Lab 2 - Task and Motion Planning

Task and motion planning was covered by a guest lecturer on Tuesday 11/6/2018. The slides from the lecture are posted on Piazza.

Assignment

The objective of this assignment is to implement a planning system that enables the robot to automatically plan to pick a brick and place it within the cupboard. A complication is that left cupboard door is initially closed, preventing the robot from safely placing the brick. Thus, the robot must first open the left door before manipulating the brick.

The complete goal conditions are that:

- The brick is at a fixed world pose on the left side of the second cupboard shelf
- The robot finishes at its initial configuration

An example solution to this problem is available on Youtube:

We will model this task as a PDDLStream planning problem and solve it using an existing PDDLStream algorithm (a variation of the focused algorithm). A large fraction of the required functionality has already been implemented. This includes the initial state and goal formula encoding as well as action descriptions, stream descriptions, and stream generator implementations for the move, pick, and place actions. The only missing components are the action description, stream description, and stream generator implementation for the pull action, which is used to open cupboard doors. Your job is to implement these missing components. This requires completing:

- The description for the `pull` action in `domain.pddl`
- The description for the `plan-pull` stream in `stream.pddl`
- The implementation of the `get_pull_fn` procedure in `generators.py`

You should be able to complete this assignment without any further modifications to the codebase. The procedure `get_pull_fn` takes in a `Task` and returns the generator function `fn(robot_name, door_name, door_conf1, door_conf2)` that corresponds to `plan-pull`. The generator function itself has inputs corresponding to the:

- Name of the robot performing the pull (`robot_name`)
- Name of the door being pulled (`door_name`)

- Configuration of the door at the start of the pull (`door_conf1`)
- Configuration of the door at the end of the pull (`door_conf2`)

It outputs a Python generator which yields output triplets that each give a corresponding:

- Configuration of the robot at the start of the pull (`robot_conf1`)
- Configuration of the robot at the end of the pull (`robot_conf2`)
- Combined robot & door trajectory for performing the pull (`traj`)

Implementing `get_pull_fn` requires computing combined robot and door paths that move the door from `door_conf1` to `door_conf2` .

To run the planner, execute the following in a docker bash container with drake tag `drake-20181110` :

```
$ ./run.py
```

So for example, if you are running on Mac, you would use these commands to run the planner:

```
docker/docker_run_bash_mac.sh drake-20181110 lab_2/lab2_pddl/
./run.py
```

Please upload your completed version of `domain.pddl` , `stream.pddl` , `generators.py` to Gradescope. There are no Gradescope tests for this assignment. We will evaluate the assignment through a checkoff during lab hours.

Additional Information

When prototyping your solution, it may be helpful to run a version of the planner that ignores collisions. Doing this temporarily simplifies the problem and as a result decreases planning time. This functionality is enabled by executing:

```
$ ./run.py -c
```

Our implementation uses random number generators when performing inverse kinematics and sampling-based motion planning. As a result, the solution and runtime for each execution may vary. We recommend running the planner several times to get a sense for the average performance. In our experiments, the planner was able to find a solution within at most 30 seconds. Additionally, the planner can be run in an (almost) deterministic setting by executing:

```
$ ./run.py -d
```

Generating trajectories that successfully open a door may require multiple invocations of `InverseKinematics` with different initial guesses.

Solutions are currently visualized by manually visualizing the planned sequence of world states rather than using drake's physics `Simulator`.

PDDLStream is an ongoing research project and thus the software is still in an experimental state. While some error checking functionality is implemented, not all cases are necessarily covered. Thus, some incorrect specifications may cause the planner to crash or fail to find a solution.

For a tutorial on PDDL, visit the following page: http://users.cecs.anu.edu.au/~patrik/pddlman/writing.html (http://users.cecs.anu.edu.au/~patrik/pddlman/writing.html)

For more PDDLStream examples, visit its homepage on Github: https://github.com/caelan/pddlstream/ (https://github.com/caelan/pddlstream/tree/master)

Movable body collision checking is modeled using external predicates (`:predicate`) instead of test streams. The external predicates `TrajPoseCollision` and `TrajConfCollision` return `True` if and only if there is a collision between a trajectory and a body at a particular pose or configuration.

Overview of Files

The file `run.py` constructs and solves the PDDLStream planning problem. It loads a manipulation `Task`, converts it into a PDDLStream problem, solves it using `solve_focused`, converts the plan into a sequence of position trajectories, and visualizes the solution using `meshcat`.

The file `domain.pddl` specifies the predicates and actions present within the PDDLStream problem. Its syntax is identical to the standard PDDL domain-file syntax.

The file `stream.pddl` contains a declarative specification of the streams used. Its syntax is inspired by PDDL syntax.

The file `generators.py` gives the Python implementation for the streams declared within `domain.pddl`. This includes `sample-grasp`, `plan-ik`, `plan-pull`, and `plan-motion`.

The file `problem.py` contains the `Task` class which manages the drake system, encodes the initial problem state, and describes the problem goal conditions.

The file `motion.py` contains procedures for performing sampling-based motion planning within drake.

The file `simulation.py` contains methods for (1) displaying position trajectories by stepping through each position in the trajectory, (2) converting a sequence of position trajectories into a sequence of splines and gripper setpoints, and (3) simulating the system using `manipulation_station/manipulation_station_plan_runner.py`.

The file `systems.py` builds the `ManipulationStation` system `Diagram`.

The file `iiwa_utils.py` contains several KUKA iiwa specific utilities for opening and closing the W50 gripper and generating grasp transformation for boxes.

The file `utils.py` contains an assortment general-purpose drake Python utilities.

The directory `manipulation_station/` contains a controller for simulating the planned trajectories.

The directory `pddlstream` contains the current implementation of the PDDLStream framework and algorithms.

Back to Lab 2 Main (../lab_2.html)