PADM Project Fall 2023 README Documentation

Annabel Gomez, Ingrid Wu

Fall 2023

1 Introduction

Given a kitchen environment with a Franka robot manipulator arm, we want to learn how to control it through activity and motion planning to execute a set of goals. Our objectives are to relocate objects using the manipulator. Finally, we will attempt to optimize it and shorten the path length to make the trajectory more intentional and direct.

2 Activity Planning

Assumptions

In our kitchen domain design, we assumed a continuous countertop with a central burner. Then located along the top of the counter is a line of closed drawers. There at three total drawers, and each one has a different color. Right above the counter is a cabinet.

Initially, we have our setup such that there is a sugar box located on the burner and a spam box located at the right countertop. The goal is to move the sugar box to the left countertop. Then, we want to stow the spam box inside the red drawer.

Files

There are two main files: the domain file (kitchen.pddl) and the problem file (pb.pddl). The domain file specifies our predicates, types, and actions. The problem files specify our objects, initial state, and goal specification.

In our domain file, our types include drawer, top, cabinet, box, and robot. We have predicates that detail whether a drawer is opened or closed and whether the robot arm is occupied or holding an item. Our actions include: having the robot arm grab an object from inside a drawer, having the robot arm grab an object from the countertop, having the robot arm place an object on the countertop or in the opened drawer, and finally, having the robot arm opening/closing the drawer.

In the PDDL problem file, we define the objects and the initial state of the kitchen as well as the goal state. The objects that exist in the kitchen domain include drawers (d1, d2, d3), tops (t1, t2, t2), cabinet (c1), and boxes (b1, b2). The initial state of the kitchen is specified with the sugar as b1, the three color assignments to the three drawers, and t2 as the stovetop. As described in the problem statement, the initial state is set such that the sugar box is on the stove (t2) and the spam is on the right counter (t1). The goal state was specified as the sugar being on the left counter (t3) and the spam being in the first, red drawer (d1).

Activity Planner

After designing our domain and problem files, we implemented an activity planner using breadth-first search (BFS). Using the pddl_parser library, [1], we parsed the domain and the problem files into a state space search problem. We then implemented a BFS algorithm to solve the plan. We chose the BFS over DFS because it is guaranteed to find the solution between the starting point to the goal in fewer steps. Additionally, BFS is better in situations where there are solutions closer to the source, which is helpful in our case since we want the robot arm to place the spam and sugar boxes in their correct respective locations in as few steps as possible.

Challenges

The main challenges we ran into were specifying the relative positions of everything in an effective manner without making the problem too complicated. It was also important for us to think through the precise steps that must be taken by the robot arm to get to the goal state but also the resulting consequences of each action on the current state. The alternative approach was to use typings, but we chose not to pursue this.

3 Motion Planning

3.1 Implementation

We aim to implement an execution engine that runs the activity planner and then executes the corresponding motion planning interface for each activity planning operator. We build a sample-based motion planner using a rapidly exploring random tree (RRT) which is designed for efficient search in non-convex, high-dimensional spaces.

3.2 Assumptions

We assumed that we would be performing the base and arm motions separately. Goal arm joints pose for boxes and drawer handling were identified by hard-coding within their limits. We also assumed that For collision checking, we only

considered visible obstacles, excluding interactions between the robot gripper and drawer handle as collisions.

Additionally, we assumed that the robot would be initialized at a random position before the activity started, so we found the path from a randomly sampled position.

We will assume that we do not have to fix the plan if it fails.

3.3 Integration

Integrating our activity and motion plans involves several steps. Initially, "enterprise.py" ties the process, generating an activity plan using a planner and PDDL files for problem and domain definition. This plan is parsed and executed by "execution_engine.py", where the extract_act function maps PDDL parameters to actions, locations, objects, and corresponding robotic joints.

Activity execution entails planning and performing actions. Our motion_planner, employing RRT, creates paths for base and arm motions based on start and goal poses, robot components, and the environment. Specific paths are executed using 'executeArm' or 'executeBase' functions, with outcomes visualized in a simulator.

The 'execute_act' function integrates motion execution and planning. It converts a parsed activity into a motion sequence, typically comprising pre_action, base action, arm action, and post_action. Each action is planned and executed in sequence by the planner.

3.4 Results

The results of our motion planner and the robot executing the goals can be found in the video linked below:

Motion Planner Video

As demonstrated in the video, the RRT successfully completes the required tasks of having the robot and its manipulator move around the environment to relocate the sugar and spam boxes to their respective positions. It also successfully avoided collisions with itself and the environment. However, it is obvious from the way that the robot moves and with the time it takes to execute its goals, that the generated path is perhaps not the most ideal or direct.

3.5 Challenges

The main challenges we face fall into two categories: PyBullet and RRT. The challenges with PyBullet involved the inverse kinematics and the gripper. At edge cases, the outputted RRT path would violate the inverse kinematics. For example, when we attempted to have our gripper grasp the drawer handle, since it's a small, exact point to reach, getting there with the RRT would not satisfy the required inverse kinematics. We decided to hard code a goal pose for the arm joints instead. Additionally, attempting to get a hold of the drawer handle without having the RRT report this as a collision was another challenge we

faced. With our assumption in mind, we accepted the path with this 'collision' happening.

Within RRT, one of our challenges was computing the distance from our current node, to our closest nearest node (within a specified exploration radius that satisfies the joint limit constraints) that we would want to connect to. We decided to explore the joint space (instead of the physical space) from our current node to our next node for the arm motions. Because of this, this method of numerically expanding does not always guarantee that the movements are going to follow the inverse kinematics. In edge cases, this then led to further challenges with local minima and our manipulator's arm getting stuck in a position in which it would collide with itself and couldn't escape. At this point, the plan would fail, and we would restart the configuration from the beginning of the plan. We can note, however, that this is not a problem that is unique to us. It's known that "especially in the complex scenario, the existing RRT planning algorithms still have a low planning efficiency, and some easily fall into a local minimum" [2].

4 Trajectory Optimization

4.1 Implementation

For this next step, we picked a free space manipulator trajectory (a trajectory of the Franks arm, moving in the free space without grasping or holding an object. Instead of using the constrained nonlinear optimization approach, we decided to optimize our trajectory using an RRT*. Without explicit mathematical nonlinear program objectives and constraints, RRT* still implicitly optimizes for the path.

4.2 Optimization Problem

We are trying to optimize the path length of the robot. We would like to minimize the generated path from the RRT so that our motions are more efficient and we accomplish our goals faster.

The path length for each path, P, as a function of the start pose and the goal pose: $L(s, g)$. To optimize this, we can set in place a post process of the path such that if we consider each node, n_i, in P, where $i \in L$. Thus, if for n_i, there exists a feasible connection to n_{i+2}, we can remove node n_{i+1} from our path. Essentially, this is an RRT*. An RRT* is an optimized version of RRT such that when the nodes approach infinity, the RRT* algorithm gives the shortest possible path to the goal. The main difference is that RRT* records the distance each vertex has traveled relative to its parent vertex and saves it as the cost of the vertex. After the closest node is found in the graph, vertices within a fixed radius from this new node are analyzed to see if one can be found with a smaller cost. RRT* also requires the tree such that after a vertex has been connected to the cheapest neighbor, the neighbors are examined again to see if

being rewired to the newly added vertex will decrease their respective costs. As a result of these two changes, RRT* tends to produce straight graphs.

4.3 Challenges

Theoretically, if you let RRT* run infinitely, it would be optimized. However, since we would like to extract a path, we cannot let it run forever. This leads to the problem of balancing path cost and run time. As a result, we can have an available path that's not optimized with few iterations. This would lead to the performances not being improved by much. Another option is to let the RRT* run for a long time to generate a better path. However, if we iterate too much, the improvement to the path could be marginal compared to previous iterations. So determining when to stop iterating was a challenge. We approached this by setting a fixed limit of 20 iterations before checking the quality of the improvement and deciding whether to proceed. This limit was determined through experimentation.

Additionally, as the tree grows with nodes, there comes a case where even with a small radius, exploring all of the parent nodes to extend from the nearest possible node (with no collision) becomes computationally heavy. However, given that we have a fixed amount of iterations, we avoided this.

4.4 Results and Comparison

The results of our RRT* trajectory optimization for a single arm motion in the open space can be found in the video linked below:

RRT* Video

The video above shows the robot's manipulator trajectory in free space. The difference between the RRT and RRT* performance is clear. The first half of the video shows an RRT arm motion, and the second half of the video shows an RRT* arm motion. The main noticeable difference is in how the RRT generates a more random and less direct sequence, whereas the RRT* gets to the goal faster, and the generated path is more intentional. However, computationally, the RRT* does take longer to generate a path, which is expected. Finally, just like with the RRT, we still deal with the rare edge cases in which the local minimum is not resolved. When this happens, we assume the task failed and restart the configuration.

5 Conclusion

In this project, we were given a kitchen environment with a manipulator arm we wanted to control through activity and motion planning to relocate two boxes. We generated an activity planner that solves a plan using BFS. Then we implemented a sample-based motion planner using RRT and PyBullet as a simulator. The RRT successfully completed the required tasks. However, we realized that from the way the robot moves, the generated path was not the

most ideal or direct. As a result, we attempted to improve our trajectory by implementing an RRT*. The RRT* iteratively worked to generate the shortest path. The result was a more intentional and direct motion plan.

References

[1] Mau magnaguagno. Pddl parser: Classical planning in python. 2015.

[2] Haojian Zhang, Yunkauan Wang, Jun Zheng, and Junzhi Yu. Path planning of industrial robot based on improved rrt algorithm in complex environments. In *2018 IEEE Access*, pages 1–11, 2019.