AI Planning

1. Bellman Equation

- Apply concept of dynamic programming and optimal substructure to do **policy evaluation**

$$V(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) V(s')$$

- s: Any given state
- s': Any possible next state
- a: Any possible action at state s
- V(s): Value of state
- R(s); Reward of state
- A(s): Set of all possible actions at state s
- γ: Exploration Factor
- P(s'|s,a): Probability of s' given (s,a)
- V(s'): Value of s'
- **Value of a state** = **the state's value** + Exploration Factor x **the value of the best action at that state**
    - What is the value of the best action?
        - Given (s,a), Sum all possible next states x the probability of reaching those states.

Bellman Equation Variants

R(s,a,s')

$$V(s) = \max_{a \in A} \sum_{s' \in S} T(s, a, s')[R(s, a, s') + \gamma V(s')]$$

- Reward for transitioning from s to s' via a

- Choose the action that has gives the highest reward (based on transition), and discounted future reward for reaching that state

R(s,a)

$$V(s) = \max_{a \in A} R(s,a) + \gamma \sum_{s' \in S} T(s,a,s')V(s')$$

- Reward for taking action a at state s
- Choose the action that has gives the highest reward, and also factor in the discounted next state's reward

Converting between reward functions

$$R'(s,a) = \sum_{s' \in S} T(s,a,s')R(s,a,s')$$

- The reward for taking an action is the sum of (all its possible transitions) x (related transition rewards)

$$R'(post(s,a)) = \gamma^{-1/2}R(s,a)$$

- s' is abstracted out to "post(s,a)", which refers to the "post-state" for every (s,a). In this R:

```
// (s,a) always goes to post(s,a)
T'(s,a,post(s,a)) = 1
// The probability is factored in here
T'(post(s,a),b,s') = T(s,a,s')
// The states themselves don't have rewards
R'(s) = 0
// The reward for taking the action is put into the pseudo "post-state"
R'(post(s,a)) = gamma^(-1/2)R(s,a)
gamma' = gamma^(1/2)
```

2. Value Iteration Algorithm

- Repeatedly update the Utility function U(s) with the bellman equation / bellman update above.

$$U_{t+1}(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_t(s')$$

- Notice that U is only updated after every single state has been looped through.

```
// Returns Utility function U(s)
function Value-Iteration(mdp, err_threshold)
    loop:
        U_t = U_t+1, max_delta = 0
        for all states s in S:
            Update U_t+1 with bellman equation.
            Update max_delta if abs(U(s) - U_t+1(s)) exceed max_delta
        break loop if max_delta less than err_threshold(1-gamma)/gamma
    return U_t
```

3. Policy Evaluation Algorithm

- Same as Value Iteration.
    - But we want to **evaluate a given policy pi_i**. Thus, **we don't need to take the max, we just need to use the policy.**
    - Hence: Instead of taking the best action, take the action that the policy thinks is the best action

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

4. Policy Iteration Algorithm

- We don't need to calculate U(s) so accurately if we just want to find the optimal policy
- Two step approach:

1. **Init**: Start with initial policy pi_0
2. **Policy Evaluation**: Calculate Utility function U_i given current policy. See Policy Evaluation above.
3. **Policy Improvement**: Calculate new policy pi_i+1 using U_i.

4. **Termination**: Repeat until the new and old policy are the same (no change).

```
// Returns policy pi
function Policy-Iteration(mdp)
    U = set all to 0
    pi = random policy
    loop
        unchanged = true
        U_i = Policy-Evaluation(pi, U, mdp)
        for all states s in S:
            If the best action at s is different:
                Update pi[s]
                unchanged = false
        break if unchanged
    return pi
```

The check for the best action is done as follows:

$$\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) U_i(s') > \sum_{s'} P(s' \mid s, \pi_i(s)) U_i(s')$$

- Identify the action with highest utility at s
- Check if it is the same action taken in the policy

5. Modified Policy Iteration

- Only do k iterations instead of until no change

6. Model-Based Prediction

- Prediction: Given traces of a policy and the final reward, learn the utility function by constructing the model from the data using an agent:
    - Learn **transition model** and **reward function** using an **Adaptive Dynamic Programming** (ADP) agent.
    - **Calculate the Utility function.**

```
// persistent variables
```

```
s_prev

pi: policy

mdp: current constructed model, rewards and discount

U: table of utilities

Nsa: Table keeping count of number of times at state s, action a was
 taken

Nsas': Table keeping count of number of times the next state was s'
 given (s,a) (thus s,a,s')

// Called everytime a new percept is observed by the agent
// Percept: (previous state s, current state s', current reward r')
function PASSIVE-ADP-AGENT(percept)
    if s_curr is new:
        U[s_curr] = r_curr
        R[s_curr] = r_curr
    if s_prev not null:
        // Increment counters
        Nsa[s_prev,a]++
        Ns'sa[s_curr,s_prev,a]++

        // Update transition function
        // for every known reachable state by (s,a)
        for every state where Ns'sa[state,s,a] > 0:
            // # of times state happened when action was taken
            T(s,a,s') = Ns'as[s,a,s'] / Nsa[s,a]
    U = Policy-Evaluation(pi, U, mdp)
    if s_curr is terminal:
        s_prev,a = null
    else
        s_prev,a = s_curr, pi[s_curr]
    return a
```

7. Model-Based Control

- Learn the policy, not the utility
- Just replace Policy-Evaluation with Policy-Iteration
- The policy no longer stays fixed but changes as transitions and rewards learnt
- Note that however the algorithm is greedy and may not return the optimal value

- Hence must do e-greedy exploration (choose a greedy action with 1-e probability and random action with e probability)
    - Greedy in the Limit of Infinite Exploration (GLIE), e = 1/t
    - Start with a high epsilon, then slowly reduce the number of random actions you take (by reducing the value of e) with every policy iteration as you become more and more certain you have an optimal algorithm

8. Monte Carlo Learning

- Monte Carlo Learning (Direct Utility Estimation)
    - Given a series of states as a "trial" e.g. this is a trial: (1)-.04→(2)-.04→(2)-.04→(3)-.04→(4)-.04…→(5)+1
    - Keep a running reward average for every state; after infinite trials, sample average will converge to expected value
        - e.g. for the above trial, maybe state (1) has a sample total reward of 0.72
            - Calculate by taking sum of rewards from that state to the end)
        - If a state is visited multiple times in a trial e.g. state (2):
            - first-visit: take the sum of rewards from only on the first visit per trial
            - every visit: take multiple sums for every time the state was visited in a trial.

Assume state s encountered k times with k returns, and each summed reward is stored in G_i(s).

- Note that G_i(s), given infinitely many i iterations, will converge to the true value; hence G_i(s) - U_k-1(s) is the MC error

$$U_k(s) = \frac{1}{k}\sum_{i=1}^{k} G_i(s)$$

$$U_k(s) = \frac{1}{k}G_k(s) + \frac{1}{k}\sum_{i=1}^{k-1} G_i(s)$$

$$U_k(s) = \frac{1}{k}G_k(s) + \frac{1}{k} * (k-1)U_{k-1}(s)$$

$$U_k(s) = \frac{k}{k}U_{k-1}(s) - \frac{1}{k}U_{k-1}(s) + \frac{1}{k}G_k(s)$$

$$U_k(s) = U_{k-1}(s) + \frac{1}{k}(G_k(s) - U_{k-1}(s))$$

- The difference between the current U_k(s) and the previous U(s) is the prediction error (if you want to mnimize absolute loss, you can use median instead of average)
- Note that U(s) here can also apply to U(s,a), and can also be renamed as Q(s) and Q(s,a)

```
# Monte Carlo Prediction (Learning U(s))
After every trial:
    Take every state (or first state occurence) and calculate its value
 sum
    Increment Ns
    U_k(s) = U_k-1(s) + 1/k(sum - U_k-1(s))

# Monte Carlo Control (Learning pi)
After every trial:
    Take every (s,a):
        newR(s,a) = take every/1st (s,a) and calculate value sum
        Increment Ns
        Q(s,a) = Q_k-1(s,a) + 1/k(newR(s,a) - Q_k-1(s,a))
    For every s:
        pi(s) = Take action that maximizes Q(s,a)
```

Advantages

- Simple
- Unbiased estimate

Disadvantages

- Must wait until full trial is done in order to perform learning

- High variance (reward is the sum of many rewards along the trial), so need many trials to get it right

9. Temporal Difference Learning

- Very similar to Monte Carlo Learning
- Replace the

$$U^{\pi}(s) = U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s^{'}) - U\pi(s))$$

- Transition from state s to s'.
- Alpha is the learning rate.
    - Converges if alpha decreases with the number of times the state has been visited (think GLIE)
- $R(s) + \gamma U^{\pi}(s^{'})$: **Temporal Difference target**
    - (this is basically bellman update, policy evaluation)
- $R(s) + \gamma U^{\pi}(s^{'}) - U^{\pi}(s)$: **Temporal Difference error**
    - (amount of utility change from previous state s to current state s')
    - difference between the estimated reward at any given state or time step and the actual reward received
    - Utility of current state + discounted future reward(future state) - expectedRewardWithIncludesFutureRewards(s)

```
# Prediction
TD-Agent:
for every percept (curr_state s', immediate reward r')
if s' is new:
    U[s'] = r'
if s not null:
    Ns[s]++
    U[s] = U[s] + alpha*Ns[s]*(r+gamma*U[s']-U[s])
if s' terminal:
    s,a,r = null
else: # update prev data
    s,a,r = s',pi[s'],r'
```

```
return a
```

Advantages

- Can learn with every step (in a trial)
- Usually converges faster in practice

Disadvantages

- Online; lower variance, but estimate on how good your estimate is; biased
- Assumes MDP
- Error can go in any direction

SARSA uses TD learning w.r.t. a specific policy:

$$U^{\pi}(s,a) = U^{\pi}(s,a) + \alpha(R(s,a) + \gamma U^{\pi}(s', \pi((s')) - U\pi(s,a))$$

Q-Learning uses TD learning w.r.t. the optimal policy (s' is the next state after (s,a) is taken):

$$Q(s,a) = Q(s,a) + \alpha(R(s,a) + \gamma \max_{a \in A(s')} Q(s',a) - Q(s,a))$$

10. n-step TD, TD(lambda)

Alister Reis has a very good blog post on this https://amreis.github.io/ml/reinf-learn/2017/11/02/reinforcement-learning-eligibility-traces.html