Comparison of RRT and RRT* Path Planning Algorithms for Pick-Place with a 6-DoF Arm

Scott Shaw

I. Introduction

This work will compare the performance of two path planning algorithms: Rapidly-Exploring Random Tree (RRT) and its modified counterpart, RRT* to determine pros and cons of each algorithm. Experiments will be run in simulation using PyBullet on a six degree of freedom (DoF) arm. No vision system will be implemented or used, and therefore the locations of any manipulated objects will be assumed. The task used to examine the performance of the algorithms will be the movement of a set of boxes from one side of the robot to the other. To estimate performance benchmarks, data on resulting path costs, computation time, and the count of states created during experiments will be collected.

Section II includes relevant background information about RRT and RRT*. Section III explains the details of the 6-DoF robot arm and box objects used in simulation. Section IV describes how the simulation was prepared and what external libraries were used. Section V presents experimental results gathered from simulation. Section VI discusses issues faced during the project and outlines future work.

II. Algorithm Background

Unlike common planning problems, such as in graph search problems, path planning in robotics is often done in a continuous space. Therefore, solving the robotics path planning problem involves

1) generating nodes to fill a continuous space (in order to find the goal state)
2) determining a path from the generated nodes from the start state to goal state.

Robotics-focused planning is usually done in configuration space, which is a complete space containing every point in the robot system. The following algorithms complete the search of the continuous space by randomly growing a tree within a robot's configuration space.

A. RRT

RRT creates a tree starting at the current configuration of the robot and randomly grows the tree until either a node is within the goal state or some other exit condition is reached. When a random node is generated, it is connected by an edge with the closest node within the tree. RRT very quickly explores a continuous space with an increasing amount of accuracy as it is run. The algorithm generally grows with a "cubic" structure because of how new nodes are connected to the tree. The search problem following the exploration of the space is straightforward because RRT creates a tree, therefore no graph search is needed. RRT is probabilistically complete, but has not guarantee of optimality.

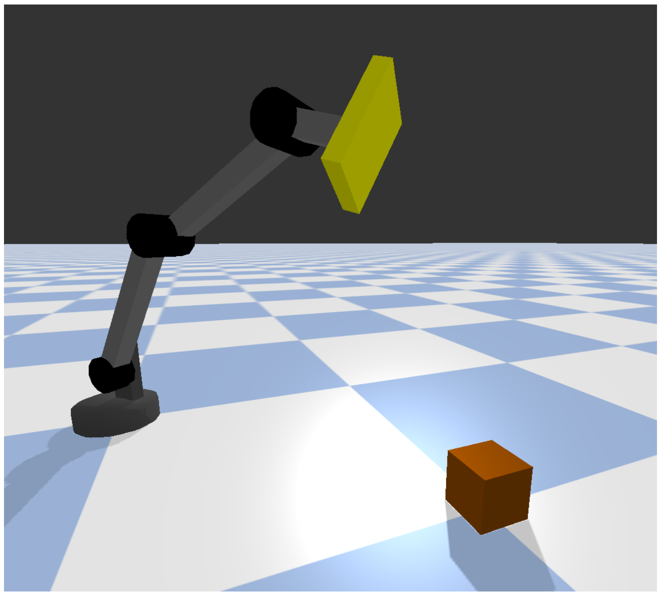
Fig. 1. 6-DoF arm posed to pick up a small box

B. RRT*

RRT* is a modified version of RRT which seeks to guarantee optimality, unlike the standard algorithm. When new nodes are added to the tree with RRT*, the algorithm looks at surrounding nodes and connects the new node to the node with the least cost in the area. Additionally, the parents of the connected node are examined to see if their cost can be reduced by going through the new node and their edges changed if it's favorable. RRT* is both probabilistically complete and guarantees optimality, but has worse computational performance when compared to the standard RRT algorithm.

```
RRT($q_{start}$, $q_{goal}$)
    T.add($q_{start}$)
    $q_{new} \leftarrow q_{start}$
    while(DISTANCE($q_{new}$, $q_{goal}$) > $d_{threshold}$)
        $q_{target}$ = RANDOM_NODE()
        $q_{nearest}$ = T.NEAREST_NEIGHBOR($q_{target}$)
        $q_{new}$ = EXTEND($q_{nearest}$, $q_{target}$, $expansion\_time$)
        if($q_{new}$ != NULL)
            $q_{new}$.setParent($q_{nearest}$)
            $T$.add($q_{new}$)
    $ResultingPath \leftarrow$ T.TraceBack($q_{new}$)
    return $ResultingPath$
```

Fig. 2. RRT algorithm

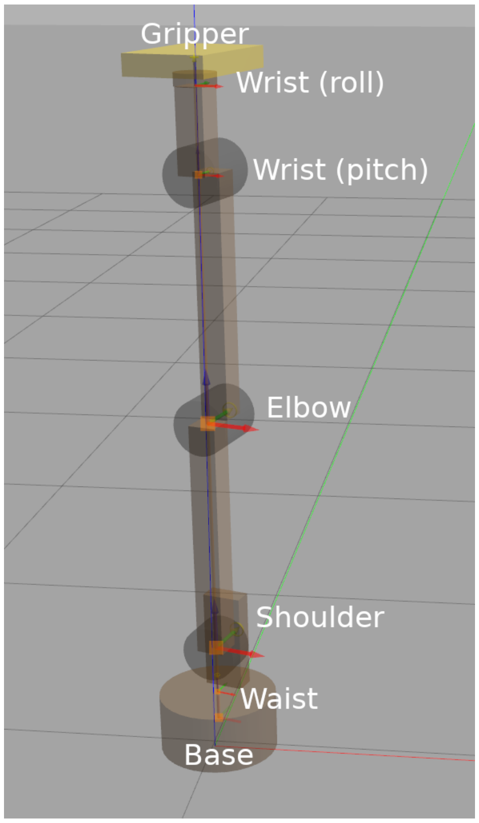

Fig. 3. Important features of 6-DoF URDF arm model

III. Robot Description

The robot arm used for this work is a simple 6-DoF arm, represented by a simple URDF model using basic shapes. It was chosen to create a simple URDF model instead using a existing, readily available robot arm model for customizability and learning. This was particularly important for the gripper because I wanted to use a suction gripper, and it was challenging to find existing models which had this type of manipulator. Moreover, creating my own URDF model allowed me to easily adjust link lengths, joint properties, joint limits, etc. The arm has two continuous joints: a waist joint connected to the base and a wrist joint to control the roll of the gripper. It also has three revolute joints: a shoulder, elbow, and wrist (for the pitch of the gripper). The main link lengths are 60cm, the shoulder joint is 12.5cm off the base, the wrist joint is 20cm, and the gripper is 20x30cm. The object used for manipulation is a small box, also created using a simple URDF model, which allowed me to easily adjust the size and mass of the object.

A. *Gripper*

The gripper is a simple rectangular prism which represents a suction gripper. The front face, which is used for manipulation, has dimensions 20x30cm. A suction gripper was chosen because it is not only easier in regards to tolerances when manipulating objects, but is also industry standard for commercial pick-and-place robots. To simulate the suction capabilities of the gripper, objects are temporarily fixed to the front face using a PyBullet constraint. One of the issues

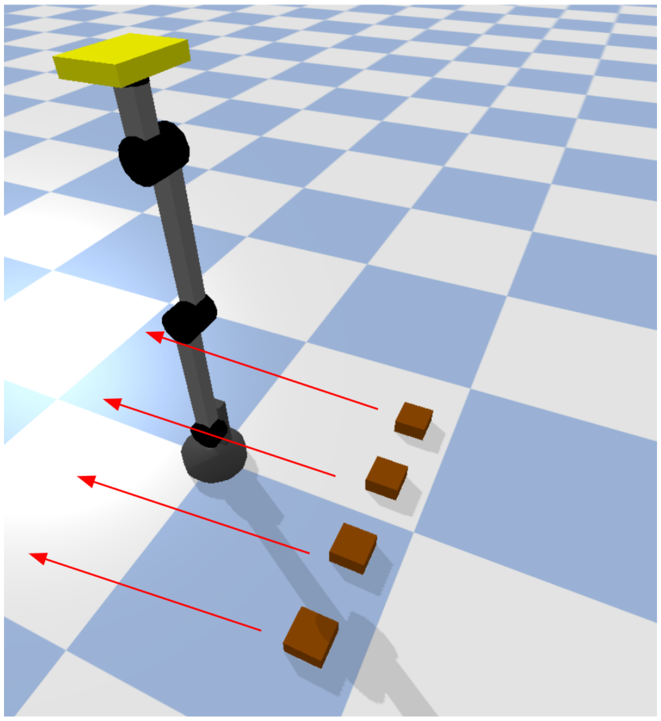

Fig. 4. Initial configuration of the simulation

observed during experiments is that the objects would not remain statically fixed to the gripper as it is moved about the world. Specifically, objects would slightly shift around while being held by the gripper; this can be seen in the results section. Although not the topic of this work, these constraints would also allow for multiple objects being manipulated simultaneously, which is another reason suction grippers have become popular. If this work is extended to simulate a more realistic pick-place application, this capability would be crucial.

IV. Simulation Setup

All experiments were simulated in PyBullet using the arm and box objects described in section III. RRT and RRT* were implemented using the Open Motion Planning Library (OMPL). OMPL contains many different planners, but does not give any collision detection or visualization ability. As such, we also use the PyBullet_OMPL library to better integrate OMPL's python bindings with PyBullet and streamline the planning and execution of paths. This library also includes collision checking using the URDFs from the simulation. The benchmark experiment, which is used for gathering all results is set up as follows. A set of four boxes is placed in a line on one side of the robot arm such that all boxes are within the robot's workspace. The locations of these boxes is noted and used in path planning. The goal of the robot is to pick up the boxes and move them to their mirrored position on the opposite side of the robot. For all experiments, a five second bound is placed on path planning. For details about the planning bound, see section VI.

Path Planning Statistics: RRT vs. RRT*					
Algorithm	mean time	mean states	mean cost	var time	var cost
RRT	0.53337	13443.4	6.91732	2.29339	4.95393
RRT*	5.02094	11233.8	5.297	0.00004	0

TABLE I

time, state, and cost statistics of RRT and RRT* algorithms of five trials.

V. Results

Each algorithm (RRT, RRT*) was run over five trials and the time, states, and cost of each planning action were recorded. The means of the recorded data is shown in Table 1 as well as the calculated variance for time and cost. An example of a trial of RRT can be seen in this video: `https://bit.ly/41m2L25`, and the process of a single pick-place motion can be seen depicted in Fig. 5.

From the data, it was observed that the RRT experiments produced a large amount of variance for both time and cost, but had a comparatively low average computation time compared to RRT*. Moreover, RRT also suffered from planning non-optimal paths and thus had a higher average path cost during planning. The number of states generated was similar for both models, however RRT created slightly more. The better path planning of RRT* can be attributed to the algorithm's guarantee of optimality. This feature is also the reason for the slow computation time and the low variance. The variance for both calculated features is basically zero because the optimal paths for each box should be the same over any number of trials. The cost variance being zero is proof that the algorithm is actually finding the optimal path consistently over each trial.

The main takeaway from the data shown in Table 1 is the difference in time, cost, and variance. RRT is suited to tasks where speed is important but consistency and path cost are not. Alternatively, RRT* is beneficial in scenarios where path cost is the top priority and the optimal path needs to be found reliably.

VI. Discussion

Many of the issues encountered during this project were related to simulation setup. The original proposal suggested using ROS and Gazebo for simulation and MoveIt 2 for algorithm implementation. Although I was able to setup ROS and Gazebo, I ran into many problems when installing various ROS packages, namely ROS2 control. Therefore the decision to use PyBullet was made because I had more experience with PyBullet in the past.

Aside from simulation installation issues, the gripper did not perform as expected in PyBullet. The force placed on the constraint may be too great, but nonetheless, there should be a better way to statically fix an object to the gripper. Although this issue persisted throughout this work, the gripper effectiveness has not impact on the data generated and therefore was not a large concern when working through the analysis of the two algorithms.

During testing, different bounds on planning time were tested. Increasing max planning time resulted in no better

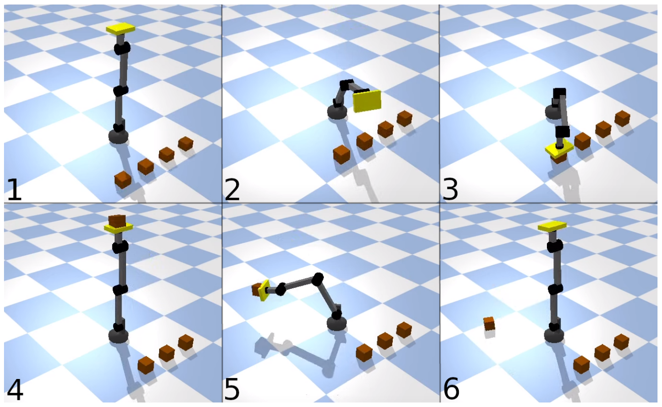

Fig. 5. Robot arm process of picking and placing a box

performance and instead simply made RRT* take longer (it would use all the time given). Attempting to decrease the bound however did impact performance negatively. A five second bound was therefore chosen as it was observed to produce good results.

If this work were to be continued, I would want to focus on

1) adding additional algorithms to test
2) creating a more challenging environment to stress the algorithms further
3) improve gripper ability (i.e., fix gripper constraint issue).

Between RRT and RRT* the former ended up having better results visually (i.e., it looked more smooth and had less disruptions due to planning time). The large increase in planning time for RRT* does not seem at all beneficial for this application (or at least in the situation covered in the work). The extra cost of RRT paths on average is far less than the time added for each planning event when using RRT*. RRT* would be beneficial in situations where either cost matters more than speed or when planning could be pre-computed as to not delay motions.