Motion planning for a KUKA youBot mobile manipulator

ED5215 : Introduction to Motion planning
Course project Report

by

Balaji R (ED20B008)
Nikhil S (EE20B090)
Kanishkan M S (ME19B192)

1 Abstract

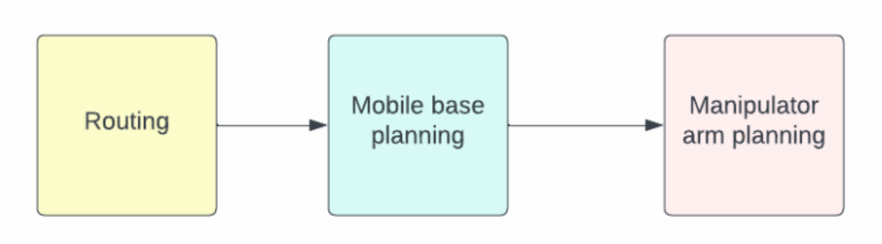
This project addresses the motion planning challenge for a KUKA YouBot mobile manipulator to efficiently pick up multiple objects and transport them to a designated basket. The robotic system comprises a mobile base and manipulator arm, with a maximum carrying capacity of three objects.

The proposed solution involves separate planning for the mobile base and manipulator arm. **Dijkstra algorithm** calculates costs between objects, start, and goal states, enabling optimized path planning for the mobile base. A modified **Traveling Salesman algorithm** is then utilized to find the shortest route for the mobile base, considering the object pickup sequence. The **RRT*** algorithm is implemented for manipulator planning, incorporating pick-up and drop mechanisms required for object manipulation. By combining these algorithms, an optimized motion plan is developed.

The project aims to enhance the efficiency of the KUKA YouBot mobile manipulator in pick and place operations. By planning the mobile base and manipulator arm separately and optimizing paths, the system can effectively navigate the environment, pick up multiple objects, and transport them to the designated basket within the given carrying capacity.

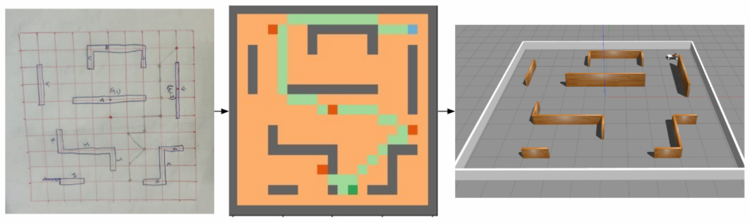
2 Methodology

The mobile base planning and manipulator arm planning are done separately and combined together in the Gazebo simulation world using ROS. Thus, The implementation methodology could be divided into four parts: World building, Mobile base planning based on the world, Manipulator arm planning and Pickup-drop mechanism.

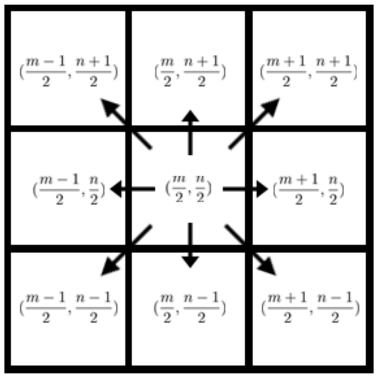


2.1 World building

For creating a world in the gazebo simulation environment, various ideations were considered. Creating an occupancy grid map using the sensors in the robot, plan by plotting graphs. But this involved a need for a considerable amount of time, hence we prioritized the effort towards implementing motion planning algorithms efficiently. We created an environment manually in Gazebo by making a custom discrete grid based on the map given for search algorithm assignments. The assignment map was modified according to the pen sketch shown below, and the gazebo map is also modified manually according to the sketch.



The robot is constrained to take discrete steps with a step value of 1/2(It could be reduced even further too). If the robot is at (m/2,n/2) where m and n are integers. Then the robot can move to the locations as shown in the figure below.



$(\frac{m-1}{2}, \frac{n+1}{2})$
$(\frac{m}{2}, \frac{n+1}{2})$
$(\frac{m+1}{2}, \frac{n+1}{2})$
$(\frac{m-1}{2}, \frac{n}{2})$
$(\frac{m}{2}, \frac{n}{2})$
$(\frac{m+1}{2}, \frac{n}{2})$
$(\frac{m-1}{2}, \frac{n-1}{2})$
$(\frac{m}{2}, \frac{n-1}{2})$
$(\frac{m+1}{2}, \frac{n-1}{2})$

2.2 Mobile base planning

For mobile base planning, three routing algorithms are implemented and tested. They are as follows:

- Beacon route searching
- Dijkstra + modified TSP + Recurrent A star
- Dijkstra + modified TSP

2.2.1 Beacon route searching:

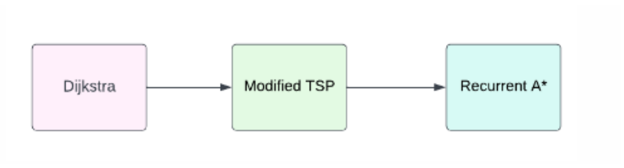
This algorithm takes all the objects as beacons and the constraint of maximum carrying capacity being three is implemented by manipulating the state expansion function in the search algorithm. The state representation is as follows:

- START STATE = [[a,b],False,False,False,False...(n times for n objects) , 0]
- GOAL STATE = [[c,d],True,True ,True,True...(n times for n objects), 0]
- STATE = [[x,y], bool, bool...(n times for n objects) , no of objects in it]

Modified state expansion function: State[-1] represents the number of objects in it. This can be used to implement the constraint of maximum carrying capacity being three. If state[-1] == 3, the bool of all objects cannot be turned true unless the goal location is reached. If the goal location is reached ([c,d]), state[-1] is set to be 0 (since it drops objects at the goal location) and the process again continues until the goal state is reached.

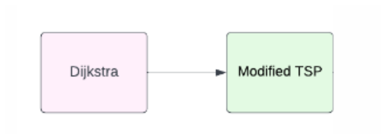
We used an A* search algorithm with a modified state expansion function and objects as beacons. This generates the optimum path for traversal considering the maximum carrying capacity being three.

2.2.2 Dijkstra + Modified TSP + Recurrent A*:



- This algorithm runs Dijkstra n times for n objects, keeping the objects to be their start node. From this, we get a three-dimensional array dist which stores the distance of all nodes from objects. (Ex: $dist[i][j][k]$ gives the value of the distance from ith object to (j, k) coordinate.
- The dist is given as input to modified TSP (TSP—Travelling salesman problem) where the permutation code in the TSP is tweaked to accommodate the constraint of the problem statement (maximum carrying capacity being three).
- The found routes are now checked for their total distance and the route with the lowest total distance is picked as the optimal route. This modified TSP returns the optimal traversal order
- The optimal traversal order is passed on to recurrent A star where it loops over the order and generates a path between every adjacent node by running A* keeping adjacent nodes as start and goal.
- At last, we get the final optimal traversal path which satisfies the constraint of maximum carrying capacity being three.

2.2.3 Dijkstra + Modified TSP(Without Recurrent A*):



- This algorithm runs Dijkstra n times for n objects, keeping the objects to be their start node. From this, we get a three-dimensional array dist which stores the distance of all nodes from objects. (Ex: $dist[i][j][k]$ gives the value of the distance from ith object to (j, k) coordinate.
- It also returns a dictionary variable called "paths" which contains the optimum path between every object, start, and goal node.
- Similar to the previous method, the modified TSP outputs optimal traversal order based on the 3d-dist matrix as input.
- A loop is run over the optimal traversal order and using the path dictionary returned by the Dijkstra, the final path is obtained by stitching the paths between every adjacent node.

2.3 Manipulator planning

The manipulation pipeline involved utilizing different sampling-based algorithms to address the problem. The algorithms were evaluated based on their execution time and path length. The entire problem was solved in the configuration space, which considers the joint angles of the manipulator. Each joint angle was subject to specific constraints. The joint limits of each of them are:

$$
\begin{aligned}
Q1 &= [-180^\circ - 180^\circ] \\
Q2 &= [-65^\circ - 90^\circ] \\
Q3 &= [-170^\circ - 151^\circ] \\
Q4 &= [-102.5^\circ - 102.5^\circ] \\
Q5 &= [-167.5^\circ - 167.5^\circ]
\end{aligned}
$$

A reset state with all the angles made to zero is initialized A pick state(say *pick_state = np.array([-0.1, -0.42, -2.8, 0.4, -0.1])*) is found using trial and error and the robot moves to the appropriate location so that the manipulator can reach this pick state and pick up the object. Multiple goal states are found to place the objects on the platform and based on the number of objects already on the platform, a given goal state is chosen by the pipeline and the object is placed on the platform accordingly. The goal states for various positions:

$$
\begin{aligned}
goal\_state\_1 &= np.array([-3.1, 0.1, -2.8, +0.1, 0.0]) \\
goal\_state\_3 &= np.array([2.9, 0.05, -2.8, +0.1, 0.0]) \\
goal\_state\_2 &= np.array([3, -0.02, -2.5, 0.1, 0.0])
\end{aligned}
$$

There are corresponding intermediate states, which are being used to prevent collision of the manipulator with the objects.

$$
\begin{aligned}
inter\_state\_2 &= np.array([3, 0, 0, 0, 0]) \\
inter\_state\_1 &= np.array([-3.1, 0.0, 0, +0.0, 0.0]) \\
inter\_state\_3 &= np.array([2.9, 0.00, 0, +0.0, 0.0])
\end{aligned}
$$

This intermediate state is added since there is no external collision checker in place. RRT, RRT*, and Bidirectional RRT are the sampling algorithms implemented and their performance was compared. Common functions for all three are made and any specific function with respect to a given algorithm is made an attribute of the particular class.

2.3.1 Implementation paradigm

1. The tree is initialized with the start node in place(in bidirectional two trees are made one from the start and one from the goal.
2. A random node is sampled in the c space with a bias towards the goal.
3. The nearest node to the sampled node is found in the tree.
4. A new node with a step towards the sampled node is taken.
5. The new node is checked for being a valid state.

RRT	RRT*	Bi-directional RRT
The new valid state is added to the tree and checked for being within a ball in cspace with a very small radius and centered around the goal configuration. • If the condition is satisfied, the path to the start node is found. • Else the process repeats.	The nodes which are within a radius defined by $10 * step\_size * sqrt(log(n)/n)$ Where n is the number of nodes in the tree, from the new node is found and the node having the least (cost+cost to the new node) is chosen as the parent to the new node. • This local neighborhood is rewired then based on the costs, and this is broadcasted for all the child nodes of these nearby nodes. • Once this is done the process is similar to RRT	Here RRT is done from both the start node and goal sides. The major difference is the "goal reached condition" where the distance condition is checked not with the start or goal node, but suppose a new node is found in one tree the node closest to it in the other tree is found and the condition is checked between these two nodes. Based on these the path from start to goal is found.

2.4 Pick-up drop mechanism

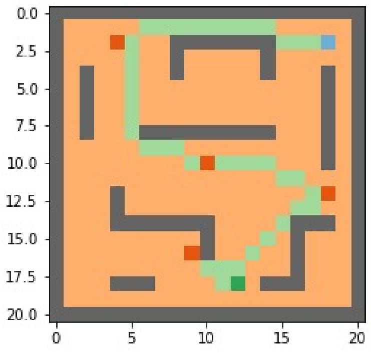
Before discussing the actual method used for pick and drop, it is important to mention a few issues encountered in the process.

- One of the main challenges in the project was determining the correct configuration for the robot's pickup and place actions. The robot's URDF had an undiscoverable issue resulting in the robot drifting slowly and continuously.
- The next major issue was with the gripper's size and the decision of the object's size. Closing the gripper did not exactly solve the problem and if the gripper closes a little more, it pushed the object out of the gripper.
- The Frictional properties of the gripper also caused issues and the object was not gripped properly and slippage occurred.

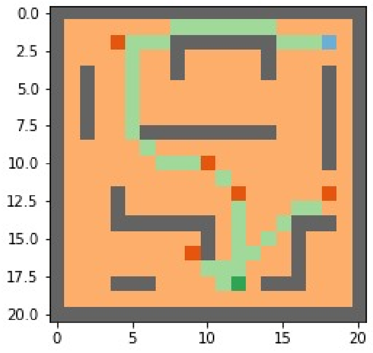
To overcome these challenges, we opted for a simple solution: attaching the object as a virtual link to the robot during pick-up and detaching it during drop-off, regardless of precise positioning. This approach allowed us to resolve multiple issues simultaneously, providing an efficient method for manipulating and transporting objects. There was an open-source **gazebo-ros-link-attacher** package that was available and it was exploited for our use case.

3 Results

Path generated based on these three algorithms: (All three algorithms provide the same path)



((a)) Four obstacle

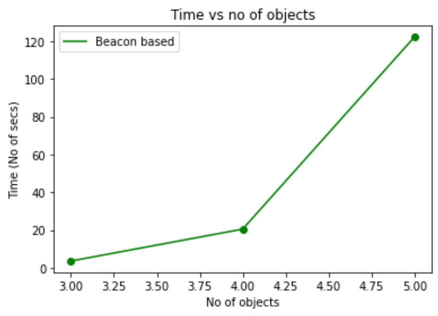


((b)) Five obstacle

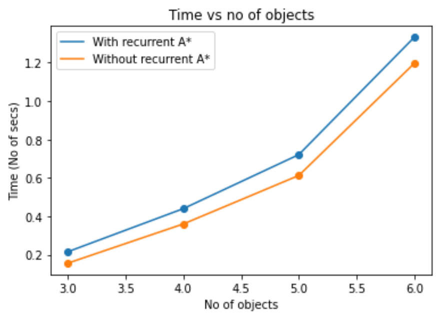
We did a comparative analysis based on the different algorithms that had been tried out and implemented for both the manipulator and mobile base.

3.1 Mobile base planner:

The below graph shows the algorithm performance time with an increase in the number of obstacles from 3 to 5 for the beacon-based search method.



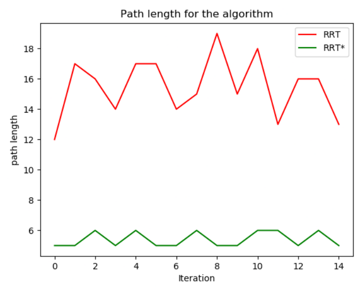
The below graph shows the algorithm performance time with an increase in the number of obstacles from 3 to 6 for the other two methods that use Dijkstra and Modified TSP.



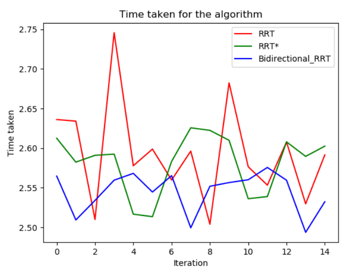
- We initially experimented with a beacon-based method which took 20secs to solve a 4-object picking problem under required constraints, This seemed a bit slow and time-consuming, prompting us to develop a new algorithm.
- Subsequently, we devised the Dijkstra + Modified TSP + Recurrent A* algorithm, which exhibited significant performance improvements over the beacon-based method by solving the 4-object picking problem in just 0.44 seconds). However, we discovered that using Recurrent A* was redundant when Dijkstra alone was sufficient.
- Finally, we implemented the Dijkstra + Modified TSP algorithm, which outperformed the other two approaches in terms of solving a 4-object picking problem. Consequently, we determined that Dijkstra + Modified TSP was the most effective algorithm and utilized it in the final implementation, achieving a solving time of just 0.36 seconds.

3.2 Manipulator planner:

The below plots show the comparisons between the three algorithms on the given problem statement. It can be seen that RRT* tries to produce the optimal path length with 5-6 configurations. And in the case of plain RRT, the algorithm is not optimal.



In the time taken graph, it can be seen that the bidirectional RRT performs better than RRT as the nodes expand in both directions with every iteration. But nothing could be said or interpreted between RRT* and Bi-directional RRT performance, both seem to take same time to solve.



4 Discussion

In our proposed method, we implemented the planning for base and manipulator separately in a sequential manner. The robot base reaches a state, then RRT planner is triggered followed by pick and drop action. Simultaneous planning is something that could have been tried. Such a planning framework would help us implement one of our mentioned extended deliverables **Planning for the "KUKA youBot" to do a blackboard drawing in "Unity game engine"**. There are certain improvements that could be done to our current method:

- **IK solver** Pick and drop states were hard coded, but inverse kinematics is a must to make this implementation more robust to noises in the navigation control and it improves adaptability with respect to the environment. The object's state in the environment can be found using gazebo plugins and using this state information IK solving should be done to find out the configuration of the manipulator.
- **Collision check:** This was not properly implemented in this pipeline and this has to be implemented to check the collision between the object and the obstacles, manipulator and base, and manipulator with the environment. This will also help the implementation to find the valid states and check if there will be any collision between the sampled states.
- **Hector mapping:** In our method, the map is built manually and planned by hardcoding obstacles and objects' positions. Instead, an occupancy grid could be created using packages such as Hector mapping, which makes use of a LIDAR sensor and does SLAM with no odometry and less computational power.
- **Automating TSP permutations :** In our current mobile base routing algorithm, the TSP permutation calculation procedure has to be manually modified for different numbers of pickable objects. It will be better if this is automated and would be easier to expand this implementation to N object cases.

Link to demo video

References:

1. ED5215 Course lecture-2 (Search algorithms) Author: Bijo Sebastian
2. ED5215 Course lecture-6 (Sampling planners) Author: Nirav patel
3. What is an Occupancy Grid Map? https://automaticaddison.com/what-is-an-occupancy-grid-map/
4. Traveling Salesperson Problem - Dynamic Programming: https://www.youtube.com/watch?v=XaXsJJh-Q5Y
5. youBot potential fields motion planning https://github.com/jonhoff14/youBot$_{p}otentialfields$
6. ROS documentation: https://wiki.ros.org/Documentation