6.4132/16.413 Principles of Autonomy and Decision Making Final Project

Toya Takahashi

Massachusetts Institute of Technology
Department of Aeronautics and Astronautics

December 11, 2023

1 Introduction

Activity planing, motion planning, and trajectory optimization for a robotic arm is challenging. To navigate around an environment, the agent must perform a state space search to plan a sequence of actions and generate a trajectory by using a sample-based planner or modeling the problem as an optimization problem. This project explores 1) defining an activity planning problem with the Planning Domain Definition Language, 2) implementing the Fast-Forward heuristic planner with Enforced Hill Climbing, 3) using RRT with goal-biasing for trajectory generation, and 4) comparing the results with the trajectory generated by constraint optimization. The kitchen simulation environment provides an excellent sandbox for testing and demonstrating the effectiveness of the proposed approach, in addition to learning the benefits and drawbacks of the implemented algorithms.

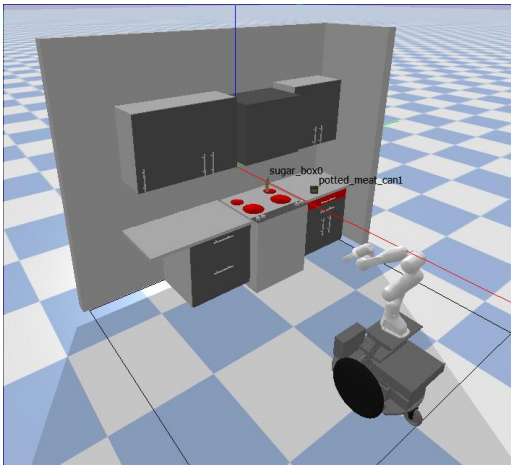

Figure 1: The kitchen simulation environment

2 Activity Planning

2.1 PDDL Domain and Problem

The PDDL domain and problem files encode the problem as sets of objects, types, predicates, actions, and goals. There are two main types of objects: the `box` type, which represents the sugar/spam boxes that are manipulated by the robot arm, and the `storage` type, which represents the storage spaces in the kitchen. In addition, the `storage` type is separated into two subtypes: `static`, which are storage spaces without actions (stovetop burner and countertop), and `openable`, which are storage spaces that can be opened or closed (drawer, cabinet).

The following assumptions were made when designing the actions and the initial state:

- The robot can only grab one box at a time
- The robot cannot open drawers or cabinets when gripping an object
- Only one box can be placed on each storage space at a time
- Drawers and cabinets must be open when placing or picking up a box inside them
- Drawers and cabinets must be closed at both the beginning and end of the plan

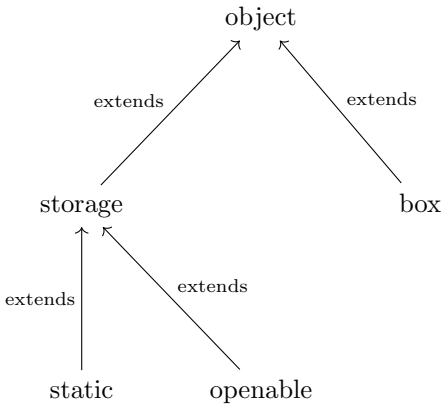

Figure 2: Hierarchical PDDL object types including static/openable storage spaces and boxes

Separating the `storage` type into subtypes allowed separating the action of placing (and picking up) a box on a storage space into `place_static` and `place_openable`, which has an extra `(open ?s)` precondition.

I also did not use any negative preconditions to simplify the implementation of the fast-forward heuristic planner. For actions requiring negative preconditions, I defined a second predicate which is always the negation of the predicate needing to express the negative precondition on (e.g. `(open ?s - openable)` and `(closed ?s - openable)`).

2.2 Fast-Forward Heuristic Planner

The `ff_planner` file contains all functions to solve the activity planning problem. The `calculate_hff` function computes the fast-forward heuristic h_{ff} of a given state by counting the number of actions (without delete effects) it takes until the goal condition is a subset of `state`. At each iteration, `new_state` is computed by taking the union of the current state and the add effects of all applicable actions. Because we defined our actions to only have positive preconditions, we can easily determine whether an action is applicable by checking if the precondition is a subset of `new_state`.

In this project, I decided to implement Enforced Hill Climbing in the `solve_ff` function. While EHC is not guaranteed to find an optimal solution, its greedy nature typically leads to higher efficiency and lower memory usage. `solve_ff` initially parses the PDDL files and enumerates all possible actions that can be taken by the agent. Next, it performs a greedy search by adding the first action that leads to a strictly smaller h_{ff} to the plan. If no such action is found, there is a "plateau" in the graph, and BFS is used to find a sequence of actions that lead to a smaller h_{ff}. If BFS fails to find such sequence of actions, then the algorithm determines that the activity planning problem is unsolvable.

Comparing the performance of BFS and FF Heuristic Planner, on the contrary to my expectation, I found that BFS consistently runs faster ($650\mu s$ vs. $950\mu s$). This is because the plan graph is rather simple for this problem. For example, the starting state has $h_{ff} = 2$, meaning EHC gets stuck on a "plateau" for most of the time, resorting to BFS. In this case, calculating h_{ff} leads to computational overhead. For more complex problems requiring many actions to be taken, I expect the FF Heuristic Planner to perform significantly faster than BFS.

3 Motion Planning

To view the sample-based motion planner in action, see FF Heuristic Planner and RRT Trajectory Generator or go to the following url: `https://drive.google.com/file/d/1sknWGIKYpLsJexYLzeWbSfdt8TrVTznH/view?usp=drive_link`.

3.1 Environment Assumptions

Hard-coded values can be found in `constants.py` and includes:

- The initial 2D poses of the spam and sugar boxes (were not modified significantly from the original positions, simply moved the sugar box closer for the robot arm to reach)
- The 3D poses of the spam and sugar boxes after moved by the robot
- The 7D goal joint position vectors for the different actions
- `JOINT_STEP_SIZE`, `GOAL_SAMPLE`, and `GOAL_THRESHOLD` (explained in Trajectory Generation using RRT)

In addition, the following environment assumptions were made:

- The sugar/spam box attaches to the end-effector once the "grab" action finishes
- The drawer opens after the "open drawer" action ends and closes before the "close drawer" action begins

Importantly, the collision detection in my RRT implementation is simplified from traditional methods. Typically, when checking if a newly sampled point can be reached without collisions, RRT calculates whether the line between the new point and its nearest point in the tree intersects with an obstacle. However, in my implementation, I only check whether the new sampled point is inside an obstacle, meaning the robot collides with an object in the kitchen environment. This simplification can be made because the robot bodies are large and the distance between each node in the tree are small, meaning if a new joint configuration does not cause a collision, it is almost guaranteed that all joint configurations between the new point and its closest point are valid.

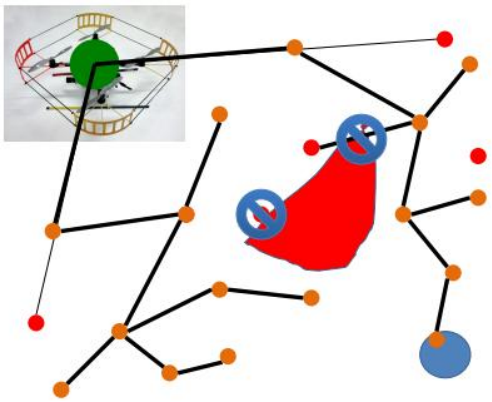

Figure 3: Collision detection in RRT

3.2 Trajectory Generation using RRT

The sample-based motion planning algorithm is written in `rrt.py`. `get_sample_fn` returns a sampling function that returns a random valid joint configuration x_{rand}. I also defined a class `Tree` which keeps track of its `parent`, `children`, and `point` (joint configuration), and has a method `add_child` which appends a new `Tree` to its `children` and sets the child's `parent` to itself.

Class `TrajectoryGenerator` contains the main motion planning algorithm in the `solve` method. It also includes four helper functions:

1. `find_nearest`: returns x_{nearest}, the node in the tree closest to x_{rand} by recursively traversing the tree.
2. `steer`: creates a line from x_{nearest} to x_{rand} limited to length `JOINT_STEP_SIZE`. Returns this new point x_{new}.
3. `obstacle_free`: returns true if the input joint configuration does not cause the robot to collide with a kitchen object.

4. `find_path`: returns a list of joint configurations from the initial state to the goal state by following the parent pointers.

RRT is then implemented as follows using the helper functions:

Algorithm 1 Rapidly-Exploring Random Tree (RRT)

```
tree ← Tree()
i ← 1
while True do
    if i  mod GOAL_SAMPLE = 0 then
        x_rand ← x_goal
    else
        x_rand ← sample_free()
    end if
    x_nearest ← find_nearest(tree, x_rand)
    x_new ← steer(x_nearest, x_rand)
    if obstacle_free(x_new) then
        x_last ← Tree(x_new)
        x_nearest.add_child(x_last)
        if dist(x_new, x_goal) ≤ GOAL_THRESHOLD then
            return find_path(x_last)
        end if
        i ← i + 1
    end if
end while
```

Instead of getting a random sample every time, the algorithm samples the goal state every `GOAL_SAMPLE` cycles to bias the tree. Additionally, we determine whether x_{new} is in the goal state by checking if the euclidian distance between x_{new} and x_{goal} is less than `GOAL_THRESHOLD`.

While the algorithm successfully generates a trajectory as shown in the video, it can be slow depending on the initial and goal state. Constants needed to be tuned to increase the performance of the algorithm.

3.3 Integration with the Activity Planner

The `robot.py` file includes a `Robot` class with methods for each action in the PDDL domain in addition to methods to move the robot base and arm to the initial state. Dictionary `function_map` maps PDDL actions to its corresponding methods, and the `act` method uses `function_map` to perform a task based on a given action.

The overall plan execution is written in `plan_executor.py`. The code initially solves the activity planning problem using the FF Heuristic Planner, which returns a list of tasks to execute. `plan_executor` simply iterates through this list and calls the `Robot().act` method for each action to fully execute the plan.

4 Trajectory Optimization

To view the trajectory optimization in action, see Trajectory optimization or go to the following url: https://drive.google.com/file/d/1ZB1DUmpl6Mqf19IPNjAWVvW_RhoJtXB6/view?usp=sharing.

4.1 File organization

Trajectory optimization is implemented in pydrake using the default IPOPT nonlinear optimization solver, and can be found in `trajopt.py`. The `solve` function in the `TrajectoryOptimizer` class solves the optimization problem defined in Optimization problem definition and takes an initial guess of the arm trajectory as an optional argument to seed the solver. Furthermore, the `Robot` class constructor has an additional `use_trajopt` argument which allows the use of `TrajectoryOptimizer` to generate the arm trajectory. Finally, the trajectories generated from the sample-based motion planner were saved as `.npy` files in the `npy` folder. To use these trajectories as inputs to the IPOPT solver, I added a new `Robot().get_init_guess` method which loads a `.npy` file and linearly interpolates `PATH_LENGTH` sample points. The solver generates a trajectory encoded as a `PATH_LENGTH` $\times 7$ matrix, so the initial guess must also be of the same size.

4.2 Optimization problem definition

The kinematic trajectory optimization formulation is inspired from the problem used in Professor Russ Tedrake's Robotic Manipulation course [1]:

$$\begin{aligned} \min_{T} \quad & T, \\ \text{subject to} \quad & X^{G_{start}} = f_{kin}(q(0)), \\ & X^{G_{goal}} = f_{kin}(q(T)), \\ & \forall t, \quad |\dot{q}| \leq v_{max}, \end{aligned}$$

Where f_{kin} is the forward kinematics function and $q(t)$ is the joint state of the robot at time t. The problem reads as minimizing time T for the arm to move from the start pose $X^{G_{start}}$ to the goal pose $X^{G_{goal}}$ while satisfying velocity constraints.

There are a few challenges when converting this problem into a pydrake `MathematicalProgram`. Since pydrake does not allow pybullet functions to be included in its problem formulation, I simplified the problem to input initial and goal joint states rather than poses, and did not encode collision detection. Moreover, pydrake does not allow indexing into a variable using another variable. To avoid this problem, I hard-coded the length of the arm trajectory to be `PATH_LENGTH` $= 200$ defined in `constants.py`. Finally, I approximated the joint velocities as the following: $\dot{q} \approx \frac{q(t+1)-q(t)}{T/\texttt{PATH\_LENGTH}} \quad \forall t \in \{0, \ldots, T-1\}$. Bringing it all together, the new optimization problem is the following:

$$
\begin{aligned}
\min_{T} \quad & T, \\
\text{subject to} \quad & q_0 = q[0], \\
& q_f = q[T], \\
& \forall t \in \{0, \ldots, T-1\}, \quad \frac{q[t+1] - q[t]}{T/\texttt{PATH\_LENGTH}} \leq v_{max}, \\
& \forall t \in \{0, \ldots, T-1\}, \quad \frac{q[t+1] - q[t]}{T/\texttt{PATH\_LENGTH}} \geq -v_{max}
\end{aligned}
$$

where q_0 is the initial joint state and q_f is the goal joint state. When running the program, the generated trajectories were smooth but inefficient as the solver found solutions with nonoptimal local minima. While this was alleviated by seeding the solver with initial guesses, it caused the optimized trajectories to be jagged since the given RRT trajectories exhibited a "shaking" motion. To prevent such trajectories, I added another set of constraints to limit the acceleration of the joints:

$$
\begin{aligned}
\forall t \in \{0, \ldots, T-2\}, \quad & \frac{(q[t+2] - q[t+1]) - (q[t+1] - q[t])}{2T/\texttt{PATH\_LENGTH}} \leq a_{max}, \\
\forall t \in \{0, \ldots, T-2\}, \quad & \frac{(q[t+2] - q[t+1]) - (q[t+1] - q[t])}{2T/\texttt{PATH\_LENGTH}} \geq -a_{max}
\end{aligned}
$$

Compared to the sample-based motion planner, the end result was smoother and more controlled trajectories but slower computation due to the greater number of constraints. Although some actions continued to cause the arm to move inefficiently, this is because the chioces of the goal joints can be improved. Implementing a custom forward kinematics function compatible with pydrake and setting the goal state as $X^{G_{goal}} = f_{kin}(q(T))$ will allow the program to find more optimal trajectories. Finally, to implement collision avoidance, I would encode the kitchen environment in pydrake and prevent the end-effector from colliding with the environment. However, I expect adding these two sets of constraints to significantly slow down computation.

5 Conclusion

Using PDDL and the Fast-Forward heuristic planner is an efficient way to encode and solve an activity planning problem. Although the benefits of using the FF heuristic was unclear in this paper, it can improve performance when solving more complicated problems without many heuristic "plateaus". Both the sample-based motion planner and nonlinear constraint optimization are valid ways to execute the generated plan. However, constraint optimization, with its ability to precisely encode robot dynamics, yielded a smoother and more optimized trajectory compared to the RRT implementation. Using complex algorithms such as RRT* may result in finding an optimal trajectory, but the required computation power may be too high. Similarly, adding more constraints to encode collision avoidance will result in slower computation for the nonlinear optimizer. For both activity and motion planning, it is essential to consider the compromise between runtime and finding a near-optimal solution, and choose the solution that benefits the program the most.

References

1. Tedrake R. Robotic Manipulation Chapter 6: Motion Planning. 2023. URL: https://manipulation.csail.mit.edu/trajectories.html#section2 (visited on 12/08/2023).