking performance will surely improve significantly.

4. Problem Encountered & Solutions

At the beginning of this project, I encountered trouble displaying the point cloud and the tracking bounding box on the rviz display. Even if I applied the small dataset, I couldn't depict the rviz output. However, I

eventually solved this problem by opening the rviz outside the docker. I assume that the resource allocated to the docker by my computer wasn't capable enough to visualize the tracking result and evaluate it simultaneously.

5. Discussion

5-1. Point Tracker vs. Kalman Filter

During the experiment, I discovered a parameter called the Kalman filter, which can switch the tracking method from the default point tracker to the Kalman filter. However, due to time constraints and the fact that the results obtained from initial testing showed some degree of deviation from my fine-tuned point tracker, I did not use the Kalman filter for tracking in this project.

Despite this, I believe that the Kalman filter still has the potential to achieve better results than the point tracker. Since the point tracker tracks objects by predicting their next position based on the current speed over time, which is a more linear prediction method. On the other hand, the Kalman filter is a nonlinear prediction based on a Gaussian distribution. I believe nonlinear prediction theory is theoretically more closely aligned with the actual movement of objects on the road. Therefore, I think the Kalman filter has the potential to achieve better results than the original point tracker after fine-tuning.

5-2. Other Fancy Detection Methods

While observing the visualized results, I noticed that sometimes there are already errors in the model when object detection, such as misidentifying a bicycle as a motorcycle or a bus as a trailer. If we can output more accurate results when doing object detection, is it possible for me to obtain better tracking results by utilizing these more reliable detection results? I think this is an area where I can continue to experiment in the future.

6. Several Captures of the Visualized Tracking Result