Path Planning and Deployment of Mobile Robot Network

Shengdong Liu

December 15, 2015

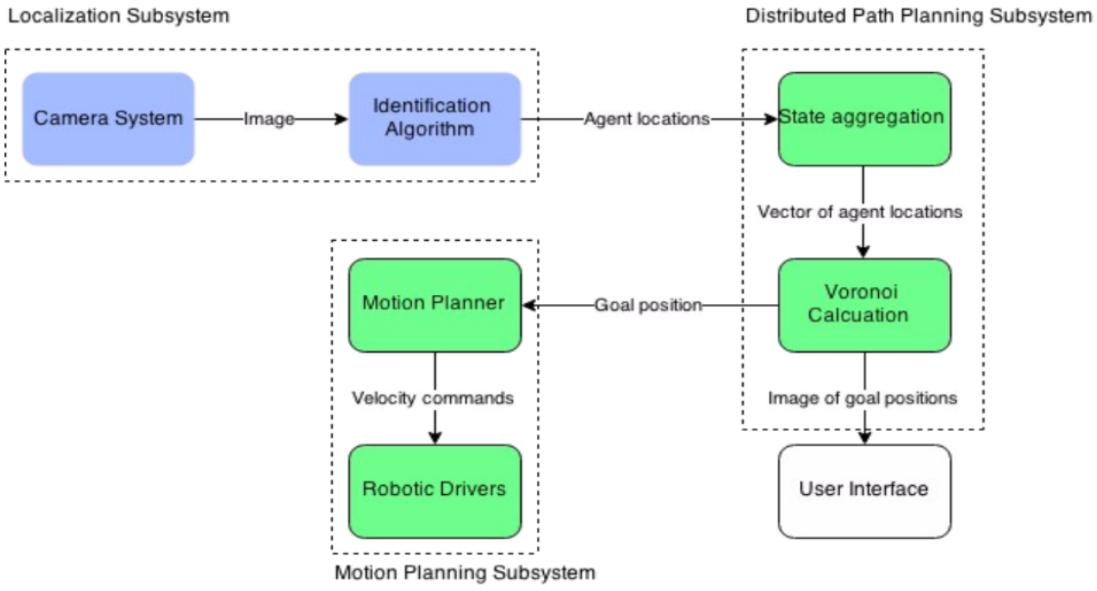
Abstract

This paper further my work from Summer 2015 and focuses on optimizing the motion control of TurtleBots, which are small ground robot consists of a mobile base, 3D sensor, and a laptop computer. The path planning algorithm implemented, Dubin Curve, simulates the a simple car with a constant forward speed and is constrained by a maximum steering angle, which results in a minimum turning radius. The path following algorithm, unicycle model proportional-integral-derivative control keep a vehicle on a path at a constant speed by adjusting the angular velocity. Optimization for the motion was added to further simulate a car like behavior and better synchronize with the multi-agent control algorithm, which prepares the TurtleBots to test inter-robot communication and coordination.

1 Big Picture

As robotics technology advance, we are seeing an increasing the number of autonomous robots in our daily lives. This boom in robots calls for inter-robot communication and coordination to complete takes in groups. The Multi-Agent Robotics Control Lab (MURO) in UCSD, led by Professor Cortes and Martinez, has been incubating a testbed to test potential multi-agent control algorithms for deployment. The lab currently possess ten TurtleBots platforms as a multi-agent robotic network. The open source ROS (Robotic Operating System) is used as the software platform for its publisher-subscriber model to allow for inter-robot communication and coordination. The objective of the ROS TurtleBot project is to provide a stable system to which distributed control system can be readily deployed for testing and validation on hardware.

The current system set up for the testbed consists of three subsystems. The localization subsystem provided the position and orientation of each of deployed TurtleBots through two overhead cameras that send video stream to a laptop as the base station. The distributed path planning system applies multi-agent control algorithms and uses the video steams to calculate the goal position of each TurtleBot. The motions planning subsystem on each TurtleBot then uses the goal position to create a path and send velocity commands to the mobile base for navigation. Since the system is implement in ROS, information can be easily passed on by means of publishers and subscribers between the subsystems. Visualizations of the multi-agent control algorithm are created and displayed the users through the user interface for monitoring the correctness and effectiveness of the multi-agent control algorithm being tested.



2 Personal Contribution

I have been working on the motion planning subsystem. Two main components of the motion planner are path planning and path following. In spring 2015, I worked on integrating Dubins Curve algorithm as the path planning algorithm to simulate a simple car. (See report-Sp15-SLiu for more detail), and my partner Daniel implement an unicycle model proportional-integral-derivative (PID) control algorithm for path following. In summer 2015, I put Dubins Curve algorithm and Unicycle model PID controller together to test on TurtleBots (see report-Su15-SLiu for more detail). Continuing my work from previous quarters, I added some optimizations to make the control a more reliable and more resemble a car.

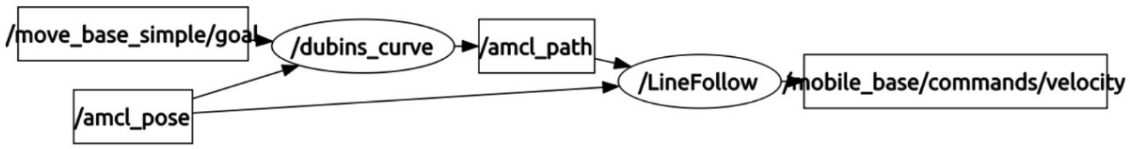
3 Preliminaries

Please refer to report-Sp15-Sliu.pdf to see more on the path planner, report Sp15-dHeideman.pdf on the path follower, and report-Su15-SLiu.pdf for the TurtleBot tests.

4 Methodology

4.1 Goal Orientation Heuristic

The required inputs for Dubins Curve algorithm are two position variablesone variable representing the initial position of the TurthrBot, and one representing the goal position where the Turtle bot wants to be. Both position variable has three fields, (x, y, θ), where x is the x position on the map, y is the y position on the map, and θ is the orientation. In the case of our TurtleBot implementation, for dubins_curve node to run, it subscribes to two topics. The node subscribes to the amcl_pose topic to retrieve the initial position of the TurtleBot, which was published by the overhead camera. In addition, the node subscribes to move_base_simple/goal topic to retrieve the goal position, which was published by the multi-agent control algorithm. One problem with this set up is that the multi-agent control algorithm only generate a (x, y) pair, as the orientation is irrelevant to the algorithm. With this case unhandled, the path generated by the Dubins Curve algorithm will end with the TurtleBot facing $x = -\infty$, since θ of the goal position would be initialized at 0. This generate awkward behavior and the path generated is not optimal unless the only movement the TurtleBot is expected to is to move toward $x = -\infty$, which is not the case.



In order to solve that problem, I used a simple heuristic. Assuming the best orientation is to face where the TurtleBot need to go next, we would need to predict the next goal position, which would be computationally expensive. However, we can make a heuristic prediction based on how the TurtleBot has moved in the past, that is we can use the angle of the line between current position and goal position in this iteration. The slope of this line is defined by:

$$slope = \frac{y_{goal} - y_{curr}}{x_{goal} - x_{curr}} \tag{1}$$

so we can find the angle between the line using:

$$\theta = \arctan\left(\frac{y_{goal} - y_{curr}}{x_{goal} - x_{curr}}\right) \tag{2}$$

However, this is not sufficient, since the range of an arctangent function is from $-\frac{\pi}{2}$ to $\frac{\pi}{2}$ radians. So for the range $\frac{\pi}{2}$ to $-\frac{\pi}{2}$, that is when the $x_{goal} < x_{curr}$ position we need to add an addition π to our answer. So then the function to initialize the goal orientation would be defined by this piecewise function:

$$\theta = \begin{cases} \arctan\left(\frac{y_{goal}-y_{curr}}{x_{goal}-x_{curr}}\right) & \text{if } x_{goal} \geq x_{curr} \\ \arctan\left(\frac{y_{goal}-y_{curr}}{x_{goal}-x_{curr}}\right) + \pi & \text{otherwise.} \end{cases} \tag{3}$$

4.2 Implementation

To implement this functionality, I added the following code to dubins_curve.c, before calculating the Dubins Curve:

```
//automatically assign goal orientation
goal_pose[2] = atan( (goal_pose[1] - cur_pose[1])/
(goal_pose[0] - cur_pose[0]) );

if ( goal_pose[0] - cur_pose[0] < 0 ){
        goal_pose[2] += PI;
}
```

Note: pose[0], pose[1], pose[2] in the implementation above maps to (x, y, θ) respectively.

4.3 Control for Linear Velocity

The unicycle model PID controller was implemented using one constant to control the linear velocity of the TurtleBots. While it is simple to implement, the speed is capped at a speed thats safe for the TurtleBot to make the turn. Imagine a car that can drive only up to 5 miles per hour because it needs to be able to drive through the parking lot. How inefficient! A car navigate through curves slowly to ensure safety, but it is safe to speeds up on a straight lane. For the path generated using Dubins Curve, there will be a line segment most of the time, so implementing the same behavior as a car to speed up on a straight line will improve the motion control further.

To indicate whether a TurtleBot is turning we examine the turning velocity right before it is published to a topic. If the turning velocity is greater than 0, it is turning, otherwise, it is moving straight forward. Knowing the Turtlebot is moving straight forward allow us to manipulate the linear velocity. To make it similar to a car, we introduce an acceleration constant α, to increase the velocity gradually as long as it remains on a straight path, so velocity v the current velocity is,

$$v_t = v_{(t-1)} + \alpha \tag{1}$$

We also introduced a max velocity constant v_{max} to enforce a limit on the TurtleBot's speed, so we obtain this minimum function:

$$v_t = min(v_{(t-1)} + \alpha, v_{max}) \tag{2}$$

After implementing a method for the TurtleBot to speed up, we need to make sure the TurtleBot slows down when approaching a curve. Luckily, in Dubins path, there is only one place where a curve is preceded by a straight line. The only time when the TurtleBot is close to reaching the goal on the correct position and orientation. To slow the TurtleBot down, we assume that the end of that line segment is the same as the goal position, and we calculate the distance from the TurtleBots current position to control the linear velocity proportionally.

$$v_t = p(\sqrt{(y_{goal} - y_{t-1})^2 + (x_{goal} - x_{t-1})^2}) \tag{3}$$

As the TrutleBot approaches the goal position, the distance decreases and thus reducing the TurtleBots movement speed. To make sure this calculation is compactable with the rules defined previously, we take the smaller velocity while the TurtleBot is moving on a straight path so that it accelerate gradually in the beginning and it slows down when approaching the goal position.

$$v_t = min(v_{(t-1)} + \alpha, v_{max}, p(\sqrt{(y_{goal} - y_{t-1})^2 + (x_{goal} - x_{t-1})^2}) \tag{4}$$

Now let add in the case for when the TurtleBot is on a curve (i.e. when $v_{augular} > 0$). While the Turtle is on a curve, it should move at a safe constant velocity of v_{min}. The TurtleBot's linear velocity, while it has not reach the goal position, is then defined by the piecewise function:

$$v_t = \begin{cases} v_{min} & \text{if } v_{angular} > 0 \\ min(v_{(t-1)} + \alpha, v_{max}, p(\sqrt{(y_{goal} - y_{t-1})^2 + (x_{goal} - x_{t-1})^2})) & \text{otherwise.} \end{cases} \tag{5}$$

4.4 Implementation

```
v = ks; // Linear velocity initialized at k_min.
c = curvature(index);  // Curvature

// Angular velocity calculated using PID
avel = -ka*v*l*sinc(dtheta) - kb*dtheta + kc*v*cos(dtheta)*c/(1+c*l);

// Checking if the Turtle is turning using angular velocity
if ( !isTurning(avel) ){
        // calculate distance from goal
        double dfg = distance(turtlePose.position.x,
        turtlePose.position.y,
        goalPose.position.x,
        goalPose.position.y );

        // Uses the smaller value of linear acceleration and proportional control
        if ( lacc < km * dfg ){
                lacc += acc_rate;
        }
        else {
                lacc = km * dfg;
        }
}
else {
        lacc = 0; // Reset linear acceleration
}

// linear and angular vel
velocity.linear.x = v + lacc;
velocity.angular.z = avel;
```

5 Tips

The math and calculation here is fairly simple to follow, a good understand of Dubins Curve would help with understanding the reason why some of the short cuts/assumption in the calculation works. Here are some resources I've used to understand Dubins Curve:

1. Planning Algorithm - http://planning.cs.uiuc.edu/node821.html
2. Implementation of Dubins Curve by Andrew Walker - https://github.com/AndrewWalker/Dubins-Curves/blob/master/src/dubins.c
3. A Comprehensive, Step-by-Step Tutorial to Computing Dubin's Paths - https://gieseanw.wordpress.com/2012/10/21/a-comprehensive-step-by-step-tutorial-to-computing-dubins-paths/

6 Pitfalls

Since optimization is in a way resulting problem, Ill brief summarize the problems that the optimizations described in section 4 resolves here.

6.1 Lack of Goal Orientation

The required inputs for Dubins Curve algorithm are two position variablesone variable representing the initial position, and one representing the goal position. Both position variable has three fields, (x, y, θ), where x is the x position on the map, y is the y position on the map, and θ is the orientation. The problem we have is that the multi-agent control algorithm only generate a (x, y) pair, as the orientation is irrelevant to the algorithm. With this case unhandled, the goal position to be used for the Dubins Curve algorithm would have a θ value of 0. This generate awkward behavior since the goal position would always face toward $x = -\infty$.

6.2 Linear Velocity Capped by a Safe Speed Moving Through Curves

Since the implementation of Dubins Curve as well as the unicycle model of PID controller assume constant linear velocity for simplicity, the linear velocity is capped at a speed that is safe for the TurtleBot to make a turn. This is by no mean optimal, but luckily, it is a simple fix by adding some more conditions and variables.

7 Conclusion

With the new optimization implemented, the motion control subsystem using Dubins Curve and unicycle-model PID controller functions quite well to simulate a simple car. However, the Dubins Curve algorithm, or rather any simple car with a minimum turning radius, would need to have collision issues resolved before deployed in a group, Since the robots are parted by Voronoi cells, and this model of motion planning would cause robots to intrude another's cell and thus causing collision. in the future, we would need to address collision avoidance through object detection and coordination with the multi-agent control algorithms.