**COSC 081/281 - Principles of Robot Design and Programming (Term: Spring, Year: 2023)**

**Assignment: PA3 - Report**

**Author: Carter Kruse**

**Date: May 3, 2023**

## Implementation / Description

*Overview*

The purpose of this assignment is to serve as an introduction to path planning with search-based methods. Familiarity with ROS, publishers, and subscribers is a prerequisite. The map environment file is provided in Canvas.

The program includes a function that reads the occupancy grid from the related topic, takes as input the start and the goal (both of them in the reference frame associated with the occupancy grid, which can be read in the header), searches for a path in the grid, and returns the sequence of poses in that reference frame that the robot should follow.

Further, there is a simple ROS node that uses the function to find the path between the current pose of the robot (using 'odom'/'lookupTransform') and an arbitrary goal in the map. A message is published containing such poses to a topic called 'pose_sequence'. In this message, the 'frame_id' is specified in the header and the orientation of each pose is calculated so that the orientation of a point with index $i$ is pointing to the next point at index $i + 1$ (the last point has the same orientation as the previous). With this information, the robot moves following such poses.

*Breadth First Search Algorithm*

The BFS algorithm is implemented to allow a path to be found, given a 'maze' (occupancy grid), starting position, and goal position. This code is based primarily on the pseudocode provided in class. A set is implemented to keep track of the visited cells, while a queue is used to continuously add cells, thus exploring the possible paths of the robot. A dictionary structure is used to store the 'parent' nodes, which is used for the backtracking when the goal position is reached.

*Design Decisions*

In developing the program, various design decisions were made, primarily to ensure the accuracy/completeness of the code. Specifically, the linear velocity, angular velocity, angle tolerance, and distance tolerance had to be closely set, so that the robot would rotate or move as expected, allowing room for error.

Rather than simply use an angular or linear velocity, along with a time (which leads to inaccuracies), the program incorporates a closed-loop that continues to operate the robot until a certain specification is met. This is where the tolerances come into play. The tolerances were set similarly to PSet 1, and thus, will not be justified in this write up.

By using this method, rather than the velocity/time method, we are able to ensure that the error in the robot's movements does not carry forward, which would greatly impact the positioning of the robot.

The linear velocity was set to a reasonable speed for the ROS Turtlebot, at 0.1 m/s, and the angular velocity was set to pi/8 rad/s.

*Main Class*

By utilizing the BFS, the main class was significantly simplified, though it still remains relatively complex. The constants are initialized at the top of the class, followed by an initialization function for the class, which is used to set the following parameters: command velocity publisher, pose sequence publisher, odometry subscriber, linear velocity, angular velocity, goal location ($x$ and $y$), current position and orientation ($x$, $y$, $yaw$), position/angle tolerances, finite state machine, transform listener, path, and resolution (map).

A finite state machine is used to control the operation of the robot, though there are only two states to consider: when the robot is stopped, and when it is moving. That is, the goal is to keep the robot in constant motion, rather than make the robot stop to turn (even though this happens to be the case).

The *move()* and *stop()* methods are responsible for creating and publishing the Twist messages corresponding to given linear and angular velocities. The *move_to_position()* method incorporates a rotation, translation algorithm. In particular, the robot does not overshoot the rotation specified, as the angular velocity remains relatively low (compared to the tolerance), and we allow the robot to make adjustments in the case of overshooting. The robot does not overshoot the position/location specified, as the linear velocity remains relatively low (compared to the tolerance).

The *odom_callback()* method is responsible for handling the Odometry data, setting the position/orientation of the robot, while simultaneously publishing the pose sequence of the robot that arises from the *find_path()* method. This was done to ensure the pose sequence could be visualized when using *rviz*, as the *odom_callback()* method is continuously called as the robot moves.

The *find_path()* method is responsible for finding the appropriate path of the robot, according to the occupancy grid, by utilizing the BFS algorithm. The method waits for a message from the "map" topic,

and then determines the relevant information. To ensure that the robot does not run into obstacles during execution, the occupancy grid is modified to provide a buffer around the obstacles.

The *run()* method handles the movement of the robot, according to the finite state machine (*MOVE* or *STOP*). The frequency rate is set at a global variable, at 10 Hz. In the case where the finite state machine is *MOVE*, the robot moves to the various positions determined by the path-finding/planning algorithm. This prompts the robot to adjust the linear/angular velocity as appropriate. Otherwise, when the finite state machine is *STOP*, the robot remains stationary. To use the *find_path()* algorithm, the points must be determined based on a conversion that uses the 'lookupTransform' from the 'map' to the 'odom' reference frame.

**Evaluation**

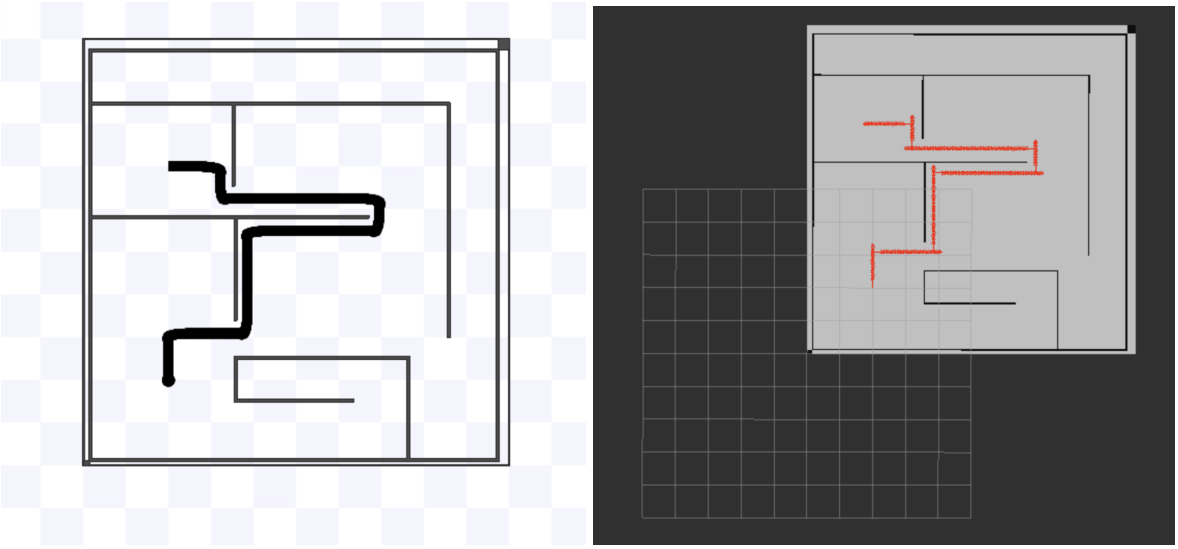
To verify the accuracy/completeness of the program, various experiments were conducted, with given parameters, highlighted as follows.

The main objective of this assignment was to implement a path planning algorithm, and use it to control the motion of the robot. As indicated previously, the BFS algorithm (with history) is used to allow the robot to determine the appropriate path from a starting position to a goal position, given the location of various obstacles. The buffer applied to the occupancy grid is used to make sure that the robot does not impact with the obstacles (walls) when getting too close.

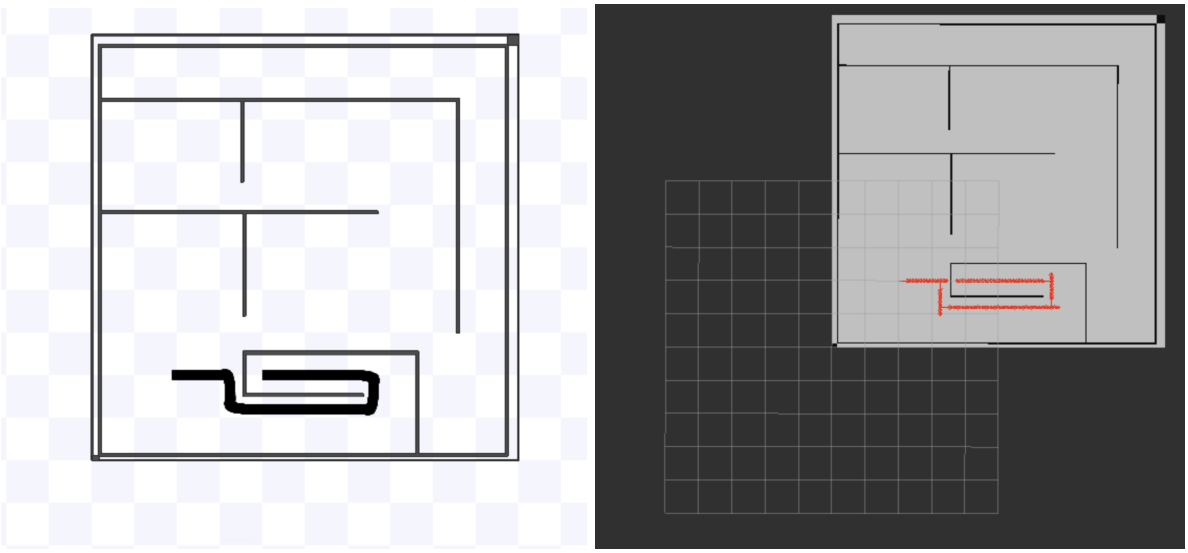
The following experiments were conducted, along with screenshots of the behavior of the robot, after speeding up the simulation. The main objective of these experiments was to ensure that the robot path planning algorithm functioned correctly. The following display the path of the robot in simulation, along with the pose sequence published to the appropriate topic and displayed using *rviz*.

See the following page.

**Starting Location: (2, 2)** **Goal Location: (2, 7)**



**Starting Location: (2, 2)** **Goal Location: (4, 2)**

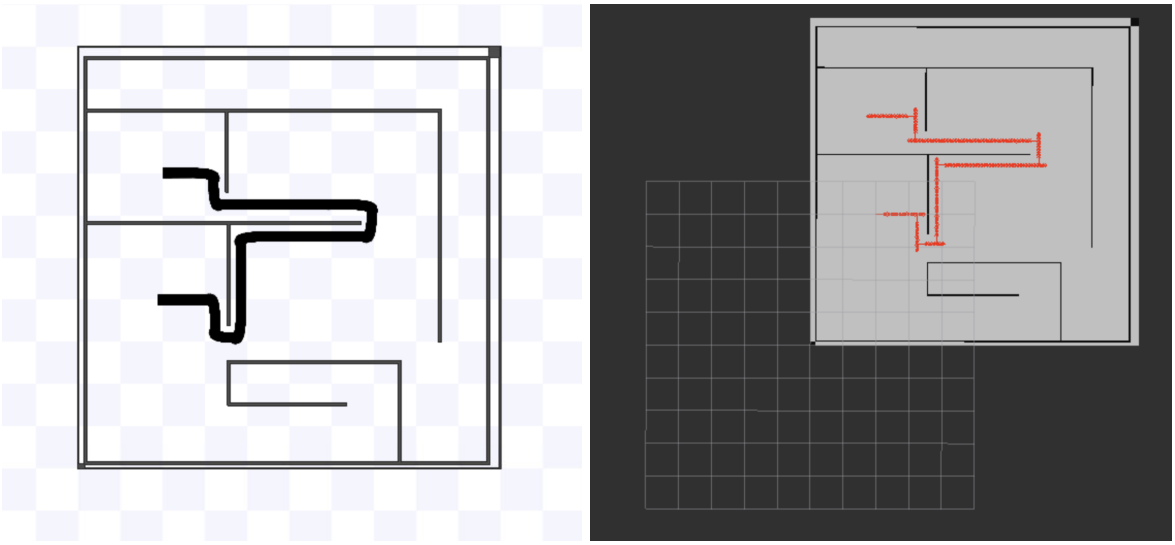


In experimenting, a reasonable result was produced, as the path planning algorithm worked as expected (even in cases where the robot was a little tight to the walls).

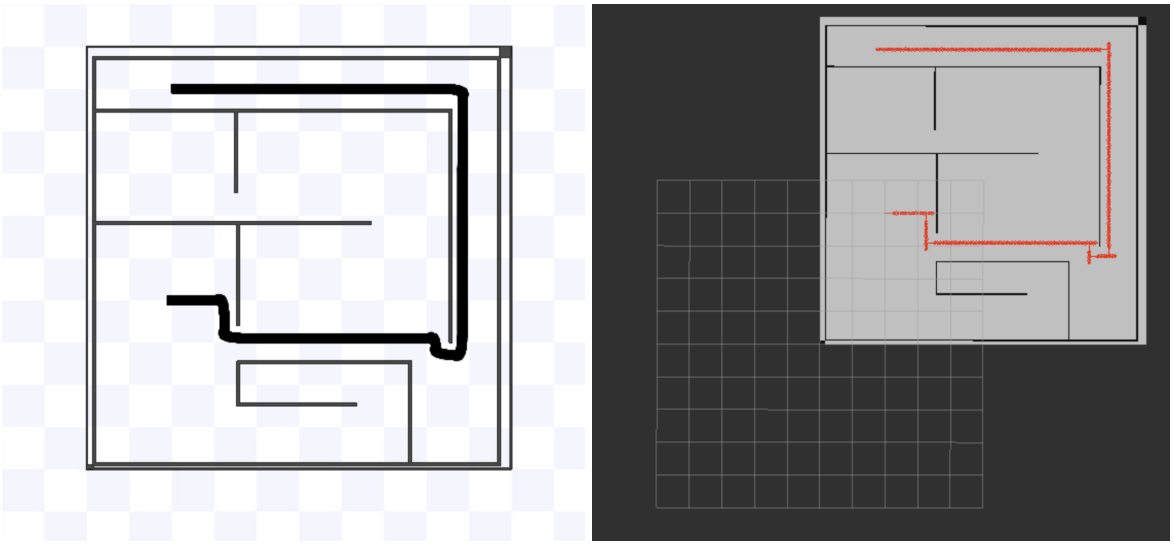
To further confirm the accuracy of the program, the starting location of the robot was modified in the 'maze.world' file to be located at (2, 4). As indicated in the following visualizations, the program functions as expected. However, to adjust the 'odom' reference frame with relation to the 'map' reference frame, you must run the following:

*rosrun tf static_transform_publisher x y 0 0 0 0 /map /odom 50*

**Starting Location: (2, 4)** **Goal Location: (2, 7)**



**Starting Location: (2, 4)** **Goal Location: (2, 9)**



*Conclusion*

The program functions as expected, according to the specifications.

The robot does not seem to encounter difficulties, in that it does not run into issues in which it collides with obstacles, if the appropriate positions are given. Further, the robot does not become stuck, oscillating between positions or rotations; the algorithm implemented includes the appropriate tolerances for the angle and distances.

The path planning algorithm seems to be relatively efficient, though it may be improved by utilizing an A* algorithm, or incorporating a different heuristic.

## Credits

OccupancyGrid Message (Documentation)
http://docs.ros.org/melodic/api/nav_msgs/html/msg/OccupancyGrid.html

PoseArray Message (Documentation)
https://docs.ros.org/en/melodic/api/geometry_msgs/html/msg/PoseArray.html