CSCE 752 - Project 4d

How the program works

Launch file is used to launch the nodes that allow the project to run. It takes in the command line argument 'robot_file' which is used to determine what robot file is needed to get the information about the current robot and 'world_file' which is used to determine what world description file should be used. It runs the built-in ROS2 node robot_state_publisher as well as custom nodes 'simulator_node', 'velocity_translator_node'.

When the program starts, world description file is read and 2D map is created for the world. Using this 2D map, a graph is created which contains empty cells as nodes. Nearby nodes are identified and edges are created between them if they don't cross any obstacle. When a goal_pose is posted using 2d goal pose in rviz2, goal_pose_callback function is executed in the simulator_node.

- If the goal pose is inside an obstacle, it is discarded.
- If the goal pose is directly reachable from current location, straight line path is followed (no Dijkstra performed, path is empty).
- Otherwise, nearest_start_node is found which is the nearest node to the current location of robot. It then finds the nearest_goal_node which is the nearest node to the desired goal location. Dijkstra algorithm is executed to find a path between nearest_start_node and nearest_goal_node.
- Goal pose is appended to the path.
- After this, simulator_node publishes Twist messages on /cmd_vel so that robot first moves to the nearest_start_location. It does the same to make the robot move along the generated path and then from the nearest_goal_node to the actual goal node and finally for angle correction to align with the goal pose. The robot stops after reaching the current goal node. If a new goal pose is published while the robot is moving, current path is discarded and new path is generated again.

Sample command used to run code:

ros2 launch project4d start.launch.py world_file:=src/project4d/world_files/maze.world robot_file:=src/project4d/robot_files/normal.robot

1. **start.launch.py**

This launches the 2 nodes that come together to create the simulator. The robot's description is loaded from a specified robot file using the load_disc_robot module.

Key components:

- **Launch Arguments:**
  We declare robot_file as a launch argument so that we can use it within the simulator node to run load_disc_robot. Additionally, we were able to access this value within the launch file without using DeclareLaunchArgument. We used sys.argv to grab that value so that we can use it in our robot state publisher node without having to hardcode it. In the same way, we

access our laser parameter values to pass to our simulator node. We also have world_file as a launch argument using the regular DeclareLaunchArgumentmethod.

- **Load_disc_robot**
  We call this function to get the robot's information from the correct robot file so that we can pass it to the other nodes.

- **Robot State Publisher Node**:
  This node publishes the robot's state to the ROS2 ecosystem. It utilizes the robot's URDF description for this purpose. This is a built-in node that we didn't have to modify at all.

- **Simulator Node**:
  This node we created for simulating the robot's behavior. It takes parameters such as the robot's description, radius, height, wheel distance, and error characteristics that we get from the robot file. It subscribes to the topics **/vl** and **/vr** taking these to be the velocities of the left and right wheels of the robot and will periodically broadcast the transform of the robot's location based on these values. It will also periodically broadcast the map of the environment from the world file on the topic **/map**. If the robot moves into an obstacle represented on the map, then it will stop instead.

- **Velocity Translator Node**:
  This node translates Twist commands into the relevant velocities of the left and right wheel of a differential drive robot for it to move in the same way the Twist command would tell it to. It takes the robot's description and uses that to interpret what the velocities should be.

2. velocity_translator_node.py

This node subscribes to twist messages with topic **/cmd_vel** and uses the angular and linear velocities received to calculate the velocities of the left and right wheels of the differential drive robot should be. Once it calculates them it publishes them as Float64 messages to the **/vl** and **/vr** topics respectively.

Key components:

- The node declares two parameters, 'distance' and 'radius', representing the distance between the wheels of the robot and the radius of the wheels, respectively. Which we get from the robot file.
- The node **subscribes** to the **/cmd_vel** topic, for messages of type **Twist**. When it receives them it calls the **cmd_vel_callback** function.
- In the **cmd_vel_callback** function, we get the linear and angular velocity and then calculate the vr and vl we need. We then publish these on the topics **/vl** and **/vr** with the type **Float64**.

3. simulator_node.py:

This node receives the **/vl** and **/vr** topics and uses them to simulate the robot's movement using

tf transforms. It also keeps track of goal poses, generates path towards goal and makes the robot follow the path.

Key components:

- The node subscribes to **/vl** and **/vr** and has a timer with a period of 0.1 that has a callback called **timer_callback**.
- It also declares various parameters for the robot information that are passed in from the launch file.
- When the vl or vr callbacks are triggered they update the **self.vl** and **self.vr** values with the received message's value. If the error is not zero they multiply the velocities by their respective multipliers. Additionally, they update the **self.velocity_update_time**
- The **timer_callback** function handles calling all the other functions within the code. It gets the current time and if it has been more than a second since **self.velocity_update_time** then it sets the velocities to 0. If the time since the last error update is greater than the error update rate it calls **update_errors**. It will then call update_robot_pose to get the pose after the current velocities are applied. Then it calls broadcast_tranform to actually update the change.
- **update_errors** calculates the error multipliers from a random gaussian function with the variance from the robot file.
- **update_robot_pose** figures out the change in pose that would occur from the vl and vr velocities using equations from the differential drive section of our course. It calls is_collision before actually broadcasting the transform to make certain that the move wouldn't cause the robot to enter any obstacles.
- **broadcast_transform** calculates the translation and rotation needed to move the robot into its new position and then calls the tf2 broadcaster to do this.
- **get_map_data**
  Opens the inputted world file and parses the data within. From the data it gets the initial pose and sets the current self.pose to that and it stores the resolution of the map. Then it iterates through the map characters themselves and creates a 1-dimensional array that can be published as an occupancy grid and 2 2-dimensional array that can be used in other functions to represent the occupancy grid. It also stores the height and width of the map. It then creates a timer to call publish_map every 2 seconds.
- **publish_map**
  Publishes the map as an occupancy
- **is_collision**
  This is called before update_robot_pose makes an update. It takes in an updated x and y and returns whether the new coordinates would result in a collision. It does this by checking the points in the 360 range around the center of the robot that are a radius' distance away to see if they intersect with an obstacle.
- **create_graph**
  This function constructs a graph representation of the environment based on a given map. It iterates through the map, dividing it into grid cells determined by a specified resolution. For each empty cell, it creates a corresponding node in the graph, identifying it uniquely by its coordinates. It then finds neighboring empty cells for each node and establishes edges between them, forming the graph structure.

- **find_neighbors**
  This function identifies neighboring empty cells for a given cell position (x, y) in the map. It considers four cardinal directions (right, down, left, up) and calculates potential neighboring cells. It validates these neighbors, ensuring they are within the map boundaries, empty, and not obstructed by obstacles. It returns a list of valid neighboring nodes for the specified cell.
- **path_passes_through_obstacle**
  This function checks if the straight-line path between two points in the map intersects with any obstacles. It calculates intermediate points along the line connecting (x1, y1) and (x2, y2) using a specified step size. For each intermediate point, it checks in a circular pattern around that point with a defined radius to determine if any part of the path intersects with obstacles. If any intersection is detected, it indicates that the path passes through an obstacle.
- **goal_pose_callback**
  This function serves as a callback for receiving the goal pose. When a goal pose is detected, it first checks if the specified goal pose is within an obstacle. If the goal pose is inside an obstacle, it discards it as it's not a viable target. If the goal pose is valid, it initializes various parameters such as goal_pose, path, angle_corrected, and reached_goal. It then creates a temporary dictionary to hold the goal pose coordinates. Next, it determines if there exists a direct obstacle-free path between the current robot pose and the goal pose. If there's a clear path, it sets the path as empty. Otherwise, it proceeds to find the nearest nodes in the constructed graph based on the current robot pose and the goal pose. Using Dijkstra's algorithm, it generates a path from the nearest node to the start to the nearest node to the goal in the constructed graph. This path represents a feasible trajectory for the robot to traverse. If there's a direct path available between the current pose and the goal pose, the function simply appends the goal pose coordinates to the path. Otherwise, it appends the goal pose coordinates to the generated path.
- **follow_path**
  This function controls the robot's movement along the path. It sets tolerances for both distance (0.01) and angle (0.01). If a path exists, it initiates the movement by creating a Twist message. It considers the current pose and the next goal in the path. If the current goal is the final destination, it calculates the distance and angle error between the current pose and the goal. It adjusts the robot's angular velocity to align its orientation with the goal direction and linear velocity to reach the goal position. If the goal is an intermediate point, it computes the distance and angle error similar to the final destination. It adjusts the robot's angular velocity to align its orientation and linear velocity to move towards the goal position.
- **correct_angle**
  This function corrects the robot's orientation (theta) towards the goal's orientation. It calculates the angle error between the robot's current orientation and the goal's orientation. Using a Twist message, it adjusts the robot's angular velocity to align its orientation with the goal's orientation.
- **find_nearest_node**
  This function identifies the nearest node in the graph based on the robot's current pose. It iterates through the graph nodes, calculating the Euclidean distance between the robot's pose and each node, ultimately returning the nearest node.

- **calculate_distance**
  This function calculates the Euclidean distance between two poses using their x and y coordinates.
- **minDistance**
  This function finds the minimum distance vertex among nodes not yet included in the shortest path tree. It iterates through the graph nodes and returns the node with the minimum distance that is not already part of the shortest path tree.
- **generate_path**
  This function utilizes Dijkstra's algorithm to find the shortest path from the nearest start node to the nearest goal node in the graph. It initializes distances to nodes, explores adjacent nodes, and constructs the shortest path using predecessors. Finally, it returns the generated shortest path.

Did the results meet your expectations? Why or why not?

The results met the expectations on an average. It was hard to figure out the velocities which should be published. Creating collision free edges was also challenging. Sometimes robot will take unnecessary path even though the goal is not in that direction. I believe that it is because I have used a simple path generating algorithm which does not give an optimal path. Robot reaches the goal pose and follows the path correctly eventually. It also successfully discards goal pose which is either in obstacle or out of reach.

Credit: Part of the document has been taken from previous submissions of Group 11 (Jonas Land and Ayushri Jain)