National Taiwan University 114-1 (Fall 2025)
Department of Electrical Engineering Reinforcement Learning
Name: James, Tan
ID: R13921031

Assignment 1 Report

Q1. What methods have you tried for async DP? Compare their performance.

I implemented and tested the three asynchronous dynamic programming methods taught in class: In-Place Dynamic Programming, Prioritized Sweeping, and Real-Time Dynamic Programming.

Method 1: In-Place Dynamic Programming. This method eliminates the need for two copies of the value function by updating values immediately. The algorithm sweeps through all states sequentially:

$$V(s) \leftarrow \max_a \left(R_s^a + \gamma \sum_{s'} P_{ss'}^a V(s') \right), \quad \forall s \in S.$$

Unlike synchronous value iteration which maintains separate V_{old} and V_{new} arrays, in-place updates use the most recent values immediately. Later states in the sweep benefit from earlier updates within the same iteration, potentially accelerating convergence.

Method 2: Prioritized Sweeping. This method selects states for backup based on the magnitude of their Bellman error, using a priority queue to focus computation on states where updates matter most:

$$\text{Priority}(s) = \left| \max_a \left(R_s^a + \gamma \sum_{s'} P_{ss'}^a V(s') \right) - V(s) \right|.$$

After updating state s, the algorithm recomputes priorities for all predecessor states and adds them to the queue if their priority exceeds threshold θ. This implementation queries the environment during both initialization and planning to compute transition dynamics.

Method 3: Real-Time Dynamic Programming. This method uses agent experience to guide state selection, backing up only states encountered during trajectory execution. Starting from the initial state, the agent follows a greedy policy with respect to current values. After each transition $(s_t, a_t, r_{t+1}, s_{t+1})$, state s_t is backed up:

$$V(s_t) \leftarrow \max_a \left(R_{s_t}^a + \gamma \sum_{s'} P_{s_t s'}^a V(s') \right).$$

The algorithm runs multiple episodes until values stabilize along typical trajectories from start to goal.

Comparison. The table below summarizes the performance of all implemented methods. Among the class methods, In-Place DP performs best with 1,056 steps, followed by Real-Time DP with 1,628 steps. Prioritized Sweeping requires the most steps (2,000) because it computes Q-values for all actions at each state during both queue operations and value updates. Real-Time DP focuses on relevant states but requires multiple episodes for convergence. In-Place DP provides a good balance with systematic state coverage and immediate value propagation.

Method	**Environment Steps**	**Method Type**
Policy Iteration	3,256	Synchronous
Value Iteration	1,144	Synchronous
In-Place DP	1,056	Async (Class)
Prioritized Sweeping	2,000	Async (Class)
Real-Time DP	1,628	Async (Class)
Model-Based PS (Novel)	**88**	**Async (Novel)**

Table 1: Performance comparison on test maze (22 states, 4 actions).

The novel Model-Based Prioritized Sweeping method achieves 88 steps, representing 12× speedup over In-Place DP, 22.73× speedup over Prioritized Sweeping, and 18.50× speedup over Real-Time DP.

Q2. What is your final method? How is it better than other methods you've tried?

My final method is **Model-Based Prioritized Sweeping**, a novel variant that fundamentally separates model learning from planning to minimize environment interactions.

Algorithm Description. The method operates in three distinct phases:

Phase 1 – Model Learning: Build a complete deterministic model by querying each state-action pair exactly once:

$$\mathcal{M}(s,a) = (s', r, d) \text{ where } (s', r, d) = \text{env.step}(s,a), \quad \forall s \in S, a \in A.$$

For the test maze with 22 states and 4 actions, this requires exactly $22 \times 4 = 88$ environment steps.

Phase 2 – Planning: Use the cached model for all subsequent computations. Initialize a priority queue with states having Bellman error above threshold θ. Iteratively update the highest-priority state:

$$V(s) \leftarrow \max_a \left[r(s,a) + \gamma V(s'(s,a))(1 - d(s,a)) \right], \tag{1}$$

where all quantities $r(s,a)$, $s'(s,a)$, and $d(s,a)$ are retrieved from cached model $\mathcal{M}$ without environment queries. After each update, recompute priorities for predecessor states and update the queue. This entire phase uses **zero** additional environment steps.

Phase 3 – Policy Extraction: Compute the greedy policy using final values and the cached model.

How It Differs from Class Methods. The table below summarizes the key distinction between my novel method and the three async methods from class.

Method	When Environment is Queried	Steps
In-Place DP	Every sweep through all states	1,056
Prioritized Sweeping	During priority computation and value updates	2,000
Real-Time DP	During trajectory execution across episodes	1,628
Model-Based PS	**Only during initial model building**	**88**

Table 2: Comparison of environment query patterns across async DP methods.

My method fundamentally differs by **decoupling model acquisition from planning**. After building the model with $|S| \times |A|$ queries, all planning operations access cached transitions with zero environment steps. This architectural change transforms prioritized sweeping from an online algorithm (interleaving queries with planning) into a model-based algorithm (query once, plan offline).

Performance Advantage. The table below presents the speedup achieved by the novel method over all baseline approaches.

Baseline Method	Steps	Speedup
Policy Iteration	3,256	37.00×
Value Iteration	1,144	13.00×
In-Place DP	1,056	12.00×
Real-Time DP	1,628	18.50×
Prioritized Sweeping	2,000	22.73×
Model-Based PS (Novel)	**88**	–

Table 3: Speedup of Model-Based PS over baseline methods.

Why It Works Better. The method exploits GridWorld's deterministic dynamics: each (s, a) pair maps to exactly one outcome (s', r, d). Traditional methods repeatedly query identical state-action pairs during iterative planning. By caching all transitions upfront, my method eliminates this redundancy entirely. The fixed cost of $|S| \times |A|$ environment steps pays for complete model knowledge, after which planning proceeds with zero additional queries. Combined with prioritized updates that focus on high-error states, this achieves optimal convergence with minimal environment interaction—exactly $|S| \times |A|$ steps, which is the theoretical minimum for any model-based approach in deterministic environments.