Multi-Robot Systems: Control, Communication, and Security
Professor: Stephanie Gil
CS 286 - Spring 2022
Assignment Leads (TFs): Henry Cerbone, Ninad Jadhav
Due: Wednesday, April 6, 2022 at 11:59pm EDT

Programming Assignment 4

Instructions

Collaboration: Collaboration is allowed and encouraged, but you must write everything up by yourselves. You must list all your collaborators. MAX group size is 3 persons.

References: Please consult the assigned class readings listed below while completing this assignment. *Note that it is not permitted to use any existing code base unless provided by us or explicitly stated otherwise.* We strongly encourage you to explicitly cite additional reference material, readings where applicable. Standard libraries (matplotlib, numpy, scipy, pandas) are also acceptable.

Submission: Submit your solutions on Canvas in a zipped folder by the indicated due date.

Papers: Refer to the Bertsekas paper [1] from class for reference.

Dynamic programming basics

[1] We consider N stage dynamic programing problem (DP), where state x_k at time k evolves to state x_{k+1} after applying a control u_k as follows,

$$x_{k+1} = f(x_k, u_k, w_k), \tag{1}$$

where w_k is the disturbance is given by the probability distribution $P(.|x_k, u_k)$, which only depends on x_k and u_k, but not on values of prior disturbances $w_{k-1}, \dots, w_0$.

The stage cost is $g_k(x_k, u_k, w_k)$ and the cost of a policy $\pi = \{\mu_0, \dots, \mu_{N-1}\}$ starting at state x_0 is given by

$$J_\pi(x_0) = E\Big\{g_N(x_N) + \sum_{k=0}^{N-1} g_k(x_k, \mu_k(x_k), w_k))\Big\}. \tag{2}$$

In a multiagent DP, with m agents, the set of controls available to a taxicab ℓ is denoted by U^ℓ, where $\ell = \{0, \dots, m-1\}$. The set of controls available to all taxicabs is denoted by $U = (U^0 \times U^1 \times \dots \times U^{m-1})$.

[1]The description given here is taken from the paper[1]

1 Standard rollout

The rollout policy is denoted by $\widetilde{\pi} = \{\widetilde{\mu}_0, \ldots, \widetilde{\mu}_{N-1}\}$, where each control function

$$\widetilde{\mu}_k(x_k) = \min_{u \in U} E\Big\{g(x_k, u, w_k) + J_\pi(f(x_k, u, w_k))\Big\} \tag{3}$$

Each stage of the standard rollout takes $O(C^m)$ computations, where $C = \max\{|U^0|, |U^1|, \ldots, |U^{m-1}|\}$.

2 One-agent-at-a-time rollout

Here, m sequential minimizations are performed, one for each agent. The rollout policy is denoted by $\tilde{\pi} = \{\tilde{\mu}_0, \ldots, \tilde{\mu}_{N-1}\}$, where each control function $\tilde{\mu}_k = (\tilde{\mu}_k^0, \ldots, \tilde{\mu}_k^{m-1})$ has m component, $k = \{0, \ldots, N-1\}$.

$$\begin{aligned}
\tilde{\mu}_k^0(x_k) = \min_{u^0 \in U^0} E\Big\{& g(x_k, (u^0, \mu_k^1(x_k), \ldots, \mu_k^{m-1}(x_k)), w_k) \\
&+ J_\pi(f(x_k, (u^0, \mu_k^1(x_k), \ldots, \mu_k^{m-1}(x_k)), w_k))\Big\} \\
\tilde{\mu}_k^1(x_k) = \min_{u^1 \in U^1} E\Big\{& g(x_k, (\tilde{\mu}_k^0(x_k), u^1, \mu_k^2(x_k), \ldots, \mu_k^{m-1}(x_k)), w_k) \\
&+ J_\pi(f(x_k, (\tilde{\mu}_k^0(x_k), u^1, \mu_k^2(x_k), \ldots, \mu_k^{m-1}(x_k)), w_k))\Big\} \\
&\cdots \\
\tilde{\mu}_k^{m-1}(x_k) = \min_{u^{m-1} \in U^{m-1}} E\Big\{& g(x_k, (\tilde{\mu}_k^0(x_k), \ldots, \tilde{\mu}_k^{m-2}(x_k), u^{m-1}), w_k) \\
&+ J_\pi(f(x_k, (\tilde{\mu}_k^0(x_k), \ldots, \tilde{\mu}_k^{m-2}(x_k), u^{m-1}), w_k))\Big\}
\end{aligned} \tag{4}$$

Each stage of the one-agent-at-a-time rollout takes $O(Cm)$ computations.

Taxicab pickup problem

In this assignment, you will implement the taxicab routing problem, beginning with simulating a base policy for the taxicabs and extending to single and multi-agent rollout. The rollout method gives a suboptimal policy, that improves the cost of a base policy (a simple/heuristic based policy) using a lookahead optimization. We have prepared a dataset data/requests_details_day1.csv for this problem. This is a smaller dataset prepared from the San Francisco taxicab dataset: `http://crawdad.org/epfl/mobility/20090224/index.html`. Your goal is to find a control policy that gives a requests pickup ordering for the agents. An optimal policy will result in minimizing the wait time of all requests. The taxicab pickup problem is formulated as a finite horizon Dynamic Programming (DP) problem.

The locations are discretized into a 5×4 grid. The grid will use $(0, 0)$ index for the upper left corner. A position $(2, 1)$ is 2 to the right and 1 to the down from the upper left corner. Moving the to the left, right, up, and down would move it to locations $(1, 1), (3, 1), (2, 0)$ and $(2, 2)$, respectively. Each line in the 'requests_details_day1.csv' file gives the time of request generation, pickup location's x and y coordinates (in the 5×4 grid), dropoff location's x and y coordinates (in the 5×4 grid). The taxis can either move in all its five neighboring locations in the grid, denoted by $0, 1, 2, 3$, and 4 (Stay, Left, Right, Up, and Down, respectively.) If a request's pickup location is the same as the taxicab, the taxicab can also choose to pick up the request ($5 + s$ denotes the control if an agent chooses to pick up sth request in the request list[2]). We use a base policy that directs an agent towards the nearest pickup location or picks up a request if its location is the same as the pickup location of the request. At each stage, there are some known requests and in the next stage, one/more request can be stochastically generated using the pickup and dropoff location distributions. See the section below for more details about the states, controls, and state transition functions.

3 DP formulation for the taxicab pickup problem with m agents:

3.1 State

This is given as the class **state** to you inside folder 'util' of the base code. The state x_k consists of

1. *agent_locations*: A list of size m. ℓth element of the list is the position of agent ℓ denoted by a pair which consists of the x and y coordinates in the grid.

2. *time_left_in_current_trip*: A list of size m. ℓth element of the list is the time (in minutes) left in the current trip of agent ℓ.

3. *outstanding_requests*: A list. sth element of the list is a request, denoted by a tuple of *pickup_location* (a pair that consists of the x and y coordinates in the grid) and *dropoff_location* (x and y coordinates in the grid).

3.2 Controls:

You are given the function **available_controls_to_agent**. Set of all possible controls components for an agent is (0 : Stay, 1 : Left, 2 : Right, 3 : Up, 4 : Down, 5 : pickup request 0, 6 : pickup request 1, ...). The control set U^ℓ, available to agent ℓ consists of the following.

1. If the taxi is not occupied ($time\_left\_in\_current\_trip == 0$), then
   the control component can include $(0, 1, 2, 3, 4)$, which means the agent can move to one of the 5 neighboring locations, except the corner grid locations.

2. If the taxi is not occupied ($time\_left\_in\_current\_trip == 0$), and
   if $outstanding\_requests[s].pickup\_location == agent\_location[\ell]$, then
   the control component includes $s + 5$, i.e., pick up request s at the agent's current location.

3. If the taxi is occupied ($time\_left\_in\_current\_trip > 0$), then
   no controls available; agent moves one-step towards finishing the current dropoff.

We consider a 1-minute travel time between any two adjacent cells. Hence time to travel from one grid location to the other is the Manhattan distance between the two locations.

[2]Since we represent controls by integer values, we use $0, 1, 2, 3, 4$ to represent directional motion and $5, 6, \ldots$ to represent picking up the first request, second request and so on.

3.3 State transition:

You are given the function **next_state**, which takes the state object and the control as input and outputs the stage cost. This function modifies the input state object to generate the next state. Note that if you are using next_state(state_object, control) and want to use the state_object later on. Please save a copy (e.g., deepcopy). At stage k the state transition function f in Equation 1 does the following,

1. For agent ℓ, if the agent is occupied($time\_left\_in\_current\_trip[\ell] > 0$), time in the current trip reduces by 1 minute (i.e., $time\_left\_in\_current\_trip[\ell] -= 1$).
2. For agent ℓ, if the control component u_ℓ gives the direction of movement (i.e., Stay, Left, Right, Up, Down), the agent makes the move to reach the next location (i.e., update $agent\_locations[\ell]$ accordingly).
3. For agent ℓ, if the control component is to pickup sth request (i.e., $u_\ell == pick\_a\_request(s)$) and no other predecessor agent wants to pickup the request (i.e., $u_\kappa \neq pick\_a\_request(s), \kappa = \{0, \ldots, \ell-1\}$), then agent ℓ picks up the request by setting it's location to the request's dropoff location and updates its time left in the current trip. More formally, we set $agent\_locations[\ell] = outstanding\_requests[s].dropoff\_location$, and $time\_left\_in\_current\_trip[\ell] = Manhattan\_distance(outstanding\_requests[s].pickup\_location, outstanding\_requests[s].dropoff\_location)$
4. Sample a pickup location $pick\_loc$ from the distribution $pickup\_probability$ and sample a dropoff location $drop\_loc$ from the distribution $dropoff\_probability$. Append $(k, pick\_loc, drop\_loc)$ to the $outstanding\_requests$ list.
5. The stage cost $g_k(x_k, u_k, w_k)$ in this problem has a simplified form, $g(x_k)$, which gives the number of outstanding requests at stage k, $k = \{0, N-1\}$. The terminal stage cost $g_N(x_N) = 0$. The **next_state** function returns the size of list $outstanding\_requests$ as the stage cost.
6. Once all the pickups are finalized, any outstanding requests picked up by the agents in time k are removed from the list.

3.4 Cost function:

The cost function of policy π starting from a state x_0 is the total time the requests wait (in minutes). Consider the length of the horizon $N = 10$. The Equation 2 takes the following form,

$$J_\pi(x_0) = E\Big\{\sum_{k=0}^{9} g(x_k)\Big\}.$$

The expectation is computed by running multiple Monte-Carlo simulations from state x_0. The code for running a single Monte-Carlo simulation is given in function **MC_simulation** (which gives the summation inside the expectation). This function calls **next_state** function 10 times to find the total cost of a MC trajectory. The code for calculating the expectation in the cost function is given in function **average_MC_simulations**

We have provided the base code for each question and given you the main function corresponding to each question. The project folder structure is summarized in below in the subsection 4. You're free to make additions to any question of this code.

4 Folder structure

The code given to you has the following structure.

```
skeleton_code
 |__data   # Input and outputs
 |   |__requests_details_day1.csv   # input to your code
 |   |__trajectory_file
 |       |__base_policy_example.mov   # example trajectory movie
 |__src   # Main code base
 |   |__algorithms.py   # Finish functions in standard_rollout, one_at_a_time_rollout
 |   |__base_policy.py   # Finish functions for base policy
 |   |__get_disturbance.py   # Finish functions for pickup and dropoff distributions
 |   |__main.py   # Starting point of your code
 |__util   # Helper code base
     |__state.py   # Defines state, control, next state transitions
     |__globals.py   # Defines global variables
     |__graphics.py   # Visualization code
     |__request.png   # Picture of a request
     |__taxi.png   # Picture of an unoccupied taxi
     |__taxi_in_transit.png   # Picture of an occupied taxi
```

The input/output of your code should be placed in the skeleton_code/data folder.

```
skeleton_code
 |__data   # Input and outputs
     |__requests_details_day1.csv   # input to your code
     |__pickup_distribution.csv   # output of question 1, input of questions 3,4,5,6,7,8
     |__Pickup distribution.png   # output of question 1
     |__dropoff_distribution.csv   # output of question 2, input of questions 3,4,5,6,7,8
     |__Dropoff distribution.png   # output of question 2
     |__trajectory_file
         |__requests.mov   # output of question 3
         |__requests.csv   # input of questions 4,5,6,7,8
         |__requests/frame[0-10].png   # output of question 3
         |__base_policy_1.mov   # output of question 4
         |__base_policy_1/frame[0-10].png   # output of question 4
         |__base_policy_2.mov   # output of question 5
         |__base_policy_2/frame[0-10].png   # output of question 5
         |__standard_rollout_policy_1.mov   # output of question 6
         |__standard_rollout_policy_1/frame[0-10].png   # output of question 6
         |__standard_rollout_policy_2.mov   # output of question 7
         |__standard_rollout_policy_2/frame[0-10].png   # output of question 7
         |__one_at_a_time_rollout_2.mov   # output of question 8
         |__one_at_a_time_rollout_2/frame[0-10].png   # output of question 8
```

Questions

You need to work on the questions 1 and 2 first and then question 3. After that, you may choose to work on the rest of the questions in any order you wish.

Please submit a zip file. The zip file should contain a report which should include the following files mentioned below. The zip file may also include the movies (generated for questions 3-8)[3]. In addition to the figures and movies, please add a few lines in the report, detailing your observations and intuitions behind those observations regarding the different behaviors of the policies (namely, base policy, standard rollout, and one-agent-at-a-time rollout).

1 Find and visualize pickup probability distribution:

You need to complete the function **find_pickup_probability()** inside **src/get_disturbance.py**[4]. Populate *pickup_probability*, a dictionary, where each key corresponds to a grid location's x and y coordinates, and the value gives the probability that a request generated will have the pickup location at the grid location. Once you finished writing the code, run function **main_question_1** inside **src/main.py**

Deliverable [2 pts] pickup_distribution.png: A figure visualizing the pickup probability distribution using the dataset. The distribution is to be given by a dictionary, with key-value pair $(x, y) : p_u$. Here x and y are the x, and y coordinates in the grid, and p_u is the probability that a new request has the pickup location of (x, y). Finish the incomplete parts of find_pickup_probability() in get_disturbance.py (and then run main_question_1()).

2 Find and visualize dropoff probability distribution:

You need to complete the function **find_dropoff_probability()** inside **src/get_disturbance.py**. Populate $dropoff\_probability$, which is a dictionary, where each key corresponds to a grid location's x and y coordinates, and the value gives the probability that a request generated will have the dropoff location at the grid location. Once you finished writing the code, run function **main_question_2** inside **src/main.py**

Deliverable [2 pts] dropoff_distribution.png: A figure visualizing the dropoff probability distribution using the dataset. The distribution is to be given by a dictionary, with key-value pair $(x, y) : d_o$. Here x and y are the x, and y coordinates in the grid, and d_o is the probability that a new request has the dropoff location of (x, y). Finish the incomplete parts of find_dropoff_probability() in get_disturbance.py (and then run main_question_2()).

3 Generate and save requests:

Run function **main_question_3** inside **src/main.py** and the outputs should be saved in **data/requests.csv, data/requests.mov**. The movie frames will be saved in **data/requests/** folder.

[3] We have provided you with an example movie in skeleton_code/data/trajectory_file/base_policy_2.mov

[4] Note that in the taxicab pickup problem the distributions *pickup_probability* and $dropoff\_probability$ constitute disturbances w_k for all $k = \{0, \ldots, N-1\}$.

Deliverable [1 pts] Frames of requests.mov: You need to first generate and save (in requests.csv) a few requests over a horizon of $N = 10$ using the distributions generated in the last two questions (pickup_distribution.csv, dropoff_distribution.csv). Visualize the requests in requests.mov (by running main_part_3()) and add the frames in the report.

4 Base policy:

The base policy directs an agent to pick up a request at its current location. If there is no request in the current location of the agent, the base policy directs the agent towards the nearest request's pickup location which is essentially greedy. Complete the following functions inside **src/base_policy.py** for finding the base policy's control component for an agent.

(a) You need to first complete the function **next_direction(location1, location2)**. This function should return the direction denoted by 0, 1, 2, 3, and 4 (which correspond to Stay, Left, Right, Up, and Down, respectively). The direction is obtained while traversing the shortest distance (Manhattan) starting from location1 to the final destination location2. You can assume that the horizontal motion is preferred over the vertical motion while breaking ties. For example, in order to move from location1 (at $(2, 1)$) to location2 (at $(4, 3)$) in shortest time according to the Manhattan distance, both controls 2 (Right) and 4 (Down) can be equally viable. But, you can choose the control 2 (Right) in this case.

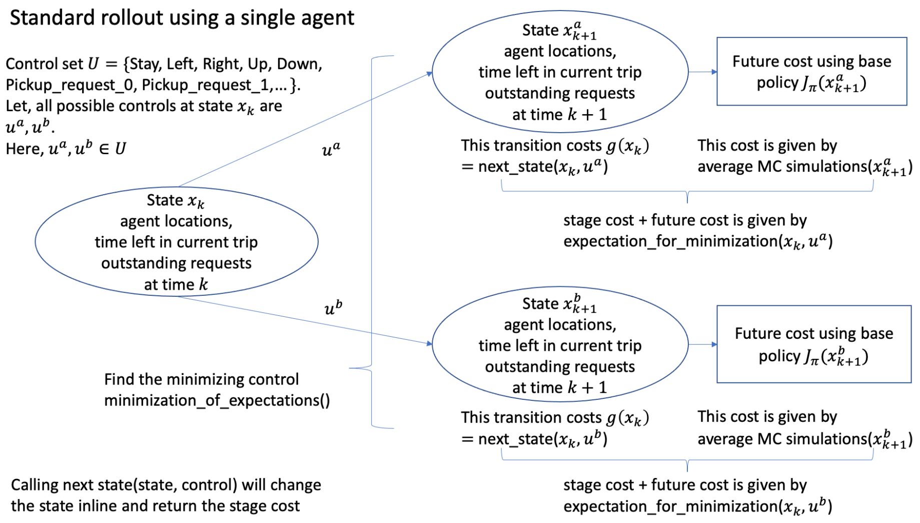
(b) You then, need to complete the function **get_base_policy_control_component (. . ., taxi_state_object, ℓ)**.
For an agent ℓ, find s', where s' is the index to the outstanding_requests of the state object whose pickup loaction is the nearest to the agent's location ($agent\_location[\ell]$).
If the s'th outstanding requests's pickup location is the same as the agent's location, then set the control component to pickup request s' (i.e., $5 + s'$ in the code).
If the s'th outstanding requests's pickup location is not the same as the agent's location, then set the control component to the direction returned by the next_direction(location1, location2). Here use location1 as the agent's current location, and location2 as the s'th outstanding request's pickup location.
If no s' found, then set the control component $= 0$ (Stay).

Deliverable [4 pts] Frames of base_policy_1.mov: These figures will illustrate a trajectory generated by the base policy with a single agent. You should use the generated request samples from requests.csv in Question 3. You need to finish incomplete code in base_policy.py. The output of main_question_4() is a movie, saved as base_policy_1.mov. Add the frames of the movie in the report.

Deliverable [1 pts] Frames base_policy_2.mov: These figures will illustrate a trajectory generated by the base policy using two agents. You should use the generated request samples from requests.csv in Question 3. The output of main_question_5() is a movie, saved as base_policy_2.mov. Add the frames of the movie in the report.

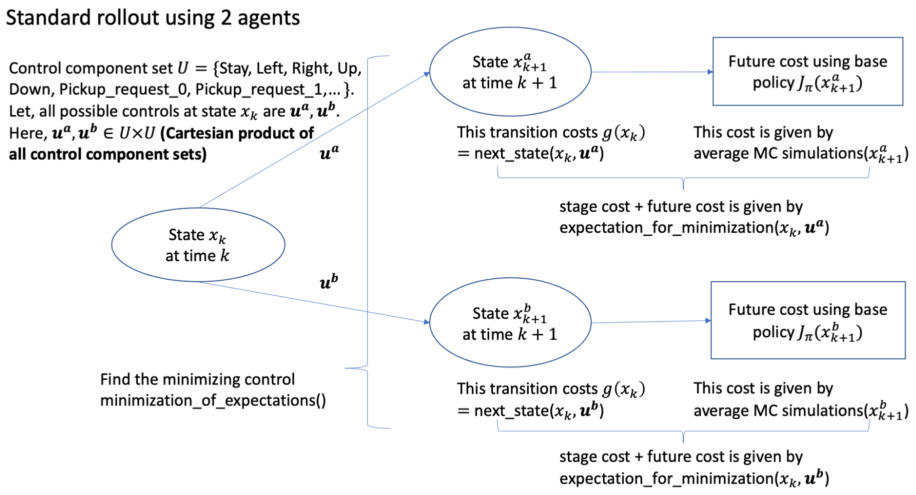
5 Standard rollout:

Complete the following functions inside **standard_rollout** class in **src/algorithms.py**. Figure 1 and Figure 2 show the schematics of the standard rollout performed with 1 agent and 2 agents, respectively.



Standard rollout using a single agent
Control set U = {Stay, Left, Right, Up, Down, Pickup_request_0, Pickup_request_1,… }.
Let, all possible controls at state x_k are u^a, u^b.
Here, $u^a, u^b \in U$
State x_k agent locations, time left in current trip outstanding requests at time k
u^a
u^b
State x^a_{k+1} agent locations, time left in current trip outstanding requests at time $k+1$
Future cost using base policy $J_\pi(x^a_{k+1})$
This transition costs $g(x_k)$ = next_state(x_k, u^a)
This cost is given by average MC simulations(x^a_{k+1})
stage cost + future cost is given by expectation_for_minimization(x_k, u^a)
State x^b_{k+1} agent locations, time left in current trip outstanding requests at time $k+1$
Future cost using base policy $J_\pi(x^b_{k+1})$
This transition costs $g(x_k)$ = next_state(x_k, u^b)
This cost is given by average MC simulations(x^b_{k+1})
stage cost + future cost is given by expectation_for_minimization(x_k, u^b)
Find the minimizing control minimization_of_expectations()
Calling next state(state, control) will change the state inline and return the stage cost

Figure 1: Standard rollout with 1 agent. The number of controls used in the minimization is $O(|U|)$.



Standard rollout using 2 agents
Control component set U = {Stay, Left, Right, Up, Down, Pickup_request_0, Pickup_request_1,… }.
Let, all possible controls at state x_k are $\boldsymbol{u^a}, \boldsymbol{u^b}$.
Here, $\boldsymbol{u^a}, \boldsymbol{u^b} \in U \times U$ (Cartesian product of all control component sets)
State x_k at time k
$\boldsymbol{u^a}$
$\boldsymbol{u^b}$
State x^a_{k+1} at time $k+1$
Future cost using base policy $J_\pi(x^a_{k+1})$
This transition costs $g(x_k)$ = next_state($x_k, \boldsymbol{u^a}$)
This cost is given by average MC simulations(x^a_{k+1})
stage cost + future cost is given by expectation_for_minimization($x_k, \boldsymbol{u^a}$)
State x^b_{k+1} at time $k+1$
Future cost using base policy $J_\pi(x^b_{k+1})$
This transition costs $g(x_k)$ = next_state($x_k, \boldsymbol{u^b}$)
This cost is given by average MC simulations(x^b_{k+1})
stage cost + future cost is given by expectation_for_minimization($x_k, \boldsymbol{u^b}$)
Find the minimizing control minimization_of_expectations()

Figure 2: Standard rollout with 2 agents. Note that the number of controls used in the minimization is $O(|U|^2)$.

(a) You need to complete function **get_cartesian_product(..,list_of_sets)**, which gives a Cartesian product of all sets in the **list_of_sets**.

(b) You need to complete the function **expectation_for_minimization(…, taxi_state_object, control)** by doing the following. a) Call **next_state(taxi_state_object,control)** to get next state and the stage cost. b) Call **average_MC_simulations** using the next state and 'base_policy' (as parameter) to get future cost.

(c) Complete function **minimization_of_expectations** to find the control that minimizes the expected sum of current and future costs.

(d) Complete function **get_control(...)**, by doing the following,
a) Find the control component set for each agent using **available_control_agent** of the state object.
b) Find the Cartesian product of all control component sets using **get_cartesian_product(..,list_of_sets)**.
c) Construct a dictionary for each control as the key in the Cartesian product and its corresponding cost as the value. The cost of a control is obtained using **expectation_for_minimization**.
d) Use **minimization_of_expectations** to find the minimizing control in the dictionary.

Deliverable [5 pts] Frames of standard_rollout_1.mov: These figures will illustrate a trajectory generated by the standard rollout policy using a single agent. You should use the generated request samples from requests.csv in Question 3. You need to finish incomplete code in algorithms.py (class: standard_rollout). The output of main_question_6() is a movie, saved as standard_rollout_1.mov. Add the frames of the movie in the report.

Deliverable [5 pts] Frames of standard_rollout_2.mov: These figures will illustrate a trajectory generated by the standard rollout policy using two agents. You should use the generated request samples from requests.csv in Question 3. You need to finish incomplete code in algorithms.py (class: standard_rollout). The output of main_question_7() is a movie, saved as standard_rollout_2.mov. Add the frames of the movie in the report.

6 One-agent-at-a-time rollout

Complete the following functions inside **one_at_a_time_rollout** class in **src/algorithms.py**. Figure 3 shows the schematic of the one-agent-at-a-time rollout performed by two agents. You can reuse the **expectation_for_minimization**, and **minimization_of_expectations** from standard rollout functions. Complete the **get_control** function by doing the following.

(a) Call **base_policy().get_control()** using the state object to construct *base_policy_control*, which is the control given by the base policy.

(b) Initialize an empty list *one_at_a_time_rollout_control*.

(c) For each agent ℓ (starting from $\{0, \dots, m\}$ in that order), do the following.

- Construct a list of control components of size m. Let's call it **control_construction**. Set

$$constructed\_control[0:\ell-1] = one\_at\_a\_time\_rollout\_control[0:\ell-1]$$

  and

$$constructed\_control[\ell+1:m-1] = base\_policy\_control[\ell+1, m-1].$$

- Find the control component set for agent ℓ using **available_control_agent** of the state object.
- Construct a dictionary for each control component of agent ℓ as key and its corresponding cost as the value. The cost of a control component u^ℓ is obtained using **expectation_for_minimization** and passing the constructed_control as parameter (set constructed_control[ℓ]=u^ℓ before passing the parameter).

- Use **minimization_of_expectations** to find the minimizing control component ($\tilde{u}^\ell$) in the dictionary for agent ℓ.
- Append $\tilde{u}^\ell$ to *one_at_a_time_rollout_control*.

Deliverable [5 pts] Frames of one_agent_at_a_time_rollout_2.mov: These figures will illustrate a trajectory generated by the one-agent-at-a-time rollout policy using two agents. You should use the generated request samples from requests.csv in Question 3. You need to finish incomplete code in algorithms.py (class: one_at_a_time_rollout). The output of main_question_8() is a movie, saved as one_agent_at_a_time_rollout_2.mov. Add the frames of the movie in the report.

References

[1] Dimitri Bertsekas. Multiagent rollout algorithms and reinforcement learning. *arXiv preprint arXiv:1910.00120*, 2019.

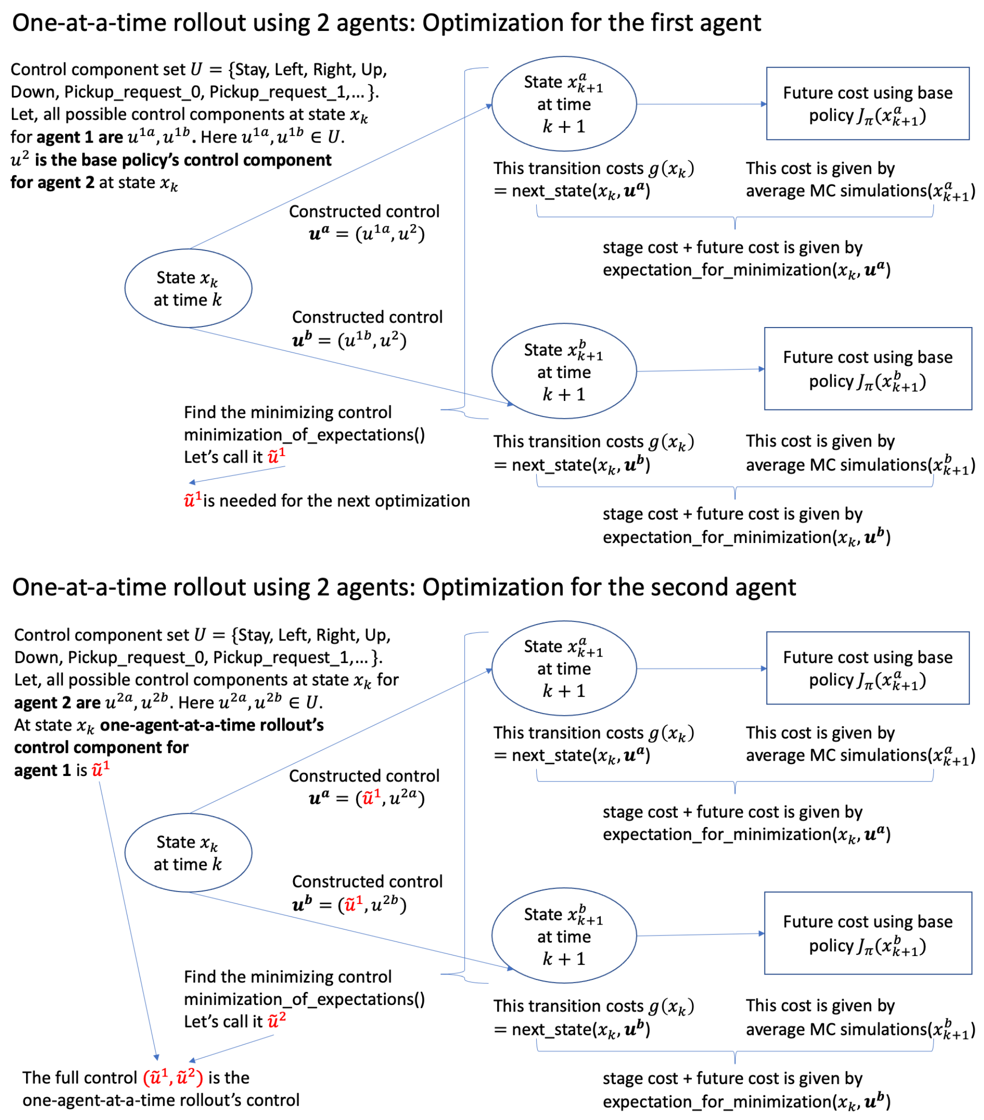

One-at-a-time rollout using 2 agents: Optimization for the first agent
Control component set U = {Stay, Left, Right, Up, Down, Pickup_request_0, Pickup_request_1,… }.
Let, all possible control components at state x_k for **agent 1 are** u^{1a}, u^{1b}. Here $u^{1a}, u^{1b} \in U$.
u^2 **is the base policy's control component for agent 2** at state x_k
State x_k at time k
Constructed control $\boldsymbol{u^a} = (u^{1a}, u^2)$
Constructed control $\boldsymbol{u^b} = (u^{1b}, u^2)$
State x^a_{k+1} at time $k+1$
State x^b_{k+1} at time $k+1$
Future cost using base policy $J_\pi(x^a_{k+1})$
Future cost using base policy $J_\pi(x^b_{k+1})$
This transition costs $g(x_k)$ = next_state($x_k, \boldsymbol{u^a}$)
This cost is given by average MC simulations(x^a_{k+1})
stage cost + future cost is given by expectation_for_minimization($x_k, \boldsymbol{u^a}$)
This transition costs $g(x_k)$ = next_state($x_k, \boldsymbol{u^b}$)
This cost is given by average MC simulations(x^b_{k+1})
stage cost + future cost is given by expectation_for_minimization($x_k, \boldsymbol{u^b}$)
Find the minimizing control minimization_of_expectations() Let's call it $\tilde{u}^1$
$\tilde{u}^1$is needed for the next optimization
One-at-a-time rollout using 2 agents: Optimization for the second agent
Control component set U = {Stay, Left, Right, Up, Down, Pickup_request_0, Pickup_request_1,… }.
Let, all possible control components at state x_k for **agent 2 are** u^{2a}, u^{2b}. Here $u^{2a}, u^{2b} \in U$.
At state x_k **one-agent-at-a-time rollout's control component for agent 1** is $\tilde{u}^1$
State x_k at time k
Constructed control $\boldsymbol{u^a} = (\tilde{u}^1, u^{2a})$
Constructed control $\boldsymbol{u^b} = (\tilde{u}^1, u^{2b})$
State x^a_{k+1} at time $k+1$
State x^b_{k+1} at time $k+1$
Future cost using base policy $J_\pi(x^a_{k+1})$
Future cost using base policy $J_\pi(x^b_{k+1})$
This transition costs $g(x_k)$ = next_state($x_k, \boldsymbol{u^a}$)
This cost is given by average MC simulations(x^a_{k+1})
stage cost + future cost is given by expectation_for_minimization($x_k, \boldsymbol{u^a}$)
This transition costs $g(x_k)$ = next_state($x_k, \boldsymbol{u^b}$)
This cost is given by average MC simulations(x^b_{k+1})
stage cost + future cost is given by expectation_for_minimization($x_k, \boldsymbol{u^b}$)
Find the minimizing control minimization_of_expectations() Let's call it $\tilde{u}^2$
The full control $(\tilde{u}^1, \tilde{u}^2)$ is the one-agent-at-a-time rollout's control

Figure 3: One-agent-at-a-time rollout: two sequential optimizations performed by two agents. Note that the total number of controls used in the minimizations is $O(2|U|)$.