Planning by Dynamic Programming

Santhosh Kumar G

Wednesday 20th August, 2025

1 Introduction to Dynamic Programming (DP)

Dynamic Programming is a powerful method for solving complex problems by breaking them down into simpler subproblems. It's particularly effective for problems that exhibit two key properties:

- **Optimal Substructure**: The optimal solution to the overall problem can be constructed from the optimal solutions of its subproblems. This is also known as the *principle of optimality.*
- **Overlapping Subproblems**: The same subproblems are encountered and solved multiple times. DP solves each subproblem only once and stores its solution to avoid redundant computations.

Markov Decision Processes (MDPs) are a perfect fit for Dynamic Programming. The **Bellman equation** provides the recursive decomposition needed for optimal substructure, and the **value function** serves as a cache to store and reuse solutions to overlapping subproblems.

2 Planning in MDPs

In the context of reinforcement learning, Dynamic Programming is used for planning—calculating the value functions and finding optimal policies when you have a complete model of the environment (the MDP). There are two main types of planning problems that DP helps us solve:

- Prediction: Given an MDP and a specific policy, the goal is to predict the long-term value of each state.
- Control: Given an MDP, the goal is to find the optimal policy that maximizes the long-term reward.

3 Core DP Algorithms for Planning

The two main algorithms for planning in MDPs are Policy Iteration and Value Iteration. Both are iterative approaches that use generalized policy iteration (GPI), an elegant concept in which we have an interaction between policy evaluation and policy improvement.

3.1 Policy Evaluation

Policy evaluation is the process of computing the state-value function, v_π, for a given policy, π. It answers the question: *How good is this policy?*

Algorithm 1 Algorithm: Policy Evaluation

```
This is an iterative approach based on the Bellman Expectation Equation.
Initialize $v(s)$ arbitrarily (e.g., all zeros) for all $s \in S$.

  Initialize a change counter, $\Delta = 0$.
  for each state $s \in S$:
    1. Store the old value: $v_{old} = v(s)$
    2. Update the value using the Bellman Expectation Equation:
       $v(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$
    3. Update the change counter: $\Delta = max(\Delta, |v_{old} - v(s)|)$
  If $\Delta < \theta$ (a small threshold), break the loop.
  The final v(s) values represent $v_\pi$
```

3.2 Example: The Small Gridworld

This classic example helps visualize policy evaluation. In a simple gridworld, states are the grid squares. The agent can move North, South, East, or West. A policy might be to move randomly with equal probability in all directions. The goal of policy evaluation is to determine the value of each square under this random policy.

By iteratively applying the Bellman Expectation Equation, the values of the squares will converge. Squares closer to the terminal state (the goal) will have higher values, while squares near a "hole" or "cliff" will have lower values.

4 Policy Iteration

Policy Iteration is a full control algorithm that finds the optimal policy, π^*, and the optimal value function, v It consists of two main alternating steps:

- **Policy Evaluation**: Evaluate the current policy,π, to compute its value function, v_π
- Policy Improvement: Greedily improve the policy with respect tov_π. For each state, choose the action that maximizes the expected value, creating a new policy, π'. This is presented as Algorithm 2 below.

4.1 Example: Jack's Car Rental

This is a well-known example of Policy Iteration. The problem involves a car rental company with two locations. The state is the number of cars at each location. The

Algorithm 2 Algorithm: Policy Iteration

```
Initialize $\pi$ and $v(s)$ arbitrarily (e.g., all zeros).

  LOOP:
    1. **Policy Evaluation**: Compute $v_pi$ for the current policy $\pi$.
    2. **Policy Improvement**: for each state s:
       $\pi'(s) \leftarrow arg\ max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$
    3. If $\pi'$ is the same as $\pi$, then the optimal policy has been found, and the loop terminates. Otherwise, set $\pi \leftarrow \pi'$ and repeat.
```

actions are the number of cars to move between locations overnight. The reward is based on rental profits. The optimal policy determines the best number of cars to move to maximize profit over time. Policy Iteration finds this policy by repeatedly evaluating the current policy (e.g., ”move one car”) and then improving it (e.g., ”move two cars”) until no further improvement is possible.

5 Value Iteration

Value Iteration is another control algorithm that finds the optimal policy and value function. Unlike Policy Iteration, it combines the policy evaluation and improvement steps into a single update. It essentially finds the optimal value function first, and from that, the optimal policy can be derived.

Algorithm 3 Algorithm: Value Iteration

```
This is approach is based on the Bellman Optimality Equation.
Initialize $v(s)$ arbitrarily (e.g., all zeros) for all $s \in S$.

  Initialize a change counter, $\Delta = 0$.
  for each state $s \in S$:
  Loop until convergence:
    1. Store the old value: $v_{old} = v(s)$
    2. Update the value using the Bellman Optimality Equation:
       $v(s) \leftarrow max_a \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v(s')]$
    3. Update the change counter: $\Delta = max(\Delta, |v_{old} - v(s)|)$ If $\Delta < \theta$ (a small threshold), break the loop.
    4. Once the values converge, the optimal policy can be found by a single policy improvement step:
       $\pi^*(s) \leftarrow arg\ max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')]$
```

5.1 Example: The Shortest Path Problem

Value Iteration is a great fit for shortest path problems. The states are the nodes of a graph, and actions are moving along the edges. The reward is the negative of the edge weight. Value Iteration iteratively updates the value of each node (representing the shortest path distance) until it converges to the true shortest path. The optimal policy is then to simply move along the path that leads to the lowest value (shortest distance).

6 Policy Iteration vs. Value Iteration

Both algorithms find the optimal policy, but they differ in their approach and efficiency. Policy Iteration has two distinct phases per iteration: a full policy evaluation sweep followed by a policy improvement step. It typically requires fewer iterations to converge but each iteration is more computationally expensive due to the full policy evaluation step. Value Iteration has only one phase per iteration. It is computationally cheaper per iteration, but often requires more iterations to converge. In summary, Policy Iteration is often faster when the state space is small, while Value Iteration is preferred for larger state spaces where the cost of a full policy evaluation sweep is prohibitive.