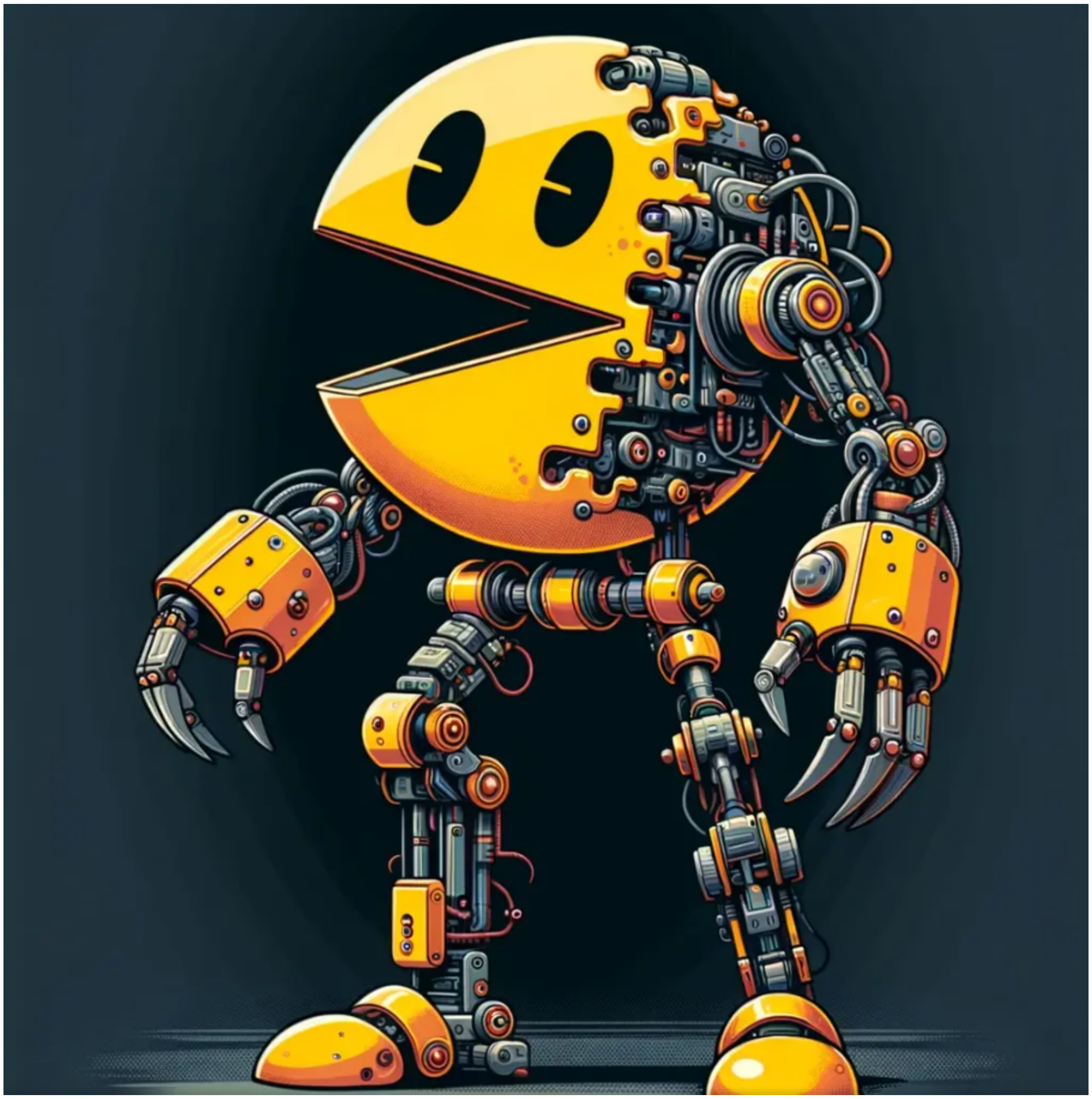
Homework 6: Planning

Due: May 6, 11:59pm

Late policy: 3%/day, see course webpage for details



Robotic Pacman / A task and motion planner / Can plan far ahead

Introduction

In this homework you will implement planners for a simulated robot. You will program a high level task planner based on PDDL, and a low level motion planner based on RRT. You will also combine them in a task-and-motion-planning (TAMP) system.

As in previous programming assignments, we include an autograder for you to grade your answers on your machine. This can be run with the command `python autograder.py`.

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore. You can download all the code and supporting files as a zip archive from canvas.

Files you will edit and Submit

You will edit and upload these to Gradescope. (Please *do not* change the other files in this distribution or submit any of our original files other than these files.)

`pddl_plan.py`	task planner
`rrt.py`	motion planner
`demo_too_narrow.py`	demo file that will contain a simple answer to a question about hierarchical planning

Files you might want to look at

`search_graph.py`	code for search RRT search graphs, particularly the classes `SearchGraph`, `Edge`, and `Node`
`motion.py`	code for `Path` class

Software environment

IMPORTANT! This homework requires installing extra packages. Assuming that you have been using an anaconda environment called 4700, please run the following:

```
conda activate 4700
python -m pip install shapely pyYAML pycollada scipy PyQt5 adjustText
```

For creating the initial anaconda environment, see Ed Discussions pinned post at the top titled 'Python environment/dependencies". The autograding server, which decides on your actual grade, runs python 3.12, so you are strongly encouraged to use that version to ensure compatibility. Technically, you can still run the code using a newer python version, but take care to double check that the autograding server on gradescope is reporting the same final grades as what you see on your local machine.

Evaluation

Your code will be autograded for correctness. Please *do not* change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. You should regard the Gradescope autograder's judgments *on the server* as your final grade, *not* what you see locally on your laptop (though ideally these agree). Remember that you can submit as many times to the server as you want before the deadline in order to make sure that the code works correctly on the grading server. In unusual circumstances we will manually run your code, for example if we suspect grade-server hacking, but by strong default, what the server says is your final grade.

Academic Dishonesty

We will be checking your code against other submissions in the class for logical redundancy. If you copy someone else's code and submit it with minor changes, we will know. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help

You are not alone! If you find yourself stuck on something, contact the course staff for help. Office hours and the discussion forum are there for your support; please use them. We want these projects to be rewarding and instructional, not frustrating and demoralizing. But, we don't know when or how to help unless you ask.

Ed Discussion: Please be careful not to post spoilers. Make private posts if sharing your code or explaining your attempted solution, and a public post if asking a question that does not expose your solution.

Part 1: Task Planning in PDDL

First, read the source code for some important classes for PDDL. Open `pddl_plan.py`, and you will see a class called `Fluent`. Remember that a Fluent is a *proposition* about the state of the world, and that which fluents are true can change over time. (Think of a fluent as a binary flag.) Each fluent has a name, followed by arguments, such as `Fluent("holding", "?object", "?robot")`. In this example the arguments are variables, because they start with a `?`, meaning that you could instantiate this fluent in different ways depending on what the robot and object are. You could also have a fluent such as `Fluent("on_top_of", "red_block", "blue_block")`, whose arguments are not variables: this fluent means that in the current state, a specific red block is on top of a specific blue block.

Next, look at the class called `Action`. An action is a single step in a plan, and like `Fluent`, it too takes arguments. The arguments to an action correspond to objects in the world, such as a robot or a table, and each object has a *type*. For example, we might have an action called `"move"`, and to move `"robot5"` to `"table7"`, we would pass `"robot5"` and `"table7"` as arguments to the move action, specifying that the first argument is of *type* `"robot"` and the second argument is of *type* `"thing_we_can_move_to"`. To specify what it means to `"move"` we need to give preconditions and postconditions to the action, which are going to be `Fluents`. The precondition checks if the action is something that we can actually do in a state, and the postcondition changes the state by adding and deleting fluents. Precondition checks work by verifying if fluents are true/false about a state. The real power of these actions come from putting variables into their definitions: These so-called *lifted actions* can have their variables instantiated in any way consistent with the current state, generating a wide range of *ground actions* that the planner can search over; see the `groundings` method in `Action` and `replace` in `Fluent`.

For example, here is an example action for moving between adjacent rooms:

```python
move_action = Action('move',
 ['?agent', '?from', '?to'], # args are variables => lifted action
 ['robot', 'room', 'room'], # types of each argument
 # precondition: fluents that must be true/false before the action can be executed
 [('+', Fluent('at', '?agent', '?from')),
  ('+', Fluent('adjacent', '?from', '?to')),
  ('-', Fluent('at', '?agent', '?to'))],
 # postcondition: fluents that will be true/false after the action is executed
 [('+', Fluent('at', '?agent', '?to')),
  ('-', Fluent('at', '?agent', '?from'))])
```

The above lifted action, in a given state, has a set of *groundings*, each of which is a ground action (no variables) that we might actually do in that state, and which can have its preconditions checked and its postconditions applied:

```python
ground_actions = move_action.grounding(state)
for a in ground_actions:
    if a.check_pre_condition(state):
        print("We could do action", a, "in this state.")
        print("This would transform the state to be:", a.apply_post_condition(state))
```

You can see in the `Planner` class how this can be used to build a graph search algorithm, (in this case BFS), but we won't have you repeat what you did on Homework 1, so you don't have to implement any of that. Instead, focus on making sure that you understand the core data structures and ideas behind planning, reviewing the code and the lecture slides as appropriate, before you proceed to the first question.

Question 1

You will be defining PDDL actions that work with a more fine-grained representation of locations and objects, which will make it more amenable to TAMP because the actions will be closer to the world geometry. We differentiate between *rooms*, which contain *spawns*, which are locations that objects can be placed, and where *objects* might initially be. A *robot* is always in one room, and it might also be *oriented* toward a particular spawn within that room. When oriented toward a spawn, the robot can issue pick and place commands to either pick up or place down an object. The robot can only hold at most one object at a time.

The following fluents describe the robot state:

`robot_in_room(?robot,?room)`	Whether a particular robot is in a particular room
`empty(?robot)`	Whether the robot has an empty gripper for holding objects
`oriented(?robot)`	Whether the robot is oriented toward (any) spawn location
`holding(?object,?robot)`	Whether the robot is holding a particular object

These fluent describe the relative locations of objects, spawns, and rooms:

`spawn_in_room(?spawn,?room)`	Whether a spawn location is located within a particular room
`adjacent(?room,?room)`	Whether a pair of rooms are adjacent

This fluent relates both objects and robots to spawn locations:

`at_spawn(?object_or_robot,?spawn)`	Whether a particular object or robot is at a spawn location

What code you write: Implement the following actions:

`transit(?robot,?from_room,?to_room)`	When not oriented, move robot from room to *adjacent* room	code in `create_action_transit`
`recenter(?robot,?spawn,?room)`	If the robot is oriented at a spawn location in a room, unorient by moving to the middle of the room	code in `create_action_recenter`
`pick(?robot,?object,?spawn)`	Pick up an object when both the robot and the object are at the same spawn. Robot can only hold one thing at a time. The object will no longer be at the spawn, because it will be on the robot.	code in `create_action_pick`
`place(?robot,?object,?spawn)`	Place down a held object on the target spawn location	code in `create_action_place`

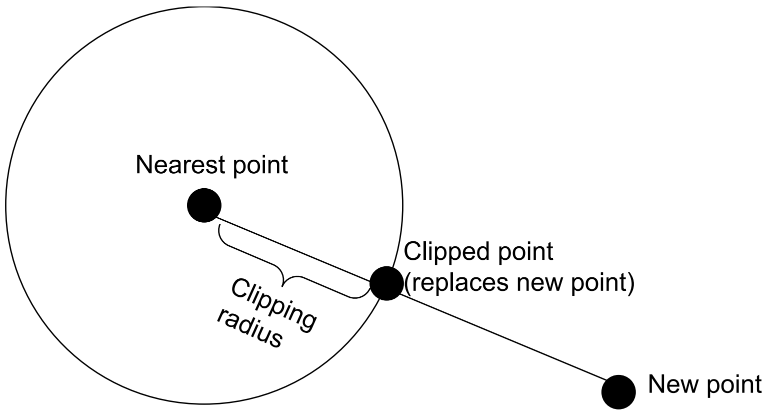
Check out `create_action_orient` for an example of what this should look like for the `orient` action.

Run `python pddl_plan.py` to see a demo of your working code. You can also run the corresponding autograder: `python autograder.py -q q1`

Part 2: Motion Planning with RRT

Next, you are going to implement a low level motion planner using RRT. Look at `RRTPlannerPolygon`

You're going to implement a simple but important optimization: When trying to add a new point to the growing RRT graph, we will clip its distance to the nearest node in the graph so far. Equivalently, you can think of this as drawing a line between the new point and its closest neighbor in the old graph, and then moving the new point so that it is within some clipping radius of its closest neighbor. This is illustrated below.



After performing clipping, the original new point is discarded and the clipped point is used in its place.

The effect of this optimization is to make smaller jumps in configuration space, which means that each new point is less likely to intersect with obstacles, so new randomly generated points have a much higher chance of being incorporated into the graph.

Question 2

Implement RRT by filling in the `plan` method in `RRTPlannerPolygon`. You will need to use the following methods:

`self.sample_configuration`	samples a random configuration in the state space
`self.graph.add_edge`, `self.graph.add_node`	extends RRT graph, see `search_graph.py`
`self.graph.nearest`	returns the node closest to a configuration
`self.is_connectable`	returns if two nodes can be connected
`self.make_clipped_node`	takes the nearest node and a candidate new configuration; clips distance and returns replacement new node
`self.try_connect`	tries to connect a pair of graph nodes (such as a new node to the goal node)

Be mindful of the difference between robot configurations and graph nodes. Nodes are wrappers around configurations and contain pointers and other bookkeeping to track their connectivity in the growing graph, and also track their distance from the start state. See `search_graph.py`.

To visualize the trajectories that your planner produces, run `python demo_rrt.py`. Press the continue button at the top to run the plan.

You can check the correctness of your actions by running the autograder: `python autograder.py -q q2`

Question 3

Now you will implement RRT*. First, add some code in the `plan` method (where you were just editing) so that after you add the new (clipped) node, you check if `self.rrt_star` is `True`, and if so, call the method `self.rewire_node`. You will have to implement `self.rewire_node`.

To visualize the trajectories that your planner produces, run `python demo_rrt_star.py`. Press the continue button at the top to run the plan. You should notice that the path is more efficient compared to `demo_rrt.py`.

You can check the correctness of your actions by running the autograder: `python autograder.py -q q3`

Question 4

Finally you will implement one last flourish on RRT* known as RRT-Connect. The idea behind RRT-Connect is to try harder to connect to toward goal: in addition to checking if the newly added node can be directly connected to the goal, we are going to repeatedly march toward the goal and add new nodes as long as we can.

Within the method `try_connect` you should see a block that you can fill out which is guarded with `if self.rrt_connect`. Fill in that code to implement this strategy:

1. Make a new node in between the current node and the goal by clipping the line that connects them. This way the new node is close to the current node, but it inches a small amount toward the goal.
2. Check if the current node and the new node can be connected. If so, connect them together.
3. The current node is updated to be the new node, and the process repeats until new connections cannot be formed.

Generally RRT-Connect does not find highly more efficient paths, but it does find them quicker, because it focuses more on making progress toward the goal.

You can run `python demo_rrt_connect.py` to see an example path. Visually you should be able to see that it tries to tunnel in a line toward the goal whenever it can. (Don't compare the absolute path length to the earlier RRT demos: We tweaked hyperparameters to make the tunneling effect more prominent.)

You can check the correctness of your actions by running the autograder: `python autograder.py -q q4`. The autograder checks that enabling RRT-Connect allows finding solutions in fewer search iterations.

Part 3: Task and Motion Planning

Now you will see what happens when you put together your task planner and your motion planner! We have provided a basic TAMP system that combines both of your planners. It works by generating a high level task plan, and then refining it into a motion plan by *sampling*, meaning that it generates a concrete low level state for each high level intermediate state, and then asks the motion planner to link together these low level states. You can think of the TAMP system as implementing this logic:

```
low_level_state = get_low_level_state() # get current state
high_level_state = abstract(low_level_state) # get the corresponding high level fluents

high_level_plan = PDDL_planner(high_level_state, goal_state) # from part 1
# high_level_plan is a list of high level states
# first is the current state
# last is the goal state

# now we need to refine into a low level (motion) plan
motion_plan = [] # list of low level states
for abstract_state in high_level_plan[1:]:
    # generate a low level state consistent with this abstract state
    new_low_level_state = sample_low_level_state(abstract_state)
    new_motions = RRT(low_level_state, new_low_level_state) # from part 2
    motion_plan.extend(new_motions)
    # next motion plan starts where this one left off
    low_level_state = new_low_level_state
# now we are done with refining the high level plan!
# motion_plan should contain a feasible trajectory accomplishing the task

# helper function, you don't need to implement this or think about its details
def sample_low_level_state(abstract_state):
  # sample a low level state that is consistent with the abstract state
  # in actuality, this is done deterministically
  # real TAMP systems usually do this stochastically
  ...
```

Fruit Mischief. Run a basic test of TAMP. The planner will figure out a way of moving an apple and a banana into the bathroom, then head to the bedroom to await further instructions. When you run this command, you should see the robot successfully putting fruit where it doesn't belong:

```
python demo_tamp.py
```

Second, run a more difficult test. The robot has the same mission, but for some reason the planner, is not going to be able to find a solution:

```
python demo_too_narrow.py
```

Question (1 pt)

You should see that the second test failed, but the first succeeded. Interestingly, task planning succeeded for the difficult test, but motion planning failed. This is a known challenge with TAMP, for which there exist effective solutions, but which we are not asking you to solve on this homework. Instead we're just going to quiz whether you remember the relevant hierarchical planning terminology.

When the second test failed, it was because of a failure of a critical property of hierarchical planning that does not hold in our situation. What is the name of that critical property? Please enter your answer in `demo_too_narrow.py` in the function `q5_answer`. The autograder will check if you got it right: You should be able to find it by reviewing the lecture slides if you did not make it to class.

Credits

The robot simulator and RRT geometry code comes from the open source library pyrobosim. The verbiage about course policy is lightly adapted from Berkeley CS188. Cornell TAs Sikai Shen and Spencer Dunn assisted significantly in developing this assignment.