Mobile Robot Programming Project

Zach Busser
Kevin Cheek
Ziyan Zhou

Algorithm Design

Overview

The algorithm design can be broken down into four parts: the main Retriever loop, the localizer, the planner and obstacle avoidance.

When the program is started up, waypoints are read from the provided file. The floorplan is loaded into a GridMap. A C-space map is created from that floorplan. A local map with a higher resolution is created to map sonar readings.

The mapper runs on a separate thread and continuously updates the local map. The localizer also runs on a separate thread and continuously calculates the most likely starting position of the robot based on the local map generated by the mapper. The planner runs on a separate thread to plan and keep track of the robot's progress along the path to the goal. The obstacle avoidance algorithm also runs on a separate thread to avoid obstacles and work towards a nearby sub-goal given by the planner.

Localization

The localization algorithm can be broken down to two parts:

- The local mapper thread (localization.Mapper), which takes sonar range reading and updates a local probabilistic map. The coordinate system of the local map is relative to the starting location of the robot. The odometry starts at (0, 0, 0)
- The localization thread (localization.Localizer), which constantly compares the generated local thread to the floorplan. This thread, tries to rotate and reposition the local map onto the floorplan using the position candidates. It figures out how much portion they match. The most likely starting position is determined at any given time by the ranking of those position candidates. Then, the robot's real location is calculated by offsetting and rotating the odometry based on the starting location and orientation.
  Since the local map will be continuously updated by another thread, this thread continuously recompute all the scores for all the position candidates. It provides the path planner with the most likely position at any given time.

Path Planning

Our path planner used the brushfire algorithm to construct a Voronoi graph of the map. We could then use Dijkstra's algorithm to search the map from start to end points. Unfortunately, our implementation had some problems: either the Voronoi graph was disconnected at points, or the graph generated was too "thick", meaning that there were too many nodes in the graph. The first case caused Djikstra's algorithm to fail to find a path, because a path did not exist on the disconnected graph. The second case returned a path; unfortunately, this path took an extremely long time to find and often used graph nodes that should have not been in the graph in the first place. For instance, nodes directly along a wall. Because of this, we were forced to fall back on an A* search over the grid map. We constructed the cost function for A* in such a way that it vastly preferred staying equidistant from walls, adding an overall Voronoi feel to the approach. The unsuccessful plan planning code for Voronoi and Djikstra is located in the src/pathPlanner folder; the relevant algorithms are located in src/pathPlanner/PathPlanner.java. The A* code over the grid map is located in the src/planning folder. The planner actually creates a list of every grid point along the path; only every 10th point is sent as a waypoint to the obstacle avoidance algorithm. In general, this allows the obstacle avoidance algorithm to do its work while still maintaining the overall shape of the planned path. The differences between the planned path and the path the robot actually travels are most pronounced during turning.

A planner thread was used to keep the robot on track. The planner is able to detect which segment of the path has been successfully navigated by the robot. It provides the obstacle avoidance algorithm with the next sub-goal along the path. If the planner detects that the robot has been kidnapped, or it has strayed too far from its path due to avoiding obstacles, it will replan so as to maintain an efficient path to the goal. This was most evident in simulation when we were able to literally kidnap the robot, without having to physically pick it up. We also needed to replan if the robot relocalized itself to a different robot and starting position; obviously, the old path would need to be changed if this happened. This was most evident in the two robots in the right hall, which are difficult to tell apart until the end of a hallway is reached.

Finally, when a robot reaches a goal point, it sleeps for five seconds to indicate that it has indeed reached the goal.

Obstacle Avoidance

We used artificial potential fields to do low level obstacle avoidance. The basics of the algorithm are fairly simple: the robot is attracted to the goal point, and repelled by obstacles. We used a K value of 1.0 and an inverse cube repulsion to model the potential fields; the inverse cube means that the robot will be extremely strongly repelled by objects that are very close to it, but less affected by objects that are further away. The threshold point for "close" was settled upon as half a meter away. We also added a linear term so that the robot would be affected somewhat by objects that were a bit further away; this term modifies the potential field for objects two meters away or closer. Sonar information is read in a separate thread and passed to the potential field; this interaction is controlled by a semaphore. The code for the potential fields is located in src/navigation/PotentialField.java.

Experiments

Simulation

The grand simulation experiment involved running robots p0 and p2 on projpoints1, and robots p1 and p4 on projpoints2, with staggered start times. At this point the machine used for testing became a bit sluggish, so we stopped with four simultaneous robots. All four robots successfully localized very quickly, on the order of a few seconds, and began planning the path to their goals. Robots p0 and p2 reached their first goal at times where they did not conflict, but as robot p2 started towards the second goal it had to avoid robot p0 moving towards the first goal. This was successfully accomplished. After robot p2 reached the second goal it stopped, finished with the goal points. This meant robot p0 had to maneuver around it to land within the half meter radius; this was also accomplished successfully. The second set of robots were more staggered because robot p4 starts further away from the initial goal and was started later. The upshot to this was

that p1 and p4 met in the northeast hallway and had to avoid each other, which they successfully did. The final difficult part of the simulation was going in and out of the robot lab for the fourth waypoint in projpoints2. The robots were somewhat slow to navigate the doorway and the white robot was just beyond it, but in both cases they successfully made it into, and back out of, the robot lab.
The times for each robot are as follows:
p0, projpoints1: 9:19
p1, projpoints2: 12:53
p2, projpoints1: 5:42
p3, projpoints2: 14:24

Real World

Since we did not have every ready at the time of real world testing, we did a number of real world tests on various component of the project.

The artificial potential field was the first thing tested, and required the most modifications since it was the closest to the robot. In particular, we needed to add code to slowdown or stop when very close to objects to prevent smashing into them due to momentum in the real world. We also significantly revised our general potential field equations to come up with parameters that would be successful both in terms of avoiding objects, and squeezing through tight spaces like doorways and crowded hallways. The sub-goal navigation for APF (navigating through points on a path) was also tested in real world. The robot was able to successfully reach the sub-goal and avoid obstacles on the way.

The localization/mapping was also tested on the field. The algorithm worked for some starting points while others required a little bit help from the human operator. For example, the starting point inside the robot lab was very hard to localize because there are a lot of furniture in the room. But once the robot was herded outside of the room, it ran down the hallway and successfully localized. It definitely took a longer distance and more time to localize in all cases because the evidence collected was so noisy. On the other hand, the biggest problem with the real world localization was that the odometry of the robot was not accurate enough. For lewis, the robot tends to deviate to the left. Also, turning tend sto skew the odometry and make the map impossible to localize. A much more fine-grained localization is needed in the process (SLAM).