Reinforcement Learning
Architecture 10/09
ORCS 4529: Fall 2025

Team Waymo
waymo-rl@columbia.edu
Due: 10/09/2025 11:59 PM ET

State

Discrete-Time Markov Decision Process (MDP)

Time steps are 1 minute long

Environment

- Static class - no state changes (mostly)
- weather conditions (sunny, rainy, snowy, etc)
- time of day (could be encoded as a float from 0 to 24)
- day of week (could be encoded as an int from 0 to 6)
- Embedded into l_2
- Graph:
  - Nodes - intersections
  - Edges - roads
- Nodes:
  - Only places rides can start/end
  - l_2 coordinates
  - bool is_charging_station
  - int num_available_chargers
- Edges:
  - Length
  - 2-way speed limit, possibly asymmetric
  - *potentially:* 2-way current speed, possibly asymmetric (this would be dynamic)

Fleet

- n=10? vehicles
- Per vehicle:
  - Vehicle ID
  - Location
  - Battery level (percentage)
  - Status (available, en route to pickup, with passenger, en route to charging station)

Demand

I'm a little unsure about whether this should be *explicitly* modeled in the state or "learned" by the agent. I'm learning towards the former for a few reasons

(a) It's relatively independent of the actions of the agent

(b) It can be estimated from historical data

(c) Both assumptions above are likely to be true in the real world

(d) It greatly simplifies the learning problem

If we do explicitly model it

- Location
- Expected demand (number of requests) in next n time steps

Active Rides

I think that this class should mostly serve as an enriched tuple between a vehicle and a ride request (described below)

- Matched Vehicle ID
- Request ID
- Status (en route to pickup, with passenger)

- Time to pickup (minutes)
- Duration of ride (minutes)
- Estimated remaining time (minutes)

Ride Requests

There's one part of this process that I believe is non-trivial. In the real world, the platform (e.g. Uber, Lyft) suggests a menu of prices to the rider, who then selects one. Subsequently, all (nearby) drivers are notified of the ride request, and one can accept it.

In our simulation, if the agent recommends prices, intuitively it would be a price that the agent would accept [extension]. So that transition probability is 1. However, you could make a case that riders should "bid" on rides, and the agent can (1) accept (2) reject (3) counter-offer. For the sake of simplicity, I would rather model the process as (1) customer requests a ride (2) the agent suggests a price (3) the customer either accepts or rejects the price based on some probabilistic model. This would allow the agent to learn to price rides optimally. It would also provide a bound on the maximum price the agent could set (since the agent would learn to just rise prices without bound if the customer always accepted).

In this proposed process, the price recommendation would be part of the action space, and the customer's decision would be encoded in the transition probability.

- $n_v \sim \text{Poisson}(\Lambda)$ new requests per time step (this lives in Demand class)
- Each request:
  - Account ID (drawn from a pool of u users)
  - Request ID
  - Pickup location
  - Dropoff location (!= pickup location). This could be modeled as a uniform distribution at first, but ideally would be learned from historical data.
  - Request time
  - Status (categorical)
    - Price suggestion pending (brand new request generated between t and $t-1$). Agent must suggest a price before next time step
    - Price suggested, pending customer acceptance/rejection (customer immediately accepts/rejects based on probabilistic model)
    - Accepted, pending assignment to vehicle (this state only exists if there are no available vehicles)

    * Assigned to vehicle, pending pickup (vehicle is en route to pickup location)
    * Cancelled by customer (optional - if pickup time is too long)
    * Active (joined with Vehicle class as Active Ride)
    * Completed

Note that we don't need to "explicitly" model aggregate supply demand imbalance, hopefully the agent will learn the notion of surge pricing on its own. However, a simple float variable representing the current ratio of supply to demand could be added to the state to accelerate learning.

Actions

Per Vehicle

- Move (velocity vector) (always happens unless vehicle is charging)
- bool flip_charging_state (go to / from charging station)

Global

- $\mathbf{p} = [p_1, p_2, \ldots, p_{n_v}]$ price vector for all pending requests
- Assign accepted requests to vehicles (if any are available)

Transitions

Environment

Proposition 1: Static, no transitions (unless we want to model traffic & speed)
Proposition 2: Transition model learnt implicitly using model-free algorithm.

Fleet

Set of cars is represented as

$$\{\mathbf{c}_i\}_{i=1}^{n} \subset \mathbb{R}^{[n]\times[0,1]\times[4]}$$

where each $\mathbf{c}_i = (\text{position}, \text{charge}, \text{status})$.

- Location - updated based on velocity vector, speed limits, and road network
  - Proposition 1: Model location as a Cartesian grid (easier zoning, but more states).
  - Proposition 2: Model location as a node in graph (natural weighted edge representation for traffic).
- Battery level - decreases based on distance traveled, increases when at charging station
- Status - updated based on actions and ride assignments: occupied, repositioning, idle, charging

Demand

- Updated based on historical data (e.g. time of day, day of week, weather conditions)
  - Proposition: feed historical distribution of demand as prior to algorithm input. Note: to keep policy stationary, we can initialize rewards according to area popularity.

Active Rides

- Status - updated based on vehicle status and ride progress
- Time to pickup - decreases as vehicle approaches pickup location
- Duration of ride - increases as ride progresses
  - Note: traffic changes as ride progresses, does it mean that car should change its action during the ride? That exponentially increases amount of decisions to be made (hence longer exploration time) although it's unlikely that car needs to change the route.
- Estimated remaining time - updated based on current time to pickup and duration of ride

Ride Requests

- New requests - generated based on Poisson distribution with rate parameter Λ
- Status - updated based on customer acceptance/rejection and ride assignments
  - Expired requests - removed if not accepted within a certain time frame
  - Cancelled requests - removed if customer cancels before assignment
  - Completed requests - removed once ride is finished
  - Price acceptance - probabilistic model based on suggested price and customer sensitivity
- $p_{accept} = \frac{1}{1+e^{-z}}$ (logistic function)
- z-features:
  - customer bias (base acceptance rate per customer)
  - pickup & destination location bias (proxy for neighborhood wealth, proximity to public transit)

    * how connected node is $\sim$ subway proximity
    * avg income in neighborhood $\sim$ census data (or just random manifold)
- price offered by agent
- distance
- global supply-demand imbalance
- date & time
- weather conditions

Rewards & Costs

Rewards

- Revenue from completed rides
- *stretch:* customer loyalty / retention

Costs

Explicit

- Operational costs (electricity) - per mile cost
  - Cost not paid when vehicle is charging
  - Cost proportionate to distance (hence speed)
  - Repositioning costs
- Customer dissatisfaction
  - Rejected rides (long term effect on customer loyalty, hard to learn implicitly due to credit assignment problem)
  - Long wait times
  - Incomplete area coverage

Implicit / Learned

- Rejected ride requests (lost revenue)
- Unfulfilled demand (due to supply shortage; lost revenue)
- Long wait times (due to supply shortage — uneven vehicle distribution — traffic — underpricing (hence excess load))
- Non-shortest path routing (traffic, geography)

Implementation Miscellany & Thoughts

- Leverage OpenAI Gym framework
- Leverage existing graph libraries (e.g. NetworkX) for road network?
- Instant distribution of amortized costs & rewards at each time step (speed up learning)
- Start incredibly simple (shrink dimensionality)
- Initialize with heuristic policy (e.g. nearest vehicle, price = cost * (1 + margin))
- Log metrics (rewards, costs, wait times, rejection rates, etc) for analysis, visualization, debugging & final presentation
- Strict and thoughtful versioning of both code and experiments (match up version of code with results, reduce ambiguity in analysis)
- Burning question: is algorithm a centralized dispatcher or multi-agent? I think multi-agent is too complicated, slides for Multi-Agent RL only cover 2-player Nash equilibrium

Policy Search Approaches

- Q-learning
- Monte-Carlo
- (benchmark) Heuristics: nearest idle, least recently busy
- (benchmark) Robo-taxi's Network Flow solution Section 4.2
- Policy Gradient
- Actor-Critic
- Deep Q-Network [extension]
- Multi-Agent

Timeline

- Set up environment for demand and traffic model
- Determine policy for car dispatch
- Add pricing suggestions
- Add charging