COMP3431

Robotic Software Architecture

Assignment 1: Report

Christopher Manouvrier
Aaron Ramshaw
Simon Robilliard
Oliver Tan
Aneita Yang

September 8, 2015

Introduction

The objective of this assignment was to autonomously navigate an unknown maze and visit beacons in a specified order. To achieve this goal, the assignment was broken down into three main modules - exploration, beacon recognition and localisation and waypoint traversal.

In exploration, the TurtleBot's task is to map out an unknown maze while avoiding obstacles (walls and beacons). The TurtleBot achieves this by performing a frontier-based search on the data gathered from its laser scan.

In the beacon module, beacons are pinpointed using the Bot's camera. This information is then combined with the data from the laser scan to determine the position of the beacons. This module is also responsible for keeping track of when the TurtleBot has recognised all four required beacons.

Waypoint traversal guides the TurtleBot from beacon to beacon. The path the TurtleBot takes is determined by an A* search, using Euclidean distance to the goal as the heuristic.

These modules are combined to achieve the final assignment goal. The exploration module is in conjunction with the beacon module to simultaneously map the maze and locate beacons. Once the beacon module has detected the four required beacons, a message is sent signalling for waypoint traversal to begin.

Exploration

Before it can visit beacons in a specified order, the TurtleBot must first explore the maze. In this assignment, FastSLAM is used for its speed and accuracy to generate the map. Although HectorSLAM produces a higher resolution map (0.01), FastSLAM's 0.05 resolution is sufficient for our purposes.

Before exploration begins, the TurtleBot waits for an OccupancyGrid message and an Odom message to ensure start-up has been successful. A wall-follower, although guaranteed to map out the entire maze correctly, has limitations in its speed. It often spends more time than is necessary to finish mapping a location, following any walls it comes across, regardless of any existing knowledge about the area.

To speed up the exploration of the unknown maze, a frontier-based search is performed on any existing data we have of the maze. This data is delivered to us in the form of an OccupancyGrid message, which publishes an array of integers from -1 to 100 representing knowledge of the maze. The search looks for the closest frontier of -1s in the OccupancyGrid and the TurtleBot is instructed to explore this closest frontier. By moving to this frontier, the TurtleBot expands its map until the entire maze has been explored.

Letting θ be the angle of the target from the TurtleBot and d be the distance of the target from the TurtleBot:

$$d = \sqrt{(x_{BOT} - x_{TARGET})^2 + (y_{BOT} - y_{TARGET})^2}$$
$$\theta = \tan^{-1}\left(\frac{y_{TARGET} - y_{BOT}}{x_{TARGET} - x_{BOT}}\right)$$

Letting θ_m be the angle of the target from the origin of the map:

$$\theta_m = \theta - \text{bot}_{yaw}$$

d and θ_m are written to a Twist message which is then published.

To prevent collision, the TurtleBot's movement is interrupted when the robot is within 30cm of a wall. The laser scan is used to detect this, at which point the Bot is instructed to face the wall and reverse into a safe-zone. Once the TurtleBot is safe, an empty message is published, signalling to robot that it can resume it's frontier-based exploration.

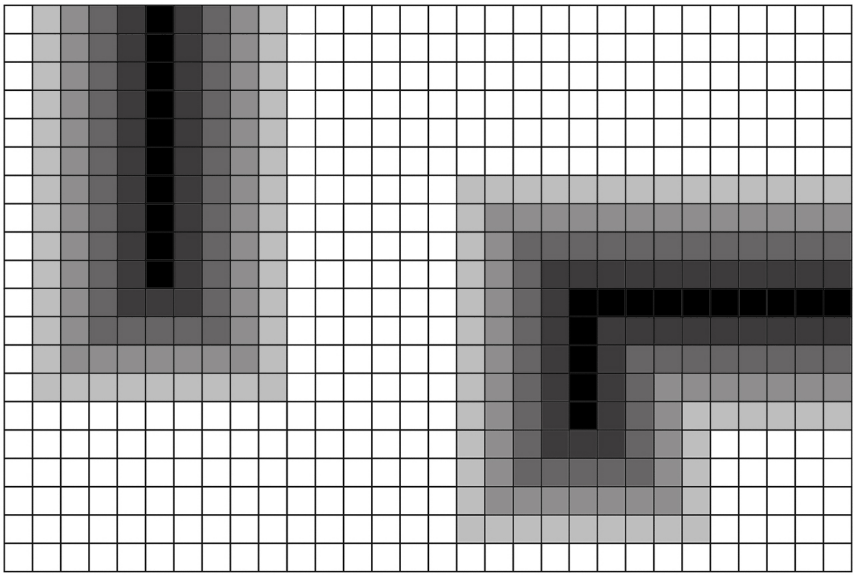


Figure 1: Example of wall fattening by 4 cells, where black is the original wall

As another safety measure, any cells in the OccupancyGrid with an occupancy probability greater than 80 are treated as having 100% occupancy and are "fattened" by 20cm. Given that the TurtleBot's radius is 15cm, this gives the robot a 5cm leeway from the walls, and a 60cm-wide pathway to travel along. To "fatten" the walls, we occupy the rings of cells surrounding every wall in the current OccupancyGrid (Figure 1). Fattening the walls also blocks any gaps between the walls that the robot may have originally seen through.

A frontier-based search is advantageous in that the robot can explore areas with an arbitrary number of obstacles (walls and beacons) and loops that may confuse a wall-follower no longer become an issue.

As the OccupancyGrid is continually being updated, our frontier-based search increases the speed of the search by eliminating the need for the TurtleBot to travel down dead-ends. Whereas a wall-follower would lead the Bot into the dead-end, the frontier-based search will not (Figure 2).

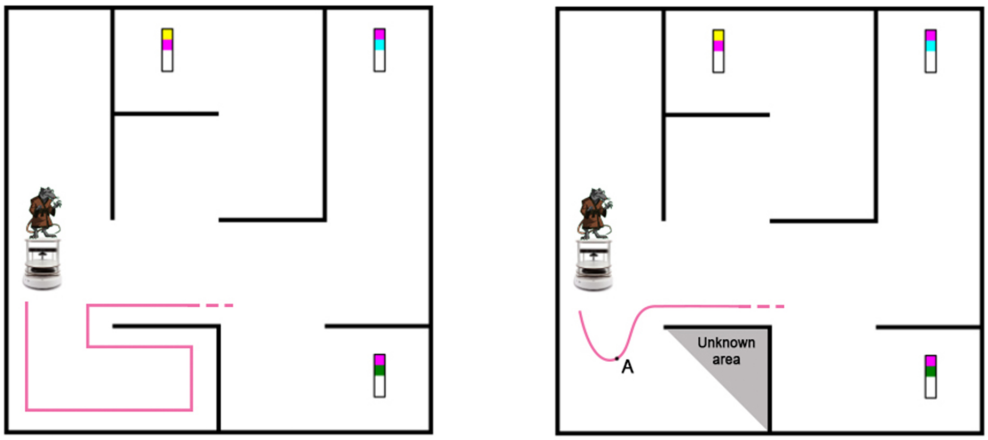


Figure 2: **Left:** Example of wall following exploration in a dead-end. **Right:** Example of frontier-based search in a dead-end.

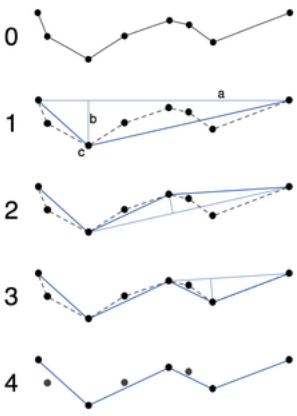


Figure 3: RDP

However, as the frontier-based search is performed on a map with a resolution of 0.05, the path can often be jagged and over-complicated. In order to "smooth" our path, the Ramer-Douglas-Peucker algorithm is used to reduce the number of intermediary points in the path (see Appendix). RDP works by recursively dividing a given path, finding an intermediary point that is the furthest from the line segment drawn from the start to the end of a path. If the distance of this furthest point is within the given threshold, the point can be discarded, thus, cutting down the number of intermediary points the TurtleBot must visit. A threshold of 0.05 is used in our RDP path simplification, meaning that points that lie within 5cm can be overlooked.

Beacon Recognition and Localisation

Detection

During the TurtleBot's first exploration of the maze, the camera is used to identify and position beacons. The beacon finder node subscribes to RGB images from the Kinect camera and processes them using OpenCV. The following criteria is used to detect the beacons:

1. Colour range filtering to see only and distinguish pink, blue, green and yellow colours of the beacons.
2. Simple blob detection to find beacon shaped objects:
   (a) Area greater than 200 pixels.
   (b) Inertia greater than 0.65.
   (c) Convexity greater than 0.5.

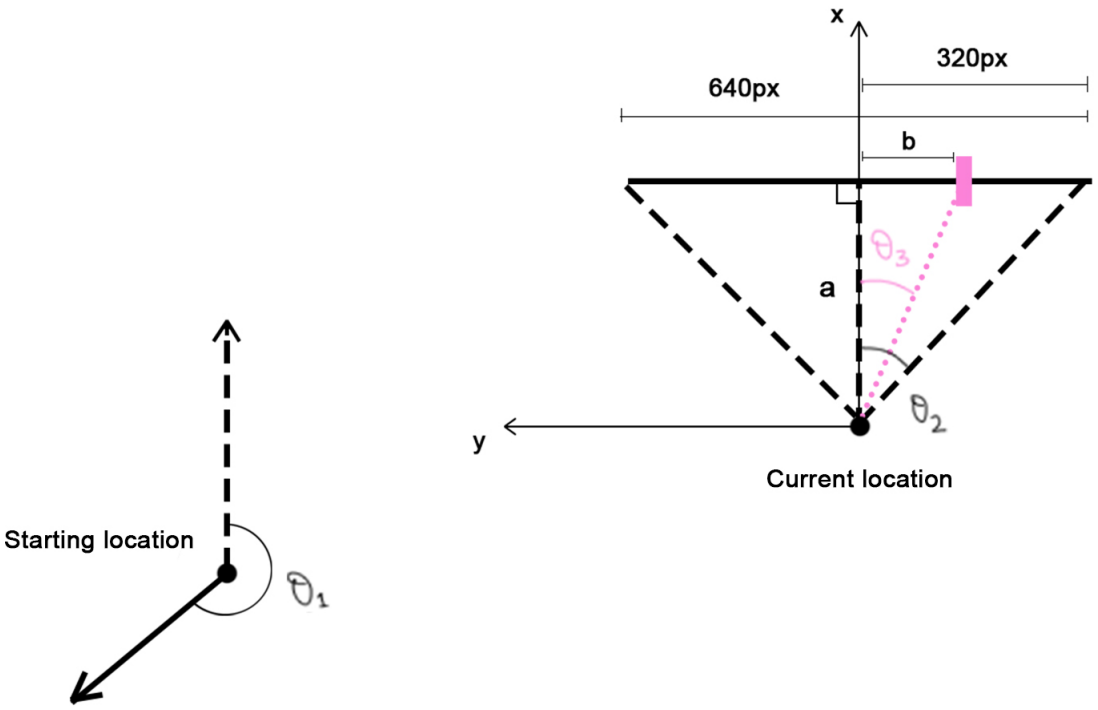
The blob detector produces key points in the form of coordinates on the 640 * 480 RGB image, in the center of the beacon colour block. We conclude that a beacon has been successfully detected if there are two "blobs" in the same image frame within a 20 pixel offset from each other in the horizontal x axis. This threshold of 20 pixels is used to account for any motion blur. The beacon's top colour is determined by comparing the two keypoints' y values.

We also use the laser scan data to find the location of the beacon relative to the bot by reading the laser data at the angle corresponding to the angle on the RGB image. This information is then used in conjunction with the odom to give the location of the beacon with respect to the global map.

Localisation

Pinpointing the locations of detected beacons is essential in order for the TurtleBot to complete Waypoint Traversal. Each beacon's position is determined by first considering the TurtleBot's position and orientation in the maze (i.e. relative to the origin of the map), and looking at data from the DepthCloud.

To determine the rotation of the beacon from the origin of the map, we consider the pixel in the image at which the beacon was detected, relative to the centre pixel of the image. It is known that the camera has a 58°field-of-view and, thus, we are able to use these pixel values to calculate the beacon's rotation from the origin.



$$\begin{aligned}\theta_1 &= \text{TurtleBot's current yaw, relative to start}\\ \theta_2 &= 29^\circ\end{aligned}$$

$$\begin{aligned}a &= \frac{320}{\tan 29^\circ}\\ b &= \text{position of beacon given as an offset from centre of image}\\ &= \text{vertical pixel in image at which beacon was detected} - 320\text{px}\end{aligned}$$

$$\begin{aligned}\theta_3 &= \tan^{-1}\left(\frac{b}{a}\right)\\ &= \tan^{-1}\left(\frac{b \cdot \tan 29^\circ}{320}\right)\end{aligned}$$

Therefore, the beacon's yaw from the TurtleBot is θ_3 and the beacon's yaw from the origin of the map is $\theta_1 + \theta_3$. Note that in the above example, θ_3 is negative as tan is negative in the fourth quadrant.

To determine the beacon's x and y coordinates in the map, we consider the TurtleBot's position, the beacon's angle from the TurtleBot (θ_3), and the array of ranges gathered from the laser scan.

$$\begin{aligned}\text{beacon.x} &= \text{bot.x} + d \cdot \cos\left(\theta_1 + \theta_3\right)\\ \text{beacon.y} &= \text{bot.y} + d \cdot \sin\left(\theta_1 + \theta_3\right)\end{aligned}$$

Letting θ_{BL} be the angle of the beacon with respect to the laser:

$$\begin{aligned}\theta_{BL} &= \text{beacon.yaw} - \text{laser.angle\_min}\\ \text{beacon\_index} &= \frac{\theta_{BL}}{\text{laser.angle\_increment}}\end{aligned}$$

$$\begin{aligned}d &= \text{beacon's distance from TurtleBot}\\ &= \text{laser.ranges [beacon\_index]}\end{aligned}$$

Using the ranges from the laser scan eliminates the need for the TurtleBot to be next to the beacon to pinpoint its location. Rather, once a beacon is within the camera's view, it's position can be determined, speeding up the process of beacon localisation. In turn, the TurtleBot's initial exploration is also sped up.

Planner

A vector of four Beacon objects is created when the beacon recognition module first begins, where the first index holds the first beacon to visit, the second index holds the second beacon to visit, and so on. Beacon objects store the top and bottom colour of the beacon as strings, x and y coordinates, and a boolean value as to whether the beacon has been found within the maze.

As beacons are detected during the exploration phase, their positions are stored in the vector. Once the four beacons have been found, their coordinates are moved from the vector to a FoundBeacons message which is published, signalling to the Exploration module to stop and the Waypoint Traversal module to begin.

This essentially allows our TurtleBot to stop exploring the maze once all four beacons have been detected, even if the full map of the maze has not been generated.

Beacon Class:

```
1  class Beacon {
2  public:
3       double x, y;
4       bool known_location;
5       string top, bottom;
6
7       Beacon(string top, string bottom) :
8            x(0), y(0),
9            known_location(false),
10           top(top), bottom(bottom) {}
11
12      bool found() {
13           return known_location;
14      }
15 };
```

FoundBeacons Message:

```
1  int32 n
2  geometry_msgs/Point[] positions
```

Waypoint Traversal

Once the FoundBeacons message is published, the TurtleBot will switch from its exploration phase to waypoint traversal.

To speed up the traversal between beacons, an A* search is performed on the available data - namely, the OccupancyGrid and the FoundBeacons message which contains the positions of the beacons. The search considers the neighbouring cells (top, bottom, left, right) of the current location and the neighbouring cell's Euclidean distance from the goal (the beacon) to find the shortest path.

A search is first performed to find a path between the TurtleBot's current location and the first beacon. We consider a TurtleBot to have reached the goal beacon if it is within a 30cm radius of the beacon. That is, if:

$$\sqrt{(x_{BOT} - x_{BEACON})^2 + (y_{BOT} - y_{BEACON})^2} \leq 0.3m$$

Once the TurtleBot reaches the beacon, another search is performed to find the path to the next beacon.

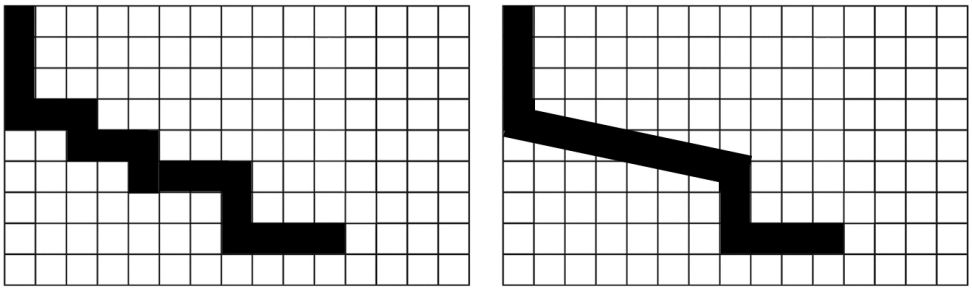


Figure 4: **Left:** Original A* path. **Right:** RDP simplified path

As the search is performed on a high-resolution map (0.05m/cell), the resulting path can be jagged and over-complicated. Like in the exploration phase, a Ramer-Douglas-Peucker path simplification is performed on each path that is returned from the A*, cutting down on the number of intermediary points the TurtleBot must visit.

To move from beacon to beacon, the same movement logic from the exploration module is used. The laser's array of ranges is used to detect any potential collisions.

Appendix

Ramer-Douglas-Peucker Path Simplification:

```
1   static vector<pair<double,double>> rdp_simplify(vector<pair<double,
        double>> in, double threshold) {
2       vector<pair<double, double>> out;
3       if (in.size() > 2) {
4           // Find the vertex farthest from the line defined by the
5           // start and and of the path
6           double max_dist = 0;
7           size_t max_dist_i = 0;
8
9           pair<pair<double,double>,pair<double,double>> line =
                make_pair(in.front(), in.back());
10
11          for (size_t i = 0; i < in.size(); i++) {
12             // Calculate the distance from the line to the point
13              double dist = distance_line_point(line, in[i]);
14              if (dist > max_dist) {
15                  max_dist = dist;
16                  max_dist_i = i;
17              }
18          }
19
20          // If the farthest vertex is greater than our threshold, we
21          // need to partition and optimize left and right separately
22          if (max_dist > threshold) {
23              // Partition 'in' into left and right subvectors,
24              // and optimize them
25              vector<pair<double, double>> left;
26              vector<pair<double, double>> right;
27
28              for (size_t i = 0; i < max_dist_i + 1; i++) {
29                  left.push_back(in[i]);
30              }
31              for (size_t i = max_dist_i; i < in.size(); i++) {
32                  right.push_back(in[i]);
33              }
34
35
```

```
36              vector<pair<double, double>> leftSimplified =
                    rdp_simplify(left, threshold);
37              vector<pair<double, double>> rightSimplified =
                    rdp_simplify(right, threshold);
38
39              // Stitch optimized left and right into 'out'
40              out.clear();
41              for (size_t i = 0; i < leftSimplified.size(); i++) {
42                  out.push_back(leftSimplified[i]);
43              }
44              for (size_t i = 1; i < rightSimplified.size(); i++) {
45                  out.push_back(rightSimplified[i]);
46              }
47          } else {
48              out.push_back(line.first);
49              out.push_back(line.second);
50          }
51          return out;
52      } else {
53          return in;
54      }
55  }
```