Final Project Report

CS-4023 Intelligent Robotics

Dr. Dean Hougen

Authors: Bach Nguyen-Ngo, Sandra Roy

The University of Oklahoma

I. Abstract

The objective of the final project is to display, test, and analyze an efficient and simple pathfinding algorithm for a TurtleBot which can be used as the basis of a tour guide robot. Since there already exists a built-in A* pathfinding algorithm in the *global_planner* package of ROS, we will be testing our written code in C++ using this plugin and observing how useful, precise, and reliable this algorithm is for the purpose of navigating the TurtleBot from one location to another within a pre-defined, mapped, and bounded environment . Our interest in the execution of this project is primarily to see how this algorithm could be beneficial to different tour guide robots. We were able to summarize that A* ROS global planner package proves to be useful for tour guiding and can be more efficient when provided with optimal heuristic values through parameters tuning. In addition, the initial placement of the actual robot also highly impacts the quality of both the planned global path and probability of the robot reaching its final target destination.

II. Introduction

1. **Motivation:**

   The motivation for this project is to test the A* algorithm for the purpose of path planning for Turtlebot tours in the REPF room B4 by moving along pre-defined nodes in an environment. In this case we create our own environment within B4 using tables to outline the space.

2. **Nature:**

   This project can be classified as an undergraduate project

3. **Approach:**

   For this project, we relied on a map we created of our environment using RViz. We also selected our coordinates within and defined these as the nodes to be traversed. Once the goal location is determined, we modified our files to utilize the *global_planner* A* algorithm to plan a path to move the turtlebot to its final destination, testing 2 different heuristic values to determine the most optimal value.

4. **Scope:**

   The project is limited to the scope of helping the TurtleBot perform the most essential task of a tour guide: moving to a desired target of the mapped environment from its initial location through the most optimal path produced by the A* path-planning algorithm. For the purposes of this project. The scope of the environment lies within the REPF B4.

5. **Results overview:**

   The A* algorithm is successful for the Turtlebot to reach a given goal; however, factors like the initial position of the robot, heuristic value, and proximity to obstacles can affect its ability to guarantee a path to a given location. Each time it was tested, the path planning looked different and the algorithm yielded varied degrees of success.

6. **Conclusion outline:** From the results we were able to conclude that A* is capable of being a suitable path planning algorithm for tour guide robots however it is not a complete algorithm i.e. does not always guarantee a path.

III. Approach

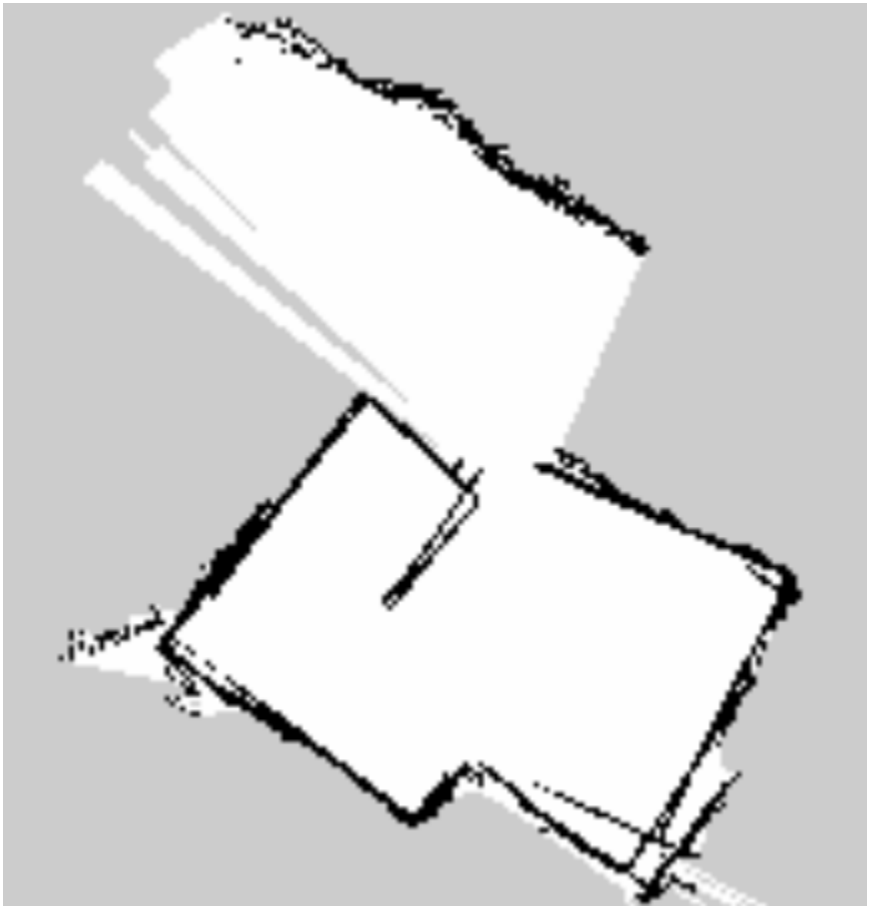
Overview: A map of a bounded environment in room B4 is created through the use of *gmapping* and *RViz*. Robot localization is done using the *amcl* package in ROS. Our program relies on specific coordinates, or nodes, in the environment that we created. One can think of these *poses* as "stations" at which the TurtleBot has to stop by on its path to the target destination. The built-in A* algorithm in the ROS *global_planner* package is used to help the TurtleBot find an efficient path to a final destination in the mapped environment. Parameters tuning of *cost factor*, *neutral cost*, and *lethal* cost is carried out to test the quality of the planned global path

1. **Building Map:**

   A mostly bounded environment was created in the REPF-Room B4 using tables to serve asr boundaries. The TurtleBot is expected to travel and navigate only within this predefined environment, which is bounded by the aforementioned tables. The initial location, the "station" positions, the final destination, and all the positions that the TurtleBot will be traveling to on both of its planned global and local paths are all defined and located within this environment:

After setting up the real world environment in which the robot would navigate in, we create a map of this environment using the *gmapping* package of ROS and RViz. After connecting the Mr. Turtle laptop to TurtleBot1, we SSH from our workstation desktop to Mr. Turtle. In separate terminals, we launch the commands (in order) *roscore* to start up the master node, *roslaunch turtlebot_bringup minimal.launch* to bring up the TurtleBot, *roslaunch turtlebot_navigation gmapping_demo.launch* to run the gmapping demo app, and finally *roslaunch turtlebot_rviz_launchers view_navigation.launch* to start up RViz. To drive the TurlteBot around from our workstation using the *keyboard_teleop* application, we launch the command *roslaunch turtlebot_teleop keyboard_teleop.launch* in a separate terminal.

Once Rviz is launched and the network is configured, we move the TurtleBot around using the *teleop* keys to construct our map. When the process of mapping is done, we save our map as *real_world_environment* using the command *rosrun map_server map_saver -f /tmp/real_world_environment*. By using *gmapping*, we are able to construct a 2-D occupancy grid map from laser, sensor, and pose data collected by the scanner of the TurtleBot as follows:



2. **Robot Localization & Obtaining "Station" Coordinates:**

After mapping, we create and edit a file called *environment.launch*, in which we define nodes, include files, pass in arguments, and configure paths for all the components we need: *minimal launch*, *3d sensor*, *maps*, *navigation*, and *sound play*. The argument for *maps* is defined as the local path in which the *yaml* file of *real_world_environment* is found (**<node name="map_server" pkg="map_server" type="map_server" args="$(find turtlebot_navigation)/maps/real_world_environment.yaml"/>**). After SSH-ing to Mr.Turtle, in separate terminals, we run the commands *roslaunch turtlebot_rviz_launchers view_navigation.launch --screen* to bring up RViz and displays the x,y,z station coordinates once the TurtleBot finishes going to a chosen location in RViz, and *roslaunch final_project environment.lauch* to load the created map into RViz and configure the environment. As mentioned earlier, we want the TurtleBot to travel to specific coordinates, or nodes, as it navigates to the final location. To obtain the x,y coordinates of these nodes, we use the *2D Pose Estimate* feature in RVIz to localize the TurtleBot in the mapped environment. We then pick random coordinates in the mapped environment in RViz for the TurtleBot to travel to by using the *2D Nav Goal* feature. As it finishes, for each location, we write down the x,y coordinates for that specific pose, ignoring the z coordinate since the TurtleBot will be navigating in 2D. The contents of our *environment.launch* file are as follows:

```
<launch>

  <!--  ************** Minimal Launch ***************  -->
  <include file="$(find turtlebot_bringup)/launch/minimal.launch"/>

  <!--  ************** 3d sensor ***************  -->
  <include file="$(find turtlebot_bringup)/launch/3dsensor.launch"/>


  <!--  ****** Maps *****  -->
  <node name="map_server" pkg="map_server" type="map_server" args="$(find turtlebot_navigation)/maps/real_world_environment.yaml"/>


  <include file="$(find turtlebot_navigation)/launch/includes/amcl/amcl.launch.xml"/>


 <!--  ************** Navigation  ***************  -->
  <include file="$(find turtlebot_navigation)/launch/includes/move_base.launch.xml"/>

 <!--  ************** Sound Play  ***************  -->
 <node name="playing_sound_node" pkg="sound_play" type="soundplay_node.py" output="screen">
     </node>
</launch>
```

3. **<u>Employing A* Path Planning Algorithm From ROS global_planner Package</u>:**

To use the provided and built-in A* path planning algorithm of the *global_planner* package in ROS, we edit the *move_base.launch.xml*, in which we add the *base_global_planner parameter* and pass in *global_planner/GlobalPlanner* as its value. The change will allow us to access the *global_planner* package, which supports both A* and Dijkstra path planning algorithms. Since we are dealing with A*, we change the parameter value of *use_dijkstra* to "false" from its default value "true". Setting *use_dijkstra* to "false" means that now the *global_planner* package will use A* for path planning instead of the default Dijkstra algorithm. Here are the contents of our *move_base.launch.xml* file:

```
<!--
    ROS navigation stack with velocity smoother and safety (reactive) controller
-->
<launch>
  <include file="$(find turtlebot_navigation)/launch/includes/velocity_smoother.launch.xml"/>
  <include file="$(find turtlebot_navigation)/launch/includes/safety_controller.launch.xml"/>

  <arg name="odom_frame_id"   default="odom"/>
  <arg name="base_frame_id"   default="base_footprint"/>
  <arg name="global_frame_id" default="map"/>
  <arg name="odom_topic" default="odom" />
  <arg name="laser_topic" default="scan" />
  <arg name="custom_param_file" default="$(find turtlebot_navigation)/param/dummy.yaml"/>

  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
  <param name="base_global_planner" value="global_planner/GlobalPlanner" />
    <rosparam file="$(find turtlebot_navigation)/param/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find turtlebot_navigation)/param/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find turtlebot_navigation)/param/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find turtlebot_navigation)/param/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find turtlebot_navigation)/param/dwa_local_planner_params.yaml" command="load" />
    <rosparam file="$(find turtlebot_navigation)/param/move_base_params.yaml" command="load" />
    <rosparam file="$(find turtlebot_navigation)/param/global_planner_params.yaml" command="load" />
    <rosparam file="$(find turtlebot_navigation)/param/navfn_global_planner_params.yaml" command="load" />
    <!-- external params file that could be loaded into the move_base namespace -->
    <rosparam file="$(arg custom_param_file)" command="load" />

    <!-- reset frame_id parameters using user input data -->
    <param name="GlobalPlanner/use_dijkstra" value="false" />
    <param name="global_costmap/global_frame" value="$(arg global_frame_id)"/>
    <param name="global_costmap/robot_base_frame" value="$(arg base_frame_id)"/>
    <param name="local_costmap/global_frame" value="$(arg odom_frame_id)"/>
    <param name="local_costmap/robot_base_frame" value="$(arg base_frame_id)"/>
    <param name="DWAPlannerROS/global_frame_id" value="$(arg odom_frame_id)"/>

    <remap from="cmd_vel" to="navigation_velocity_smoother/raw_cmd_vel"/>
    <remap from="odom" to="$(arg odom_topic)"/>
    <remap from="scan" to="$(arg laser_topic)"/>
  </node>
</launch>
```

4. **<u>Writing C++ program & Sound Play:</u>**

Next, we move onto writing our C++ program to run the TurtleBot, called *path_planner.cpp*. In our program, we prompt the users to choose between 2 options: "q" and "f". The option "q" means to quit the program, and "f" means to navigate to the final

destinations. Note that all four specific nodes or sets of x,y coordinates that the TurtleBot must travel through and to, including both the initial and final destinations, are predefined and coded in our program. When a user enters "f", the program calls on the *moveToGoal* function, which takes in 2 inputs as parameters: the x and y coordinates of a location. In the function, we define a client called *ac* that sends navigation goal requests to the *move_base server*, located in the navigation stack of ROS, through a *SimpleActionClient*. Then, the *MoveBaseAction action server* finds a global path from the robot location to the goal location through the Global Path Planner using the A* algorithm of the *global_planner* package as we specified in the *move_base.launch.xml* file, and once a path is identified, the navigation stack and move_base will instruct the TurtleBot to follows the path while avoiding obstacles through the Local Path Planner until it gets to the destination. We also define the reference frame for that position as the *map* frame, meaning that all the coordinates will be considered in the global reference frame related to the map itself. This is important since it allows the TurtleBot to "supposedly" be able to travel to any of our predetermined nodes no matter where it is currently located within the mapped environment. Note that the *moveToGoal* function is called every time the TurtleBot successfully moves from one hard-coded node (including from the initial position and to the final destination) to the next, meaning that in total, 3 different paths are constructed from A* during the duration of our program. Each time the robot successfully reaches a specific node, a distinct message is displayed, and a distinct celebratory sound is played. The sound files can be found in the *turtlebot_sounds* folder of our final_project package.

IV. Results

1. Launch 1: Turtlebot was able to reach every single node and produce the corresponding sound in order to indicate that it has reached.
2. Launch 2: Turlebot was able to reach every node except node 2.
3. Launch 3: Turtlebot reached all 4 nodes again
4. Launch 4: Turtlebot stops before it reaches final goal

We notice that the A* algorithm does not necessarily yield the exact same result for every trial and the robot may have a harder time localizing in certain instances. In certain cases it might even traverse the nodes specified, however it will fail to make the sound to signal that it has

reached if it did not move through the exact x and y coordinate. The path drawn by the algorithm in RViz was most efficient in the more open spaces, whereas the time to plan and traverse increases in tighter spaces. The initial location of the robot altered the robot's ability to reach its final goal and the path planning visualization in RViz looked different for each given trial, as A* was working with different efficiencies.

V. Discussion

Based on our research, the *global_planner* package supports different algorithms, Djkistra being the default. A* is an optimized version. Our results show that A* proves to be efficient for point to point service however we would need to do further comparison with several different algorithms to see if A* is the best choice. In the results we see that optimality of A* depends on factors such as complexity of the goal, initial position of the robot, the amount of free space, the ability of the robot to localize, the initial placement of the robot in the physical environment, and due to the heuristic nature of A*, it can be maximized by calculating a heuristic value which will allow A* to work better than Djkistra. One way to go about this is to modify the *global_planner_params.yaml* by tuning the parameters for cost factor, neutral cost, and lethal cost. It is to be noted that this kinda maximization is only possible for certain cases, whereas in other cases A* performs just like Djkistra. An algorithm is optimal when the sum of the edge costs between the initial and goal position is minimal for all possible paths. It is considered a complete algorithm when it guarantees to find a path to the final position if there exists one. Based on the results we can say that A* has the potential to be an optimal path planning algorithm. We know that a path exists for the given initial and final position due to the success of trial 1, however due to the varied degrees of traversal in trials 2 to 4, we know that A* does not always provide a path even if there exists one, therefore it is not a complete algorithm.

VI. Conclusion

We are able to conclude that A* is useful for path planning for a tour guide Turtlebot in the environment created within REPF B4 and it uses a hybrid deliberative reactive approach to plan the best path to each given node and the final goal. It has scope for optimization by changing the heuristic value and setting an adequate initial location. We also are able to observe and conclude that A* is not a complete algorithm as it does not always guarantee a path.

VI. Future Works

Future additions to this project could be increasing the versatility of the TurtleBot as a tour guide by introducing global localization, which in turn could allow users to enter their own nodes for the robot to travel to. The code can also be upgraded to be able to test and compare multiple algorithms in order to be able to determine if A* really is optimal. As previously discussed, A* works best with a heuristic value, if an optimal one exists. To achieve this, future projects could include functionality to calculate the costmap for the algorithm and tune the cost factor, neutral cost, and lethal cost parameters.

VII. Bibliography:

The following websites were consulted:

hsplanguide_icaps05ws.pdf (cmu.edu)

http://wiki.ros.org/global_planner

http://wiki.ros.org/turtlebot_navigation/Tutorials/Build%20a%20map%20with%20SLAM

https://roboticsknowledgebase.com/wiki/common-platforms/ros/ros-global-planner/

https://github.com/aniskoubaa/gaitech_edu.