Implementing SLAM and D^*lite in a simulated e-puck robot

Mikkel Antonsen

Abstract—I tried to implement SLAM and D^*lite on a two wheeled differential drive robot. D^*lite works perfectly but SLAM does not produce a map that is good enough to run D^*lite on.

Index Terms—Problems, issues, IR is problematic,

1 Introduction

In mobile robotics there are two important problems that need to be solved, localization and mapping. Before we can have autonomous vehicles in the streets and in our homes, they need to be able to correctly estimate where they are and what their environment looks like, in order to avoid damaging property or hurting people.

The problem called Simultaneous localization and mapping (SLAM) is to estimate the map and the location at the same time. Given a sensor and an accurate location estimate it is easy to build a map of the environment. Similarly it is easy to find a robots pose given sensor readings and an unbiased map. Doing both at the same time, on the other hand, is hard. Some effective and scalable solutions to this problem does exist.

In the rest of this paper I am going to discuss path planning with D^* lite and my attempt at implementing grid mapping with Rao-Blackwellized particle filters[4], which is based on fastSLAM[3], and why I failed.

1.1 Section

1.1.1 SLAM

The SLAM problem can be succinctly summarized as

$$p(x_{1:t}, m \mid z_{1:t}, u_{1:t-1}) \tag{1}$$

or stated in words as the probability of a given map and trajectory given sensor readings and odometry. Our goal will be estimating this joint posterior. One can use the following factorization

$$p(x_{1:t}, m \mid z_{1:t}, u_{1:t-1}) = p(x_{1:t} \mid z_{1:t}, u_{1:t-1}) p(m \mid x_{1:t}, z_{1:t}) \tag{2}$$

as suggested by [1]. The posterior $p(x_{1:t} \mid z_{1:t}, u_{1:t-1})$ can be efficiently approximated with a particle filter. Each particle in will represent a belief about where the robot is, given all the evidence we have gathered from the odometry and sensors. Calculating the map probability $p(m \mid x_{1:t}, z_{1:t})$ is done independently by each particle by employing "mapping with known poses". Each particle uses its estimated x and the sensor reading it has up until time t to make the map.

To maintain an estimate of the posterior $p(x_{1:t} \mid z_{1:t}, u_{1:t-1})$ at time t the particle filter will sample particles from time t-1, using a probabilistic odometry model. Each particle is then given a new weight given

$$w_t^{(t)} = \frac{p(x_{1:t}^{(i)} \mid x_{1:t}, u_{1:t-1})}{\pi(x_{1:t}^{(i)} \mid x_{1:t}, u_{1:t-1})} \tag{3}$$

such that each particle maintains its relative importance but all the importance weights sum to 1. The particles are resampled proportionally to their given importance so that we end up with n equally weighted particles. The resampling is done to keep us from ending up with one very heavy weighted particle and instead have many smaller particles covering our probability distribution. From [2] we get that the

weight of particle i at time t can be computed by

$$w_t^{(i)} \propto \frac{p(z_t \mid m_{t-1}^{(i)}, x_t^{(i)}) p(x_t^{(i)} \mid x_{t-1}^{(i)}, u_{t-1}}{\pi(x_t \mid x_{1:t}^{(i)}, z_{1:t}, u_{1:t-1})} \cdot w_{t-1}^{(i)} \tag{4}$$

This is a fairly 'naive' version of the weighting step and more advanced ones are mentioned in the articles previously cited. Even though not optimal, it should work and so I tried to implement it to see if it worked at all. The problem I quickly faced was that the term $p(z_t \mid m_{t-1}^{(i)}, x_t^{(i)})$. This probability is given by a gaussian where the expected value is given by the euclidean distance between the position x_t and the what ever obstacle is present on the map at m_{t-1} in the direction of the sensor (0 if no obstacle is within sensor range). I did not know what the standard deviation would be, so I used 1 cm. When navigating around straight walls the robot is exceedingly unlikely to sense an obstacle with more than one sensor at the time (because the limited range of the sensors makes the distance from the front sensors to the wall ¿¿ 4 cm). As a result the probability will be equal to 1 for seven of the sensor at any given time, this obviously carries very little information and does not help the robot orient itself very much.

2 Implementation

2.1 Motion model

I decided to use a velocity based motion model using the following equation from class (with the slight modification of using $2 \cdot l$ instead l)

$$\dot{\xi}_I = \begin{bmatrix} \frac{r\cos\theta}{2} & \frac{r\cos\theta}{2} \\ \frac{r\sin\theta}{2} & \frac{r\sin\theta}{2} \\ \frac{r}{2\cdot l} & -\frac{r}{2\cdot l} \end{bmatrix} \cdot \begin{bmatrix} \dot{\phi}_1 \\ \dot{\phi}_2 \end{bmatrix} \tag{5}$$

When ever the robot moves each particle is propagated according to this model given a set of motor inputs $\begin{bmatrix} \dot{\phi}_1 \\ \dot{\phi}_2 \end{bmatrix}$. Obviously, with this model all the particles will be propagated to the exact same place. Random gaussian noise is added to $\dot{\xi}$ at each time step to spread the particles out in the state space. We have discussed in class whether or not a gaussian distribution is the best for representing the actuator noise. The problem with this representation is that the target distribution is not necessarily the shape of a gaussian and might be multi modal. These deficiencies can be mediated by using a two phase sampling with scan-matching. I will give a very brief explanation of this process in the following section.

2.2 Scan-matching

I did not get around to implementing scan-matching, as it requires a working version of SLAM. I am going to mentioned what it is for completeness' sake. If the target distribution is not gaussian or is multi modal, the particle filter might end up with all the particles centered around one of the modes. To combat this, you can use a two phase sampling.

By first performing scan-matching you can get particles closer to all modes of the distribution making all modes more likely to be represented when the regular importance sampling is done afterwards [4].

The objective of scan-matching is updating the position of each particle to the x_t that maximizes $p(z_t \mid x_t, m_{t-1}) p(x_t \mid u_{t-1}, x_{t-1}^*)$.

Stated in words, maximizing the likelihood of the current pose relative to the estimated map and estimated pose at the previous time step. This has shown to reduce the error associated with multimodal distributions [4].

3 Representing the environment

3.1 The map

The environment is represented by a probabilistic grid map as described in [5]. Where the state of a given grid cell being occupied is given by a binary variable, where 1 denotes occupied and 0 denotes free. The state a cell is in is given by a probabilistic function.

The cell size has to be chosen. In literature the recommended size for each cell is 15x15 cm, which is obviously not a good choice for a robot with a sensor range of 4 cm and diameter of 7.4 cm. I decided to use a grid size of 1x1 cm which is a nice compromise between the ratio of cell size and sensor range but still

leaving the map small enough to not be too computationally taxing.

We assume that each cell m_i of the map is conditionally independent on the other cells. Everything we know about this world tells us that if one cell is occupied, there's a higher probability that adjacent tiles are occupied. Assuming independence, how ever, we can use the following factorization when calculating the probability of a map:

$$p(m \mid x_{1:t}, z_{1:t}) = \prod_i p(m_i \mid x_{1:t}, z_{1:t}) \tag{6}$$

Further more we use log-odds for representation of occupancy. This has the benefit of both being more accurate for small probability values and makes the algorithm run faster as we can do additions instead of multiplications. To get the log-odds that a cell m_i is free we use the following formula

$$l_{t,i} = \log \frac{p(m_i \mid x_{1:t}, z_{1:t})}{1 - p(m_i \mid x_{1:t}, z_{1:t})} \tag{7}$$

To get the probability back to the former form we use the following

$$p(m_i \mid z_{1:t}, x_{1:t}) = 1 - \frac{1}{1 + \exp l_{t,i}} \tag{8}$$

3.2 Updating the map

Each particle is associated with its own map. Every time new sensor readings are available the map is updated with the new evidence. For all the tiles m_i in our sensors range (which is 4 cm) we update the log-odds ratio with the following equation

$$l_{t,i} = l_{t-1,i} + inverse_sensor_model(m_i, x_t, z_t) - l_0 \tag{9}$$

where l_0 is the prior belief of the state of the cell.

3.3 Inverse range sensor model

When we update our belief about the map we need an inverse sensor model. This model tells us the probability $p(m_i \mid z_t, x_t)$ or in words, the probability of m_i being occupied given the current position and sensor readings. Each sensor reading is a ray moving radially out from the periphery of the robot. It is easy to calculate what tiles the ray crosses using the angle of the sensor and the range. If the reading indicates that there's nothing within 4 cm, the prior for occupancy is returned. If the reading indicate that something is within 4 cm, the tile with the indicated reading is given the log-odds for being occupied. The others are given the log-odds for being free.

So what should the log-odds returned be for an occupied and freec cell? In my implementation I used the probability 1 for occupied if the cell is occupied and 0 for being occupied if it was unoccupied. With less values the robot was not able to gather enough evidence for each tile as it moved along. These values are not optimal and I'm not quite sure how to find the 'right' ones.

One thing to note is that if you lower the certainty of the probabilities returned by the inverse sensor model, you should probably make sure that you get more sensor-readings per tile m_i. In literature this is solved by having a laser sensor that gives you a 360 degrees read out at every time step. With the e-puck you only get 8 rays (covering approximately 1 degree each) at each time step. This shortcoming could be mitigated by rotating the robot 180 or 360 degrees from time to time to get several readings from multiple sensors on each tile. This would, how ever, make robot extremely slow.

3.4 Using the resulting map

It is impossible to use the map that is produced to navigate the robot. The main issue is that there will be a lot of holes in the walls on the map because of insufficient sensor coverage. The path planning algorithm will use these holes to find 'shortcuts' through the walls, that are obviously not there.

Some 'engineering' solutions could be applied to solve this problem. One could find obstacles that are less than the robots diameter apart and draw a wall between them, for example. This is not a good solution because we are injecting the belief that cells are not independent in our system, which we previously

assumed. While this is not necessarily wrong, it is not very elequent. The most problematic issue with this solution is that the robot does not have a view forwards (because of the lacking sensors), so it is impossible to 'repair' walls in the map before the robot has already crashed into it.

4 Moving the robot

At each time step the robot gives the D^*lite algorithm its position and gets a destination in return. Moving the robot from x to x' in a smooth manner was harder than I expected so I decided in the end to move and rotate separately. To get the control input for the wheels I used the following formula

$$V_{wheel} = \frac{\phi C}{2\pi \cdot T \cdot r_{wheel}} \tag{10}$$

where T is the time step, ϕ is the desired turning angle (actual heading - desired heading), C is the circumference of the robot and r_{wheel} is the radius of the wheel. The $\dot{\phi}$'s in $\begin{bmatrix}\dot{\phi}_1 \\ \dot{\phi}_2\end{bmatrix}$ were then set to V_{wheel} but with different signs. In practice this did not work quite as desired because as the robot got closer to its desired angle the desired turning angle ϕ approaches 0, making the turning rate very low. The angle would also never be perfect, which caused the robot to frequently stop to rotate more.

To solve this I use a two phase controller. In the first phase the robot turns until its arbitrarily close to the desired angle and then a proportional controller takes over. The proportional controller uses 60% of full motor input to move forwards and 40% to correct the heading.

If the robot receives a new coordinate that is on the line made by the robot and the previous coordinate, the proportional takes over. If the new coordinate forms an arbitrarily big angle (I used 1 degree) on the line, the robot turns until its angle is less than 1 degree away from the desired angle and the proportional controller takes over again.

5 D^*lite

I will not write too much about D^*lite as it has been thoroughly discussed in class, in previous assignments and as its implementation is straight forward since the pseudo code given in the D^*lite paper is a few semi-colons away from compiling.

The main challenge with implementing D^* was making the algorithm self-contained in one class (so its reusable and does not clutter up the logic in the rest of the robot) and still being able to change the cost matrix at the right time from the outside of the class. I did this by having a method in the class which is called 'changeEdge', which alters the connectivity of the graph. It also does some book keeping so that in the main loop of D^* it can look at all the edges that were altered since last iteration.

6 Conclusion

I was unable to get SLAM working properly in such a manner that I would be able to navigate the robot using the map produced by the algorithm. This was to a large extent due to inefficient sensors and also a lack of knowledge on my part on how to fully utilize the sensors. With this in mind I will be replacing the IR sensors with ultra sonic sensors when attempting a real world application of SLAM. I will also spend the first phase of the project gathering statistics about the measurements of the sensors (such as mean and standard deviation of measurements for objects at different distances).

References

[1] Arnaud Doucet, Nando de Freitas, Kevin Murphy, and Stuart Russell. Rao-blackwellised particle filtering for dynamic bayesian networks, 2000.

[2] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters. *IEEE Transactions on Robotics*, 23(1):34–46, February 2007.

[3] Michael Montemerlo, Sebastian Thrun, D Koller, and B Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. *AAAI/IAAI*, 2002.

[4] Cyrill Stachniss, Giorgio Grisetti, Wolfram Burgard, and Nicholas Roy. Analyzing gaussian proposal distributions for mapping with rao-blackwellized particle filters. *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3485–3490, October 2007.

[5] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.