Writeup for Path Planning Project

Part of Udacity self driving car nanodegree

Sven Eriksson

January 16, 2018

This writeup intends to explain what states I am using in my solution, the transition between them and how they generate paths.

1 States and transitions

In my path planning solution I am esentially using two state. Lane following and lane change. The lane following state can be split into two parts, an initial state and the main lane following state. The difference is that the inital state does not have transitions to a lane change state.

The lane change states have been implemented as a change lane left and a change lane right state by the use of a parameter describing direction. They are otherwise identical. See figure 1 for a graphical representation.

In the lane following state multiple paths are generated, the main difference between them is the intended final speed. These paths are then evaluated by a fitness function that accounts for speed and acceleration limits as well as the other vehicles. These paths are regenerated and reevaluatet each time the path planner (my code) receives updates from the simulator.

The change lane state generate a single path during upon entry into the state. This is then given to the simulator. It is not replanned based on new information from the simulator.

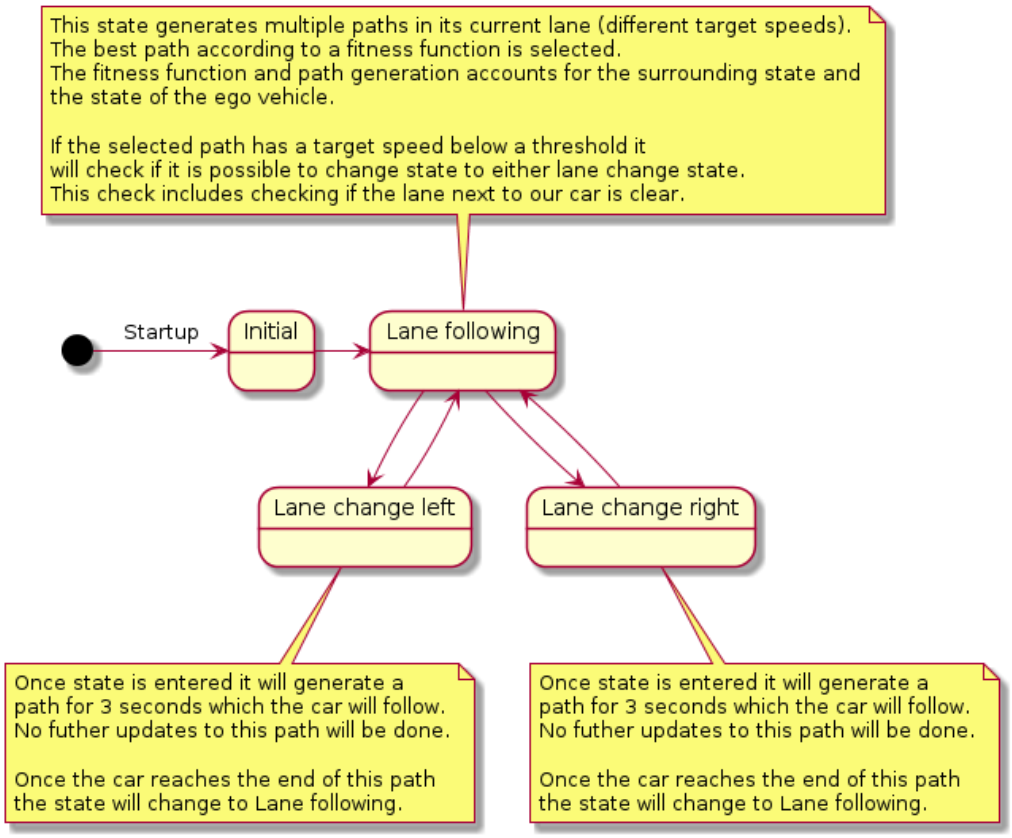

Figure 1: The basic state machine in my solution.

1.1 Transition from initial

Once the speed of the vehicle reaches a certain threshold (30mph) for the first time the state will transition to lane following.

1.2 Transition from lane following

In the case that the best path has an intended final speed of 43mph or lower and the current speed is 43mph or lower a change lane path will be generated and evaluated by the fitness function.

If this new change lane path does not violate the limits on velocity and acceleration as well as not collide with any other current vehicle positions a second check will be done. This second check makes sure that the reason for the current lower speed is due to another vehicle in the front of ours and

that the lane we a evaluating a transition into is empty for a certain interval around our vehicle in S.

If this second check also succeeds the state will transition to the lane change state. These checks are done for transitions into both lane change left and lane change right independently. Lane change left has priority if both are possible.

1.3 Transition from lane changing

Once the path generated upon entry has too few points left the state will transition back into lane following. The threshold is 25 points (0.5 seconds).

2 Path generation in states

The general idea for the two states have been described in the previous section. This section intends to explain the path generation in more detail.

2.1 Follow lane

The idea here was to create two splines. One X(t) and one Y(t). First I add a few points to X(t=0.02), ... , X(t=0.2) by using the previously planned points. I add the same interval to Y(t=0.02), ... ,Y(t=0.2). I do an estimation about the S, velocity and D position at t=0.2 based on the previously planned points.

Based on an input parameter finalV I calculate a value of S at two points.

$$S(t = 2) = S(t = 0.2) + (2 - 0.2) * (v(t = 0.2) + finalV)/2$$
$$S(t = 2.5) = S(t = 2) + 0.5 * finalV$$

These together with the D for the center of the current lane is then transformed back into X and Y. My two set of points does not contain, X(t=0.02), ... , X(t=0.2), X(t=2) and X(t=2.5). Same time points for Y.

I do now use a spline library to calcute the path for the next 1.5s. I pick points with an interval of 0.02s.

2.2 Change lane

The change lane path generation works on more or less the same idea as the follow lane path generation.

The difference is that D at t=2 and t=2.5 is that of the lane that is going to be changed into and that finalV = v(t=0.2).

A Code for path generation

Lane following state path generation

```
vector<vector<double>> StayInLane::possiblePath2(vehicleAndMapState &state, double finalV_mph) {
  double extra_t = 0.5;
  double path_t = 2;
  double max_t = 1.5;

  vector<double> X, Y, T;

  vector<double>* previousX = state.previous_path_x;
  vector<double>* previousY = state.previous_path_y;

  int keepNumberOfSteps = 10;

  double t = 0.02;
  double temp_X, temp_Y;
  double initialSpeed;
  double S, D;

  if(previousX->size() > keepNumberOfSteps * 1.5){
    for(int i=0; i<keepNumberOfSteps;i++){
      temp_X = previousX->at(i);
      temp_Y = previousY->at(i);
      X.push_back(temp_X);
      Y.push_back(temp_Y);
      T.push_back(t);
      t += 0.02;
    }
    int deltaSteps = 2;
    dynamics pastPath = dynamicsEstimator(*previousX, *previousY, deltaSteps);

    double S_pos, D_pos;
    vector<double> SD = getFrenet(pastPath.pos.at(keepNumberOfSteps/deltaSteps).X, pastPath.pos.at(keepNumberOfSteps/deltaSteps).Y, deg2rad(state.car_yaw), *state.map_waypoints_x, *state.map_waypoints_y);
```

```
32
33        S = SD.at(0);
34        initialSpeed = pastPath.vel_abs.at(
              ↪ keepNumberOfSteps/deltaSteps);
35      }else{
36        X.push_back(state.car_x);
37        Y.push_back(state.car_y);
38        T.push_back(0);
39
40        S = state.car_s;
41        initialSpeed = state.car_speed;
42      }
43
44      double finalSpeed = mph2mps(finalV_mph);
45
46
47      D = middleOfLane(laneNumber(state.car_d));
48      S = S + (path_t - t) * (initialSpeed + finalSpeed)/2;
49
50      vector<double> XY = getXY(S, D, *state.
            ↪ map_waypoints_s, *state.map_waypoints_x, *state
            ↪ .map_waypoints_y);
51      temp_X = XY.at(0);
52      temp_Y = XY.at(1);
53      X.push_back(temp_X);
54      Y.push_back(temp_Y);
55      T.push_back(path_t);
56
57      S = S + extra_t * finalSpeed;
58
59      XY = getXY(S, D, *state.map_waypoints_s, *state.
            ↪ map_waypoints_x, *state.map_waypoints_y);
60      temp_X = XY.at(0);
61      temp_Y = XY.at(1);
62      X.push_back(temp_X);
63      Y.push_back(temp_Y);
64      T.push_back(path_t + extra_t);
65
66
67      tk::spline splineX;
68      tk::spline splineY;
```

```
69    splineX.set_points(T, X);
70    splineY.set_points(T, Y);
71
72    vector<double> finalX, finalY;
73
74    for(double t=0.02; t<max_t; t+=0.02) {
75      finalX.push_back(splineX(t));
76      finalY.push_back(splineY(t));
77    }
78
79    return {finalX, finalY};
80  }
```

Lane change state path generation

```
1   vector<vector<double>> ChangeLane::possiblePath(
      ↪ vehicleAndMapState &state, double finalV_mph){
2
3     vector<double>* previousX = state.previous_path_x;
4     vector<double>* previousY = state.previous_path_y;
5
6     double t=0.02;
7     double max_t = 2;
8     double extra_t = 1;
9     double delta_t = 1;
10
11    double tempX, tempY;
12
13    double initialS, initialV, initialD, finalS, finalV,
        ↪ finalD, yaw_radian;
14
15    initialS = state.car_s;
16    initialV = state.car_speed;
17    initialD = state.car_d;
18
19    finalV = mph2mps(finalV_mph);
20    initialV = mph2mps(state.car_speed);
21
22    finalD = middleOfLane(laneNumber(state.car_d) +
        ↪ direction);
23
```

```
24    if(finalD > 13 || finalD < 1){
25      finalD = middleOfLane(laneNumber(state.car_d));
26    }
27
28    finalS = initialS + max_t*(initialV + finalV)/2;
29
30    yaw_radian = deg2rad(state.car_yaw);
31
32
33    vector<double> S, D, T;
34    vector<double> X, Y;
35    vector<vector<double>> temp;
36
37    X.clear();
38    Y.clear();
39
40    double temp_X;
41    double temp_Y;
42    double speedAfterPrevious;
43    double previousS;
44    double previousD;
45
46    if(previousX->size()>2){
47      for(int i=0; i<previousX->size() && i<25; i++){
48        temp_X = previousX->at(i);
49        temp_Y = previousY->at(i);
50        X.push_back(temp_X);
51        Y.push_back(temp_Y);
52        T.push_back(t);
53        t+=0.02;
54      }
55
56      //TODO: This does probably cause some speed
          ↪ fluctiation.
57      if(X.size()>11){
58        speedAfterPrevious = sqrt(pow(X.at(X.size() - 1)
            ↪ - X.at(X.size() - 11), 2) +
59                                  pow(Y.at(Y.size() - 1)
                                      ↪ - Y.at(Y.size() -
                                      ↪  11), 2)) / 0.2;
60      }else {
```

```
61          speedAfterPrevious = sqrt(pow(X.at(X.size() - 1)
              ↪ - X.at(X.size() - 2), 2) +
62                                    pow(Y.at(Y.size() - 1)
                                        ↪ - Y.at(Y.size() -
                                        ↪  2), 2)) / 0.02;
63        }
64        vector<double> sd = getFrenet(X.back(), Y.back(),
            ↪ state.car_yaw, *state.map_waypoints_x, *state
            ↪ .map_waypoints_y);
65        previousS = sd.at(0);
66        previousD = sd.at(1);
67
68      }else{
69        S.push_back(initialS);
70        D.push_back(initialD);
71        T.push_back(0);
72        previousS = initialS;
73        previousD = initialD;
74
75        speedAfterPrevious = mph2mps(state.car_speed);
76      }
77
78      S.push_back(previousS + speedAfterPrevious * (max_t-t
            ↪ ));
79      D.push_back(finalD);
80      T.push_back(max_t);
81
82      previousS += speedAfterPrevious * (max_t-t);
83
84      S.push_back(previousS + speedAfterPrevious * extra_t
            ↪ /2);
85      D.push_back(finalD);
86      T.push_back(max_t + extra_t/2);
87
88      S.push_back(previousS + speedAfterPrevious * extra_t)
            ↪ ;
89      D.push_back(finalD);
90      T.push_back(max_t + extra_t);
91
92
93      temp = getXY(S, D, *state.map_waypoints_s, *state.
```

```
        ↪ map_waypoints_x, *state.map_waypoints_y);
94    for(int i=0; i<temp.data()[0].size();i++){
95      tempX = temp.data()[0].at(i);
96      tempY = temp.data()[1].at(i);
97      X.push_back(tempX);
98      Y.push_back(tempY);
99    }
100
101   tk::spline splineX;
102   tk::spline splineY;
103   splineX.set_points(T, X);
104   splineY.set_points(T, Y);
105
106   vector<double> finalX, finalY;
107
108   for(double t_iter=0.02; t_iter<max_t+extra_t; t_iter
        ↪ +=0.02) {
109     finalX.push_back(splineX(t_iter));
110     finalY.push_back(splineY(t_iter));
111   }
112
113   return {finalX, finalY};
114 }
```