REINFORCEMENT LEARNING

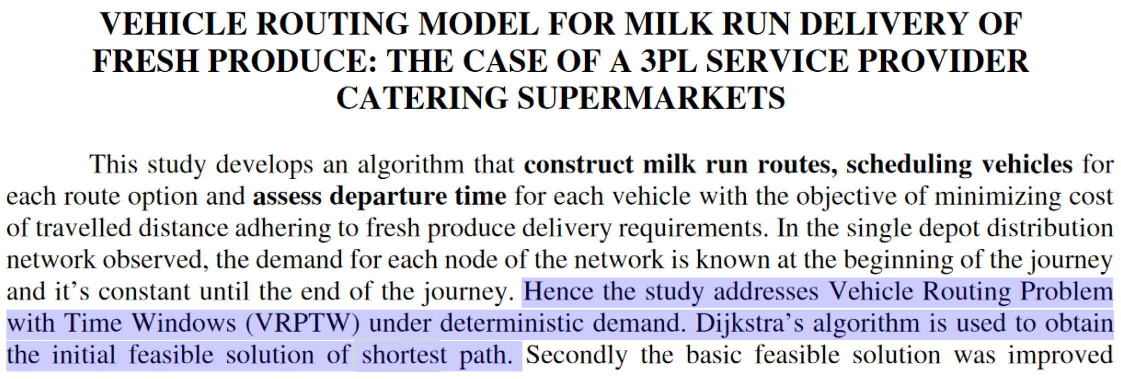
FUNDAMENTALS AND APPLICATIONS

Midterm Solutions

Please note that these are only *possible* midterm solutions. Most of the exam was open-ended, and these are merely the responses I came up with as I was completing the midterm myself.

For example, for Question 11 Jonathan Tan suggested a shortest path algorithm for vehicle routing, and Yanhao Huang noted that ant colony optimization is probably the best approach for logistics management. Those parts of their answers were both (a) not at all what I came up with myself / write here, (b) examples of good/clever thinking (not everything needs to be ML), and (c) just plain *right*:

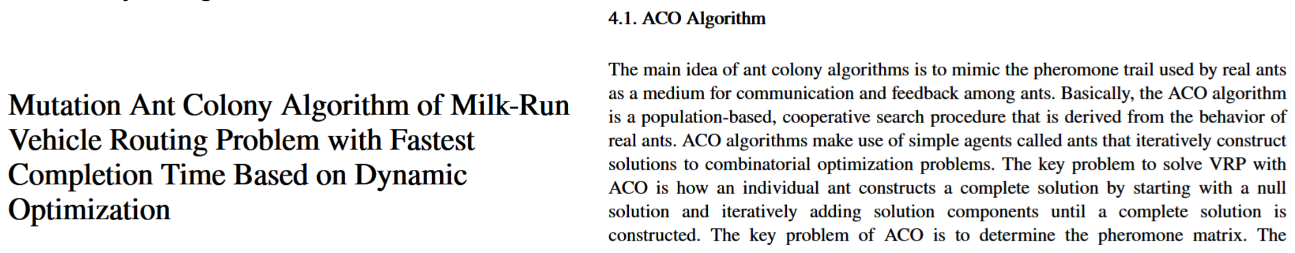
Shortest path in logistics research:



VEHICLE ROUTING MODEL FOR MILK RUN DELIVERY OF FRESH PRODUCE: THE CASE OF A 3PL SERVICE PROVIDER CATERING SUPERMARKETS

This study develops an algorithm that **construct milk run routes, scheduling vehicles** for each route option and **assess departure time** for each vehicle with the objective of minimizing cost of travelled distance adhering to fresh produce delivery requirements. In the single depot distribution network observed, the demand for each node of the network is known at the beginning of the journey and it's constant until the end of the journey. Hence the study addresses Vehicle Routing Problem with Time Windows (VRPTW) under deterministic demand. Dijkstra's algorithm is used to obtain the initial feasible solution of shortest path. Secondly the basic feasible solution was improved

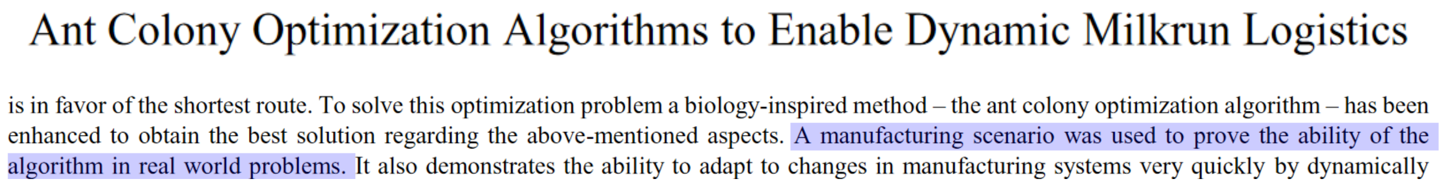
Ant Colony in logistics research:



Mutation Ant Colony Algorithm of Milk-Run Vehicle Routing Problem with Fastest Completion Time Based on Dynamic Optimization

4.1. ACO Algorithm

The main idea of ant colony algorithms is to mimic the pheromone trail used by real ants as a medium for communication and feedback among ants. Basically, the ACO algorithm is a population-based, cooperative search procedure that is derived from the behavior of real ants. ACO algorithms make use of simple agents called ants that iteratively construct solutions to combinatorial optimization problems. The key problem to solve VRP with ACO is how an individual ant constructs a complete solution by starting with a null solution and iteratively adding solution components until a complete solution is constructed. The key problem of ACO is to determine the pheromone matrix. The



Ant Colony Optimization Algorithms to Enable Dynamic Milkrun Logistics

is in favor of the shortest route. To solve this optimization problem a biology-inspired method – the ant colony optimization algorithm – has been enhanced to obtain the best solution regarding the above-mentioned aspects. A manufacturing scenario was used to prove the ability of the algorithm in real world problems. It also demonstrates the ability to adapt to changes in manufacturing systems very quickly by dynamically

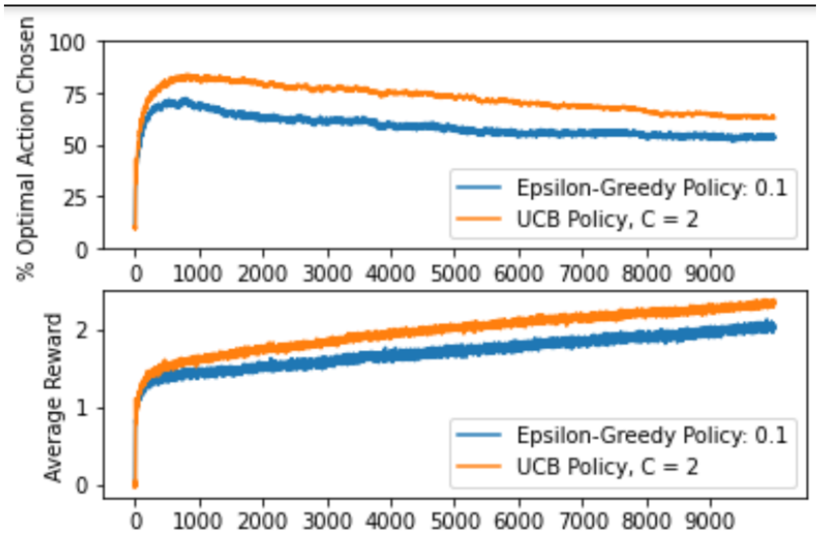
Question 1: Startup Bandit

1. **You are working for a startup in Waterloo. They have developed ten different user interface designs for the website and wish to know which is "the best". How would you use a k-armed bandit to provide the answer they seek? Provide a brief description of how you would frame reward (no need to go into extended detail).**

I would assign each UI to an arm, and set up a bandit algorithm to select the UI shown to a given new user. The current UI would be maintained as an 11th action as the control. I would use cookies to at least somewhat ensure consistency across users (i.e., so a user who reloaded the page did not receive a different UI each time, and to know if a given user had received a different UI before to maintain at least a modicum of independence). I would use a uniform random policy so my estimates of reward were unbiased (i.e. randomized controlled trial), and would use active time on page as the reward signal. "Active time on page" would mean the time since load to 5 minutes since the last action.

Question 2: Bandit Results

2. **Take a look at "Bandits Results.jpg" in the Resources tab. Interpret the results. (1 mark)**



The random walk has moved more than one action's q* upward (safe guess: 5 of them), and even with random exploration the bandit is netting more reward from the system over time, as both the greedy option and (probably) half the exploratory options are providing more reward than they did initially. The optimal action select % has degraded a bit over time because of the lag between a new optimal action and the agent's discovery of it, but it appears to have levelled off by the 9000th timestep or so, possibly because the best action has "broken away from the pack" and swaps would need many timesteps to occur except in extremely unlikely sequences of random draws.

Question 3: What is a Markov Decision Process?

3. **In your own words, tell me what a Markov Decision Process is. In the real world, is it possible to have a perfect Markov Decision Process? Why or why not? (2 marks)**

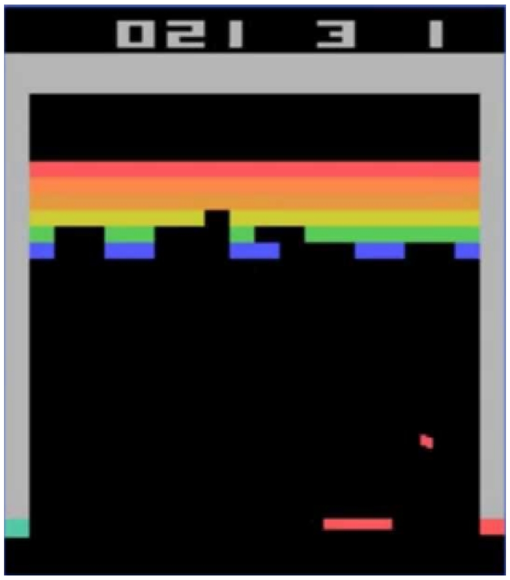
A Markov Decision Process is essentially a tuple (A, S, P, R, y) that defines the dynamics of a system:
- A finite set of actions A,
- A finite set of states S,
- A state-transition probability matrix P,
- A reward function R, and
- A discount factor gamma.

Critically, the transitions of the system depend on the Markov Property, which requires that a given state S contains all the information required to define transitions to successor states. In brief an MDP is an idealized form of the reinforcement learning problem, such that we make assumptions about the Markov Property and the state representation: we assume that our state representation really *does* contain all required information, even though this is rarely the case.

I say “rarely the case” because there are “perfect” MDPs in the world in the form of games. For example, the game of Go is such a “perfect” MDP: the state of the board provides all the information we need, the reward function is clear, and the actions are clearly defined. However, in practice “true” MDPs are rarely found and we must assume that our Markov assumption is “safe enough”, i.e., even without a perfect MDP, formulating the problem space as an MDP allows us to solve it.

Question 4: Breakout MDP

4. **A colleague of yours is learning reinforcement learning, and is excited to implement a Deep-Q Network that can play Atari. He tells you how amazing it is that all you have to do is feed the image to the network, and it will learn! He says he is planning to first teach it to play Breakout, but hasn't implemented it just yet. View the image "Breakout.jpg" in the Resources tab. Does using an encoding of the game's visuals satisfy the Markov Property? Why or why not?**



It does not. The image contains almost all the information, but we cannot tell if the "bullet" or "ball" is going up, down, left, right, or what. We need information about direction and velocity to make fully informed decisions. (This is why the state for Atari games is actually a combination of prior frames, rather than just the last frame).

Question 5: Expected SARSA and Frozen Lake

5. **You are implementing Expected SARSA to solve the Frozen Lake task on OpenGym. The first 5 setups you have tried show zero learning (zero rewards), even after 2000 episodes. You have checked your algorithm implementation and it is good. What would you consider the most important aspect of the problem to investigate? Why?**

    1. I would examine later episodes to see exactly what the agent is doing (i.e., watch some renderings of its behaviour). Perhaps it constantly gets stuck falling into a particular hole.
    2. I would test various exploration modes: different levels of epsilon, decay schedules, "don't lower epsilon until you solve it at least once", and so on.
    3. Maybe Expected SARSA is not "good enough" to solve Frozen Lake reliably, and we need other techniques.

Question 6: Tabular Method Definitions, Pros, and Cons

6. Explain the differences between TD(0), SARSA, Q-Learning, and Expected SARSA. What are the pros and cons?

TD(0): As presented in the text, TD(0) is essentially a policy evaluation algorithm; it is for prediction, rather than control. It updates each V(s) toward R+yV(s') in order to "chain predictions forward", which results in V(s) converging to the true long-term return. Put another way, as more and more episodes are run, each V(s) becomes closer to V(s') and information propagates backward through every trajectory chain. While TD(0) is better than Monte Carlo in terms of variance and efficiency, it is significantly more biased as the target contains a guess about the long-term return of the successor state.

SARSA: SARSA is on-policy TD(0) that updates according to the action that was actually selected. It functions similarly, and critically can learn while the episode is still going on. SARSA, being on-policy, requires an exploratory component (e.g., e-soft) to explore enough to ensure convergence. SARSA is a solid "workhorse" algorithm that is relatively conservative compared to Q-Learning, however it only learns a *near*-optimal policy while exploring; to get directly at the optimal policy we would need to decay exploration over time and choose the decay parameter "correctly".

Q-Learning: In contrast to SARSA, Q-Learning is an off-policy algorithm that uses only the maximal available action in its updates *regardless of the action that was actually taken.* This gives Q-Learning the advantage of "not getting discouraged", or "walking the razor's edge". Because Q-Learning focuses on the optimal action *only*, it will be willing to take risks and learn potentially more dangerous paths to success. Similar to SARSA, it requires an exploratory component to ensure convergence, yet due to the off-policy nature of Q-Learning it *directly* learns the optimal policy.

Expected SARSA updates each Q-estimates not according to the max, but to the policy-weighted average of the available actions. In this way it trains more quickly than SARSA (using all available actions in proportion to their likelihood of being selected), and is significantly less risky than Q-Learning (if one or more actions are heavily negative, that information is always included in the update in proportion to the likelihood of the negative events coming to pass). Similar to SARSA, we would need to decay exploration over time to learn the optimal policy rather than a near-optimal policy.

Question 7: Real-World Applications of Tabular Methods

7. Describe one real-world task you think each algorithm might be a good fit for.

TD(0) evaluating a policy for an MDP in which Monte Carlo is inappropriate or undesirable (e.g., if Monte Carlo would result in excessive wandering due to not learning from intra-episode experience, or if the task is continuing rather than episodic, or if episodes tend to be lengthy because of a large state space and sparse terminal events). An example might be a self-driving car, or a box-sorting robot in a warehouse.

SARSA is good for control in either of the two situations given above.

Q-Learning is good for situations where learning from risky behaviour is not a problem. For example, in games or simulated environments. A real example might be training a robot to run and jump in a simulated environment, or training a drone to fly in a simulated neighbourhood. Crashing in real life is bad, so it is better to get it out “ahead of time”. Just so, Q-Learning may be better to learn optimal action values in a simulated environment, with a later switch to a more conservative algorithm once things move into real life.

Expected SARSA would be good for a task like the insurance price optimization paper. It is likely to recognize and take into account negative situations, and we don’t want it taking risks like Q-Learning might. It’s actually curious that the authors of said paper used Q-Learning… I wonder if Expected SARSA would perform better! They used simulation, however, so the insurance price optimization paper might be better for offline learning, while a switch to expected SARSA might be best when putting the agent into production.

Question 8: Amazon Delivery Service

The year is 2031. You are working as a data scientist for the Amazon Delivery Service. ADS has recently launched a guarantee that every drone-deliverable package ordered from Amazon will be delivered to your doorstep within 60 minutes. Your data science team has been tasked with creating a machine learning system that will correctly identify the best drone to deliver a package given a delivery order (lat / long / time since order received). Leadership cares primarily about task success (either the package was delivered within the 60-minute window, or it wasn't). Each delivery station is equipped with 7 short-range drones and 3 long-range drones. Drones deliver one package at a time. Would you use RL to find a solution to this problem? Why / why not? How would you frame/model the problem?

I think I would frame this as a supervised learning problem, if machine learning is even required. If a drone comes back from a trip, it is the one to take the next package within its range "off the stack", with long-range drones usually preferring long-range packages. Would there be anything else to do? I can't imagine why it would take a package less close to the 60-minute deadline… the only control aspect seems to be the question 'When would a long-range drone take a short-range package?' Even then, it does not seem like ML would be the answer… perhaps it will help if I frame each problem approach:

a. In RL, we would set up a simulation environment using what we know about the land around a given depot. Then, we use basic data-driven assumptions about order frequency and distance to produce the orders. We use what we know about drone capabilities to specify time-to-deliver (acceleration, top speed, etc.), and insert random weather events according to publicly-available data. The control system receives +1 reward for a package delivered within 60 minutes, and zero otherwise. State would be represented as ____ and drones are actions. Selecting drones that are "out" would be allowed to enable queuing, and presumably the system would learn not to queue drones that are too far out.
b. In supervised learning, we would use a small set of features (drone capabilities represented continuously as I'm sure new models would come along) and performance data - possibly also from the simulator - to classify which drone will be able to deliver the package on time. This framing makes it clear that supervised learning is *not* the answer, as multiple drones could meet the criteria at any one time, and there is a sequential aspect to the task (choosing the 'wrong drone' out of the classified set would result in future problems). We might instead make a separate seconds-to-deliver prediction for each drone to figure out which one will get there soonest… but that may result in long-range drones getting misused.
c. Using a non-ML algorithm, we would use some optimization or scheduling algorithm to always choose the drone that would get the package there soonest, with some safeguards for long-range drone misuse (e.g., if a short-range drone can get there in time, always use the short-range drone first). However, this quickly becomes complicated when we consider the longer-term impact… maybe it will actually be good sometimes to let some deadlines pass (perfect is not possible), so that future deliveries in the queue will be better served.

Taken together, the RL framing is likely to be the most successful. This approach might be augmented with a supervised learning algorithm that predicts time-to-deliver, and another that predicts incoming orders, and maybe even another that predicts local weather. These can be fed to the RL controller as an aspect of state to improve the functioning of the model.

Question 9: Markov Property / MDPs vs. Supervised Learning

Provide the definition for the Markov Property. Given that supervised learning predicts a target conditioned on the inputs, how are the Markov Property and the full framing of transitions in reinforcement learning different from supervised learning?

Supervised learning is about P(y|X): the probability of the target given the inputs.
The Markov Property is about P(St+1|St): The probability of transition to state St+t given St.

RL is different from supervised learning in that the signal received is dependent on the prior transitions. This “chain of dependence” differs sharply from supervised learning, where the next training example has nothing at all to do with the one that came before. As well, the signal received is not exactly the same as the typical target y, but is instead information about the action chosen *from* state St. Note that this is another difference between RL and SL: while in the Markov Property we have P(St+1|St), in ‘real’ RL we have P(St+1,R|S,A): our model’s input is actually A, and that determines where we end up. State St+1 is not a target we are predicting! Further, note that our model is actually influencing - very heavily - the next observation. Even when supervised data are not i.i.d. as in forecasting or language learning, a model’s prediction for a given observation and token cannot substantially change what comes next. In RL, decisions have consequences.

Question 10: Calculations

Consider the following three episodes (note: A→B(+3) indicates a transition from A to B, receiving a reward of 3).

A→B(+3)→A(-1)→C(+1)→A(0)→B(+4)→T
A→C(+2)→B(-3)→A(+1)→C(+1)→T
A→B(-4)→B(+3)→B(+4)→T

Initialize each state's value as 0 and calculate the following:

The first-visit Monte Carlo estimate for each state's value.
V(A):
E1: (3 - 1 + 1 + 0 + 4 + 0) = 7
E2: (2 - 3 + 1 + 1 + 0) = 1
E3: (-4 + 3 + 4 + 0) = 3
(7 + 1 + 3) / 3 = **3.66…** OR **11/3**
V(B):
E1: (-1 + 1 + 0 + 4) = 4
E2: (1 + 1 + 0) = 2
E3: (3 + 4 + 0) = 7
= (4 + 2 + 7) / 3 = **4.33…** OR **13/3**
V(C):
E1: (0 + 4 + 0) = 4
E2: (-3 + 1 + 1 + 0) = -1
= (4 - 1) / 3 = **1**

The every-visit Monte Carlo estimate for each state's value.
A→B(+3)→A(-1)→C(+1)→A(0)→B(+4)→T
A→C(+2)→B(-3)→A(+1)→C(+1)→T
A→B(-4)→B(+3)→B(+4)→T

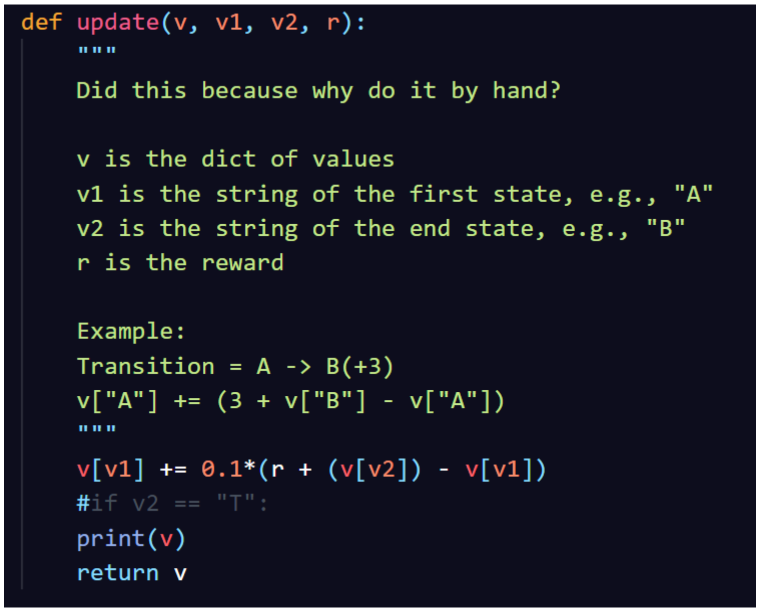
V(A):
(3 - 1 + 1 + 0 + 4 + 0) + (1 + 0 + 4 + 0) + (4 + 0) + (2 - 3 + 1 + 1 + 0) + (1 + 0) + (-4 + 3 + 4 + 0)
= (7 + 5 + 4 + 1 + 1 + 3)
= 21
21/6 = **3.5**

V(B):
(-1 + 1 + 0 + 4 + 0) + 0 + (1 + 1 + 0) + (3 + 4) + 4 + 0
= 4 + 0 + 2 + 7 + 4 + 0
= 17/6 = **2.6**

V(C):
(0 + 4 + 0) + (-3 + 1 + 1 + 0) + 0
= 4 - 1
= 3/3 = **1**

The TD(0) estimate for each state's value.

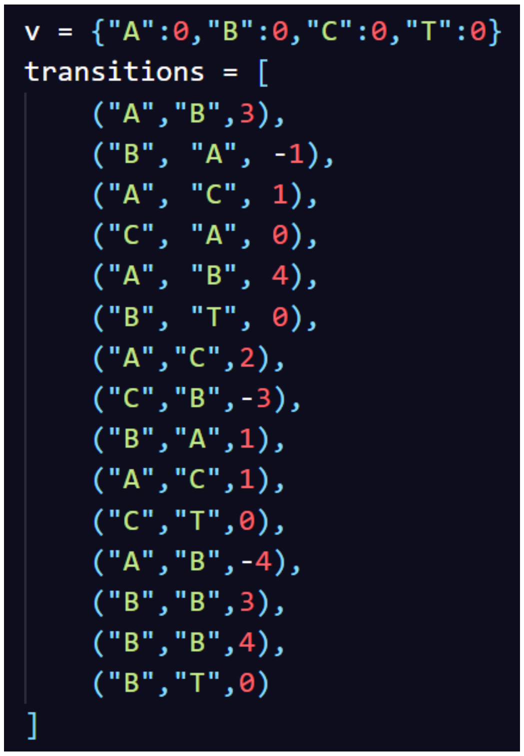
A → B(+3) → A(-1) → C(+1) → A(0) → B(+4) → T
A → C(+2) → B(-3) → A(+1) → C(+1) → T
A → B(-4) → B(+3) → B(+4) → T



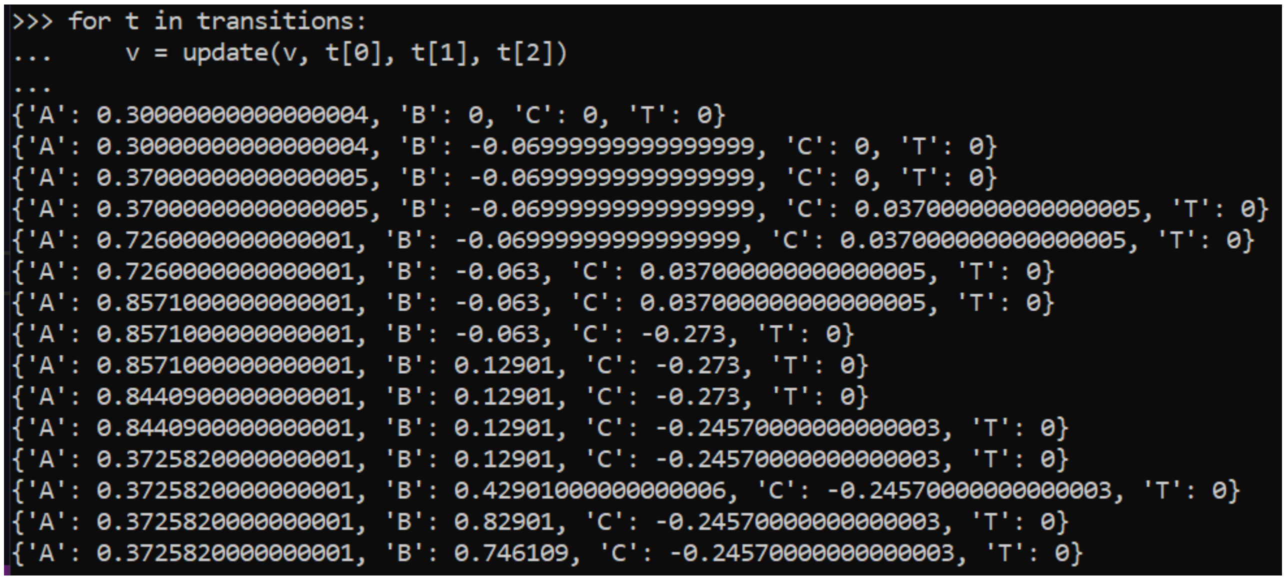
```python
def update(v, v1, v2, r):
    """
    Did this because why do it by hand?

    v is the dict of values
    v1 is the string of the first state, e.g., "A"
    v2 is the string of the end state, e.g., "B"
    r is the reward

    Example:
    Transition = A -> B(+3)
    v["A"] += (3 + v["B"] - v["A"])
    """
    v[v1] += 0.1*(r + (v[v2]) - v[v1])
    #if v2 == "T":
    print(v)
    return v
```

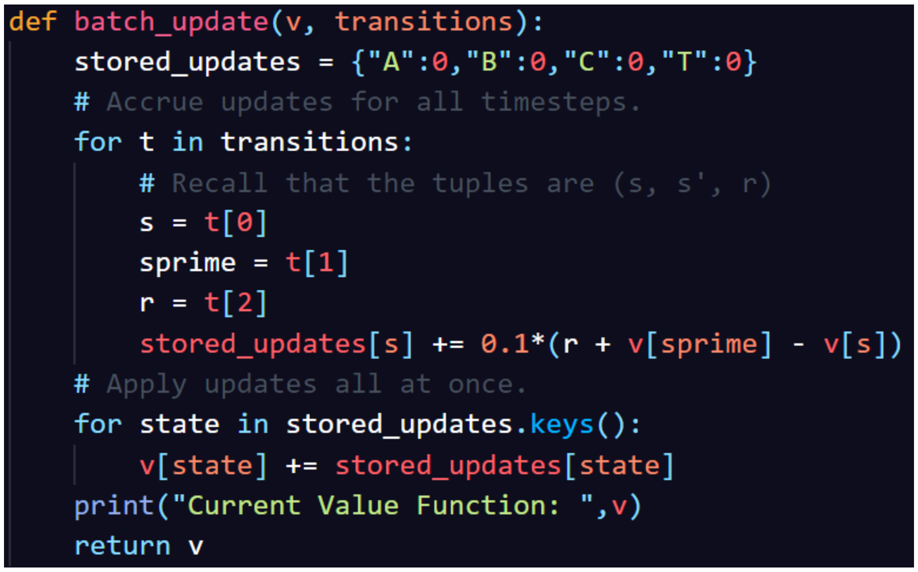


```python
v = {"A":0,"B":0,"C":0,"T":0}
transitions = [
    ("A","B",3),
    ("B", "A", -1),
    ("A", "C", 1),
    ("C", "A", 0),
    ("A", "B", 4),
    ("B", "T", 0),
    ("A","C",2),
    ("C","B",-3),
    ("B","A",1),
    ("A","C",1),
    ("C","T",0),
    ("A","B",-4),
    ("B","B",3),
    ("B","B",4),
    ("B","T",0)
]
```



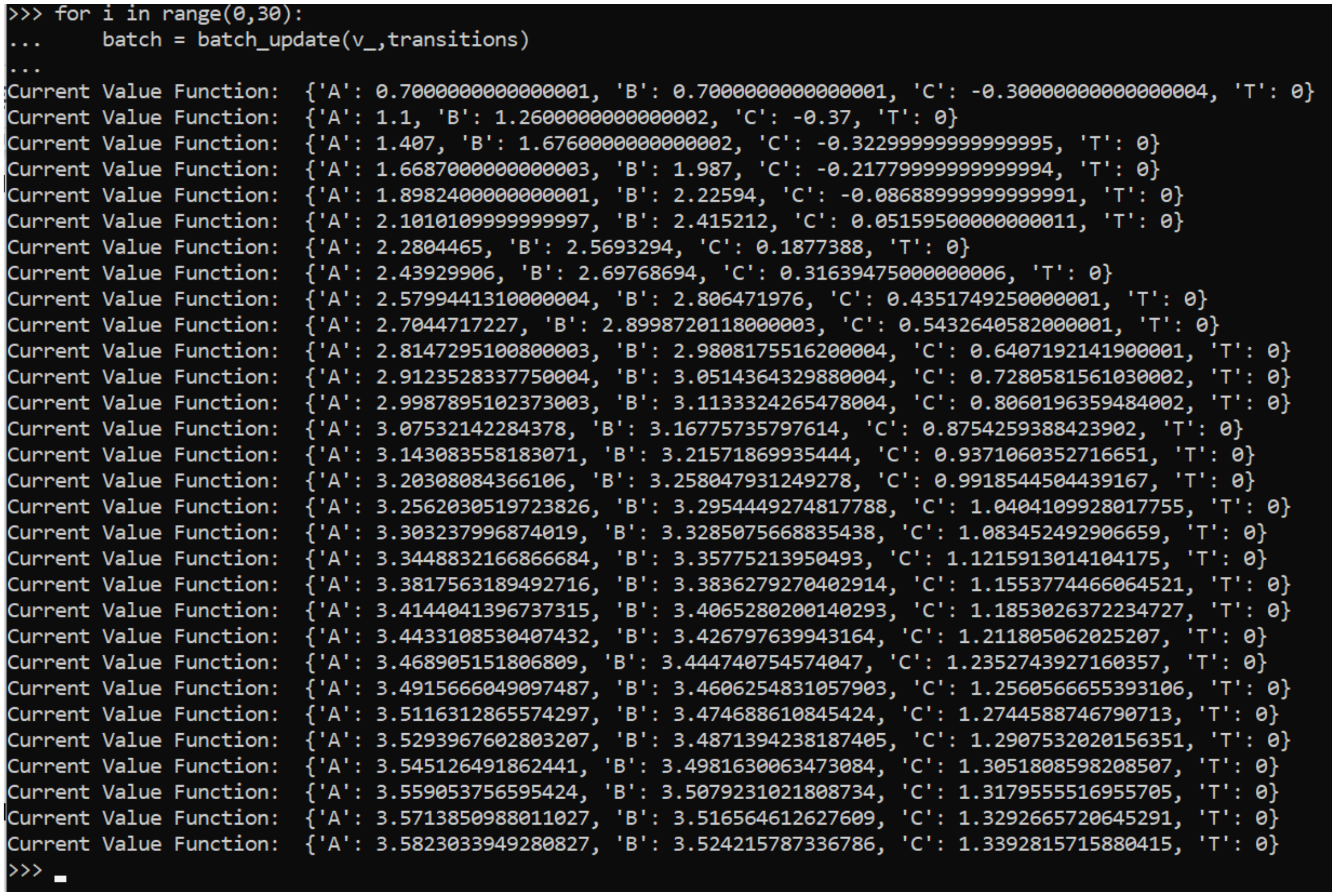
```
>>> for t in transitions:
...     v = update(v, t[0], t[1], t[2])
...
{'A': 0.30000000000000004, 'B': 0, 'C': 0, 'T': 0}
{'A': 0.30000000000000004, 'B': -0.06999999999999999, 'C': 0, 'T': 0}
{'A': 0.37000000000000005, 'B': -0.06999999999999999, 'C': 0, 'T': 0}
{'A': 0.37000000000000005, 'B': -0.06999999999999999, 'C': 0.037000000000000005, 'T': 0}
{'A': 0.7260000000000001, 'B': -0.06999999999999999, 'C': 0.037000000000000005, 'T': 0}
{'A': 0.7260000000000001, 'B': -0.063, 'C': 0.037000000000000005, 'T': 0}
{'A': 0.8571000000000001, 'B': -0.063, 'C': 0.037000000000000005, 'T': 0}
{'A': 0.8571000000000001, 'B': -0.063, 'C': -0.273, 'T': 0}
{'A': 0.8571000000000001, 'B': 0.12901, 'C': -0.273, 'T': 0}
{'A': 0.8440900000000001, 'B': 0.12901, 'C': -0.273, 'T': 0}
{'A': 0.8440900000000001, 'B': 0.12901, 'C': -0.24570000000000003, 'T': 0}
{'A': 0.3725820000000001, 'B': 0.12901, 'C': -0.24570000000000003, 'T': 0}
{'A': 0.3725820000000001, 'B': 0.42901000000000006, 'C': -0.24570000000000003, 'T': 0}
{'A': 0.3725820000000001, 'B': 0.82901, 'C': -0.24570000000000003, 'T': 0}
{'A': 0.3725820000000001, 'B': 0.746109, 'C': -0.24570000000000003, 'T': 0}
```

The batch TD(0) estimate for each state’s value.



```python
def batch_update(v, transitions):
    stored_updates = {"A":0,"B":0,"C":0,"T":0}
    # Accrue updates for all timesteps.
    for t in transitions:
        # Recall that the tuples are (s, s', r)
        s = t[0]
        sprime = t[1]
        r = t[2]
        stored_updates[s] += 0.1*(r + v[sprime] - v[s])
    # Apply updates all at once.
    for state in stored_updates.keys():
        v[state] += stored_updates[state]
    print("Current Value Function: ",v)
    return v
```

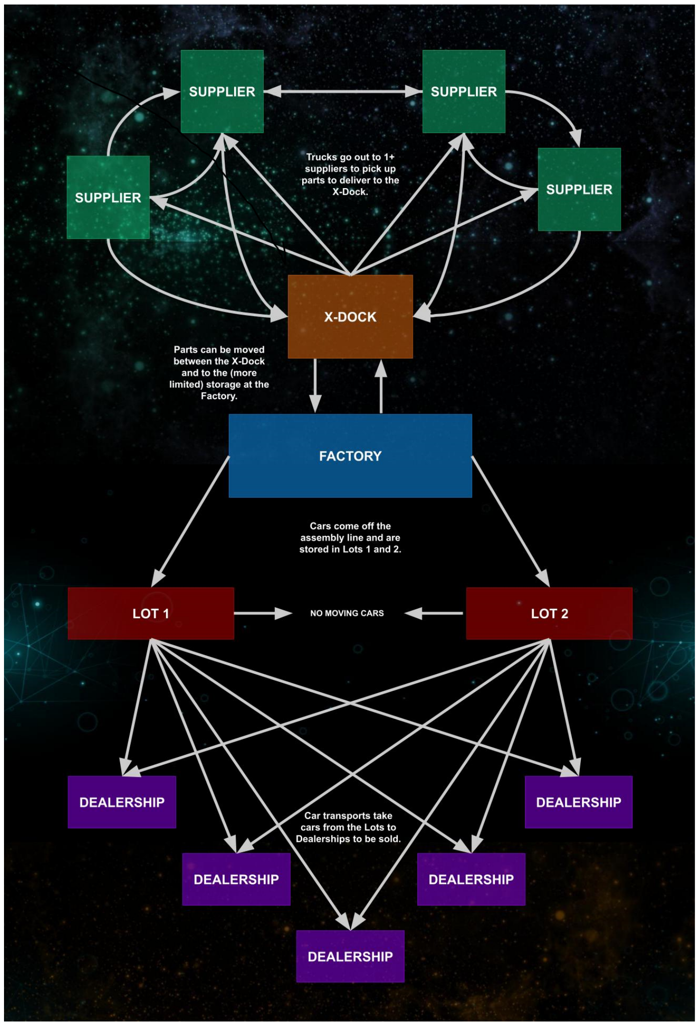
Did until (close to) convergence because once you have the function…



```
>>> for i in range(0,30):
...     batch = batch_update(v_,transitions)
...
Current Value Function:  {'A': 0.7000000000000001, 'B': 0.7000000000000001, 'C': -0.30000000000000004, 'T': 0}
Current Value Function:  {'A': 1.1, 'B': 1.2600000000000002, 'C': -0.37, 'T': 0}
Current Value Function:  {'A': 1.407, 'B': 1.6760000000000002, 'C': -0.32299999999999995, 'T': 0}
Current Value Function:  {'A': 1.6687000000000003, 'B': 1.987, 'C': -0.21779999999999994, 'T': 0}
Current Value Function:  {'A': 1.8982400000000001, 'B': 2.22594, 'C': -0.08688999999999991, 'T': 0}
Current Value Function:  {'A': 2.1010109999999997, 'B': 2.415212, 'C': 0.05159500000000011, 'T': 0}
Current Value Function:  {'A': 2.2804465, 'B': 2.5693294, 'C': 0.1877388, 'T': 0}
Current Value Function:  {'A': 2.43929906, 'B': 2.69768694, 'C': 0.31639475000000006, 'T': 0}
Current Value Function:  {'A': 2.5799441310000004, 'B': 2.806471976, 'C': 0.4351749250000001, 'T': 0}
Current Value Function:  {'A': 2.7044717227, 'B': 2.8998720118000003, 'C': 0.5432640582000001, 'T': 0}
Current Value Function:  {'A': 2.8147295100800003, 'B': 2.9808175516200004, 'C': 0.6407192141900001, 'T': 0}
Current Value Function:  {'A': 2.9123528337750004, 'B': 3.0514364329880004, 'C': 0.7280581561030002, 'T': 0}
Current Value Function:  {'A': 2.9987895102373003, 'B': 3.1133324265478004, 'C': 0.8060196359484002, 'T': 0}
Current Value Function:  {'A': 3.07532142284378, 'B': 3.16775735797614, 'C': 0.8754259388423902, 'T': 0}
Current Value Function:  {'A': 3.143083558183071, 'B': 3.21571869935444, 'C': 0.9371060352716651, 'T': 0}
Current Value Function:  {'A': 3.20308084366106, 'B': 3.258047931249278, 'C': 0.9918544504439167, 'T': 0}
Current Value Function:  {'A': 3.2562030519723826, 'B': 3.2954449274817788, 'C': 1.0404109928017755, 'T': 0}
Current Value Function:  {'A': 3.303237996874019, 'B': 3.3285075668835438, 'C': 1.08345249290659, 'T': 0}
Current Value Function:  {'A': 3.3448832166866684, 'B': 3.35775213950493, 'C': 1.1215913014104175, 'T': 0}
Current Value Function:  {'A': 3.3817563189492716, 'B': 3.3836279270402914, 'C': 1.1553774466064521, 'T': 0}
Current Value Function:  {'A': 3.4144041396737315, 'B': 3.4065280200140293, 'C': 1.1853026372234727, 'T': 0}
Current Value Function:  {'A': 3.4433108530407432, 'B': 3.426797639943164, 'C': 1.211805062025207, 'T': 0}
Current Value Function:  {'A': 3.468905151806809, 'B': 3.444740754574047, 'C': 1.2352743927160357, 'T': 0}
Current Value Function:  {'A': 3.4915666049097487, 'B': 3.4606254831057903, 'C': 1.2560566655393106, 'T': 0}
Current Value Function:  {'A': 3.5116312865574297, 'B': 3.474688610845424, 'C': 1.2744588746790713, 'T': 0}
Current Value Function:  {'A': 3.5293967602803207, 'B': 3.4871394238187405, 'C': 1.2907532020156351, 'T': 0}
Current Value Function:  {'A': 3.545126491862441, 'B': 3.4981630063473084, 'C': 1.3051808598208507, 'T': 0}
Current Value Function:  {'A': 3.559053756595424, 'B': 3.5079231021808734, 'C': 1.3179555516955705, 'T': 0}
Current Value Function:  {'A': 3.5713850988011027, 'B': 3.516564612627609, 'C': 1.3292665720645291, 'T': 0}
Current Value Function:  {'A': 3.5823033949280827, 'B': 3.524215787336786, 'C': 1.3392815715880415, 'T': 0}
>>>
```

Question 11: Audi Logistics

The year is 2025. You are working as a data scientist for Audi, a car company. You have been tasked with building a machine learning system to optimize the logistics for an area of southern Germany and adjacent countries. The details are below. Review the image "Audi Manufacturing, Southern Germany, 2025.jpeg".


SUPPLIER
SUPPLIER
SUPPLIER
SUPPLIER
Trucks go out to 1+ suppliers to pick up parts to deliver to the X-Dock.
X-DOCK
Parts can be moved between the X-Dock and to the (more limited) storage at the Factory.
FACTORY
Cars come off the assembly line and are stored in Lots 1 and 2.
LOT 1
NO MOVING CARS
LOT 2
DEALERSHIP
DEALERSHIP
Car transports take cars from the Lots to Dealerships to be sold.
DEALERSHIP
DEALERSHIP
DEALERSHIP

- **The Audi factory makes four different types of Audi cars.**

  Car Delivery Logistics
- **The cars produced by the factory are stored in two parking lots, lot A and lot B. As requests come in from dealerships in the area, cars are loaded onto specialized transports and delivered.**
- **Because the cars have no fuel in them, they cannot be moved between lots.**
- **Having a transport pick up cars from different lots adds a bit of time cost.**
- **Ideally, the transports will be fully loaded with cars and can deliver them as-needed to the dealerships.**
- **The cars to be delivered should be assigned to transports so that the transports can make the shortest, most efficient delivery runs possible.**

Car Manufacturing Side:

- **The factory handles when cars are produced and we may assume it will never produce cars it cannot store in the lots.**
- **The factory needs parts to make cars. Some stockpile of parts can be stored in the factory for immediate use, but most parts are stored in the "Cross-Dock" or "X-Dock", which is a parts warehouse. It is preferred that parts are stored in the X-Dock.**
- **Trucks run between the X-Dock and the factory, delivering parts as needed.**
- **Trucks also run between the X-Dock and various suppliers.**
- **Each supplier provides one type of part.**
- **Ideally, trucks run the shortest route between suppliers possible and return with a full load of parts (no wasted space, lowest fuel/time cost possible). The best possible situation is for a truck to go to one supplier and back, returning with a full load. However, this is often not an option as trucks need to visit multiple suppliers and attempt to minimize fuel and maximize load size.**

The Challenges

- **If the factory has too few parts, it cannot make the cars required and loses out on sales.**
- **If the factory has too many parts, the company has wasted capital and potentially cannot store the parts properly (which would waste more capital).**
- **If the trucks and/or transports are making inefficient runs, this wastes capital via excessive fuel costs and time delays.**

(a) **Describe the objectives of this task.**
(b) **Which parts of this are best addressed using RL? Are any parts of this better addressed using supervised learning? If so, which ones? Which algorithms would you use for each part?**
(c) **Describe the state space. Do not worry about constraining the size of the space. Make it as big as you need it to be to address the problem.**
(d) **Define a reward function for this problem and justify the logic of your choices.** ***Note: it is OK to refer to $ or fuel costs in a general way (e.g., dollars spent on fuel are negative reward). Don't worry about doing a perfect job specifying the rewards - it just needs to make logical sense.***
(e) **Describe the action space (you don't need to be completely specific as it will depend on the number of car models, car parts, suppliers, etc.). As with states, do not worry about constraining the size. Make it as big as you need it to be to address the problem.**

(a) We want the supplier trucks to make the most efficient trips possible, balancing trip length and load size. We want to make part use as efficient as possible, minimizing purchases of parts we don't use while keeping inventory as close to full as we can. We also want the car transports to make efficient trips, balancing trip length and load size like the supplier trucks.

(b) Almost all of this task is best addressed using RL. In particular, the supplier car trips, inventory management, and the transport trips. However, supervised learning could be used to enhance the efficacy of an RL controller by predicting sales volume (i.e., the future requirements of transports and future parts demand) and provide those predictions as a part of the controlling agent's state. These predictions could be used to potentially send out supply trucks in advance of demand, allowing transports to confidently load up early. This predictive information might even allow the controller to make tradeoffs, such as (1) making ostensibly inefficient trips that become optimal when orders come in when the truck is already on the road (freeing up inventory and saving time), or (2) providing "too many" of a given part so they are ready for the predicting spike in demand, saving time and potentially allowing more efficient use of truck space.

(c) The state space…

- The amount of each part at the X-Dock.
- The amount of each part at the factory.
- The amount of each type of car in both lots.
- The predicted sales for each type of car tomorrow (or over the next week, ... , would test).
- The predicted demand for each type of part tomorrow / next week / …
- The location, time-to-return (0 if at X-dock), current load composition, and current ‘goal’ of each truck (e.g., “at supplier A”, “en route to supplier B”).
- The location, time-to-return, current load composition, and current ‘goal’ of each transport.

(d) The reward function:

- ○ -1 for each gallon of diesel consumed.
- ○ -1 for each cubic foot of unused space on a truck’s return from delivery (partial rewards for partial cubic feet).
- ○ -1 for each cubic foot of unused space in the factory (partial rewards for partial cubic feet).
- ○ -2 for each cubic foot of unused space in the X-Dock (partial rewards for partial cubic feet).
- ○ -1 for each hour away from X-Dock for trucks and transports.
- ○ +1 for each part delivered.
- ○ -10 for each instance of a part being required but not available at the factory.
- ○ +1 for every car produced by the factory
- ○ +5 for every car delivered
- ○ -2 for every empty spot on a car transport during delivery.

(e) Let’s see...what do we have available? Let’s start with the supply trucks.

- i. Select a truck to go out on a run.
- ii. Select the location(s) for the truck to go to… this might be better framed as each truck’s next step. A truck arrives at a location and waits for orders. However, this would prevent recalling trucks if a situation changes… hm.
- Iii. Select how much of a given part to order.

Given these aspects, it might be best to have continuous actions… but let’s assume the actions are tabular, like a tuple: (num_of_part_A, ..., num_of_part_X). The locations to visit are implicit in the parts we are going to get, as each supplier gives only one part. The order of visits is not contained here, however, so we return to the “one location at a time” issue. I think I would first frame the action space in this way and run simulations to see if the agent was able to learn to avoid inefficient/impulsive “GO GET THIS” initial truck directions.

We might have a separate set of actions that can be chosen concurrently with the supply run actions. Which transport, where that transport goes, and what that transport is carrying.