**ASSIGNMENT 3**

**1. Introduction**

In this assignment I worked with the FrozenLake game (8x8, “slippery” version). The agent starts in one corner and tries to reach the goal in the opposite corner without falling into holes. Because the ice is slippery, even if the agent chooses a direction, it doesn’t always move the way, so the game is uncertain.

I did two kinds of experiments:

1. Part 1 (baseline): I tested 10 different fixed random policies( simple rules that always pick the same action for each square). For each policy I ran 100 trails, and in each trail the agent played 10,000 episodes. I recorded how many times it reached the goal and kept the best two policies, showing their action grids ad goal histograms.
2. Part 2 (smart policy): I used Value Iteration to compute a better policy that plans. Then I evaluated this policy in the same way (100 trails x 10,000 episodes) and showed its action grid, a heatmap of the state values V (s), and a histogram of results.

In short: Part 1 shows how weak random fixed strategies are on a slippery lake. Part 2 shows how a planning method ( Value Iteration) dramatically improves the chance of reaching the goal, even with randomness.

**2. Code Architecture**

**a. Briefly describe how your custom code works.**

1. part\_one() – builds 10 fixed random policies and evaluates each one (100 experiments x 10,000 episodes). Prints summary and top-2 seeds.

2. part\_two() - runs Value Iteration to compute V(s), extarcts a greedy optimal policy, then evaluates it (100 x 10,000).

3. Artifacts – calls:

* save\_artifacts\_for\_top2\_seeds() 🡪 policy grids + histograms + CSVs for the two best random policies.
* save\_optimal\_artifacts() 🡪 V(s) heatmap, optimal policy grid, histogram, CSV.

A diagram of a flowchart

Description automatically generated

A screen shot of a computer code

Description automatically generated

This creates a reproducible table that maps each state (0..63) to a single action. This is the “baseline” policy in part1.

A screenshot of a computer

Description automatically generated

This plays many episodes with a fixed policy and counts how many times the agent reaches the goal.

A screen shot of a computer

Description automatically generated

This runs many “experiments” in parallel processes and returns a vector of goal counts(length =100).

A screen shot of a computer

Description automatically generated

This solves the Bellman optimality equations by iteratively improving V(s) until changes are tiny (theta). Gamma=1.0 matches the assignment (no living cost).

A screen shot of a computer

Description automatically generated

This picks, for each state, the action with the largest Q(s, a) computed from the converged V.

**3. Computer Specs**

|  |  |
| --- | --- |
| Machine | MacBook Air(Apple Silicon) |
| CPU | 8-core Apple Silicon(physical cores:8, logical cores:8) |
| RAM | 8 GB |
| Operating System | macOS 15.6 (Build 24G84) |
| Python | 3.13.2 (virtual environment .venv) |
| Libraries used | Gymnasium, NumPy, Matplotlib |

Reproducibility notes: All experiments for Part 1 and Part 2 including artifacts generation were on this single machine and within the same Python virtual environment.

**4. Part 1 Results.**

**a. Experimental Results:**

**i. Include the top 2 policies found in your experiment.**

Setup. I evaluated 10 fixed, deterministic random policies(seeds: 3, 7,11, 13, 17, 19, 23, 29, 31, 37) on FrozenLake-v1 (8x8, slippery=True). Each policy was run for 100 experiments x 10,000 episodes (the same setting used throughout).

Top-2 policies (by mean goals per 10,000 episodes\_:

1. Seed 13

* Mean: 253.24 goals
* Std dev: 15.64
* 95% CI: [250.18, 256.30]
* Success rate: ≈ 2.53% per episode
* Raw data: 260, 250, 251, 291, 282, 254, 243, 257, 244, 277, 243, 262, 259, 256, 249, 245, 265, 267, 254, 271, 265, 251, 221, 272, 228, 231, 264, 240, 254, 253, 261, 252, 235, 288, 251, 244, 256, 273, 294, 261, 263, 254, 252, 273, 232, 249, 236, 256, 233, 250, 216, 253, 228, 263, 266, 232, 224, 264, 262, 245, 264, 232, 251, 260, 271, 246, 244, 266, 266, 259, 263, 242, 236, 249, 247, 277, 221, 255, 227, 254, 244, 258, 275, 253, 254, 247, 265, 252, 260, 249, 268, 249, 280, 263, 254, 253, 242, 218, 250, 235
* Summary row:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| seed | experiments | Episodes | Mean\_goals | Std\_goals |
| 13 | 100 | 10000 | 253.240000 | 15.641902 |
| 17 | 100 | 10000 | 24.450000 | 5.224989 |
| 13 | 100 | 10000 | 253.240000 | 15.641902 |
| 17 | 100 | 10000 | 24.450000 | 5.224989 |

2. Seed 17

* Mean: 24.45 goals
* Std dev: 5.22
* 95% CI : [23.43, 25.47]
* Success rate : ≈ 0.24% per episode
* Raw data: 18, 34, 27, 25, 19, 33, 27, 25, 21, 20, 19, 23, 31, 18, 23, 18, 24, 30, 21, 27, 24, 27, 17, 23, 25, 35, 28, 30, 25, 17, 25, 25, 21, 27, 26, 20, 20, 15, 36, 22, 30, 23, 21, 22, 28, 29, 20, 32, 27, 16, 27, 24, 20, 26, 18, 25, 26, 27, 29, 32, 14, 30, 32, 27, 28, 23, 19, 26, 15, 28, 31, 25, 30, 21, 29, 26, 22, 20, 27, 20, 22, 27, 27, 21, 23, 26, 23, 27, 30, 24, 30, 15, 24, 39, 12, 24, 26, 28, 15, 16
* Summary row:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| seed | experiments | Episodes | Mean\_goals | Std\_goals |
| 13 | 100 | 10000 | 253.240000 | 15.641902 |
| 17 | 100 | 10000 | 24.450000 | 5.224989 |
| 13 | 100 | 10000 | 253.240000 | 15.641902 |
| 17 | 100 | 10000 | 24.450000 | 5.224989 |

**ii. Provide all require statistics and plots for each of these policies.**

**A screenshot of a computer

Description automatically generated**

**Fig: policy\_13\_grid**

**A screenshot of a computer

Description automatically generated**

**Fig:policy\_13\_hist**

**A screenshot of a computer

Description automatically generated**

**Fig:policy\_17\_grid**

**A screen shot of a graph

Description automatically generated**

**Fig:policy\_17\_hist**

**b. An in-depth analysis of these results**

**i. What do you think that makes the top 2 policies you found better than the other**

**policies that you tried?**

* Slip-robust path: On slippery FrozenLake, moves often slip sideways. Seed-13’s action grid happens to trace a safe corridor where even sideways slips usually stay on safe tiles🡪 steady but small success(≈2.53% per episode).
* Seed 17 is similar but less consistent. It brushes riskier zones or loops more often, much lower success (≈0.24%).
* Many other seeds steer into hole clusters or create trap loops, so they score ≈0.

**Histograms match this:**

* Seed - 13: centered near 253 goals/10k with moderate spread (std ~15.6).
* Seed – 17: centered near 24.5 with tighter spread (std ~ 5.2).

**ii. Is it possible to start from a random policy and gradually improve it by making**

**changes manually? Please explain.**

* Somewhat, by nudging arrows along a safe lane and re-testing.
* But its fragile and unscalable: slips make local edits affect neighbors unpredictably. Algorithmic methods (Value/Policy Iteration, RL) are the reliable way.

**iii. Anything else that you consider relevant/interesting.**

* Huge gap: Seed – 13 (~253) is ~10x better than seed - 17 (~24) and far above most seeds (~0).
* Map structure rules: Only very specific layouts are slip – tolerant random search rarely finds them.
* Reproducibility: Fixed seeds + CSVs keep plots and reported stats aligned**.**

**5. Part 2 Results.**

**a. Experimental Results:**

**i. Include the optimal state values found by your value iteration algorithm.**

Setup. Value Iteration on FrozenLake-v1 (8x8, slippery=True), γ=1.0, θ=1e-10. Greedy policy extracted from the converged V \\*.

**Performance (from your artifact run):**

* Mean goals / experiment: 5088.32
* Std dev: 59.48
* 95% CI: [5076.66, 5099.98] (≈ mean ± 1.96·59.48/√100)
* Success per episode: ≈ 50.88%

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Policy | Experiments | Episodes | Mean\_goals | Std\_goals |
| Optimal | 100 | 10000 | 5088.320000 | 59.482258 |

The V(s) heatmap shows a clear gradient rising toward the goal. Low values sit near holes. The policy grid forms a stable route that avoids hole clusters while still progressing toward the goal.

A screenshot of a computer

Description automatically generated

Fig: optimal\_values\_grid

A screenshot of a computer

Description automatically generated

Fig: optimal\_policy\_hist

**A screenshot of a computer

Description automatically generated**

Fig: optimal\_policy\_grid

**b. An in-depth analysis of these results**

**i. Do the actions in the optimal policy make sense to you? Please explain your**

**answer in detail.**

Yes. Because slips send you sideways with probability 1/3 each, the optimal arrows:

* Track a safe corridor toward the goal
* Keep perpendicular slip tiles safe whenever possible
* Avoid “funnels” that route slips into holes.

This matches the high-value band in the V(s) heatmap and explains the strong, consistent results.

**ii. How much better is this policy in comparison to the best ones that you found**

**using random actions.**

The best fixed random policies from Part 1:

* Seed 13: mean 253.24 (≈ 2.53% per episode)
* Seed 17: mean 24.45 (≈ 0.24% per episode)

Comparisons to optimal (mean 5088.32)

* vs seed 13: ~20.1x higher (5088.32 / 253.24)
* vs seed 17: ~208x higher (5088.32 / 24.45)

**iii. Anything else that you consider relevant/interesting.**

* Stability: Despite stochastic slip, the optimal policy’s variance is modest (std ~59 on a mean ~ 5088), so performance is both high and consistent across 100 repeats.
* Map structure: Value Iteration “discovers” the safe lane automatically. Random fixed tables rarely align with this structure, explaining the huge gap from Part 1.
* Convergence choice: Using γ=1.0 with absorbing terminals yields a clean value gradient and straightforward greedy extraction.

**7. New Application**

**MDP for a small Bakery’s Oven Scheduling**

**Scenario:** Consider a neighborhood bakery with a single main oven, baking a few items all morning-croissants, cookies, and sandwich rolls. Customer do not turn up at perfectly times, and item go stale if they sit too long. The baker must decide, batch by batch, what to bake next so that there’s fresh product when people show up but without wasting ingredients or turning customers away.

**Why this is sequential(MDP- worthy):** Every baking decision determines what is available 15-20 minutes later, that impacts sales and freshness. Demand is a bit uncertain, with some mornings busier or slower, so you cannot bake everything at once**.**

**MDP Sketch:**

**State(s):** current time slot, e.g., every 10-15 minutes oven free/busy status, how many trays of each item are on track (and their “freshness age”), a short forecast of likely demand for the next hour (e.g., based on a day of week/time), and inventory of dough/ingredients.

**Action(a):** choose the next oven batch: e.g., 1 tray croissants, or 1 tray cookies, or skip (leave oven idle s slot), etc.

**Transition:** baking finishes after one slot and adds fresh items, items on the counter age one step, some demand arrives stochastically and removes items from the display. If demand exceeds what’s available, you miss sales.

**Reward(r):** revenue from sold items minus penalties for staleness/ waste (throwaways) and minus an “idle oven” penalty (lost opportunity).Fresh sales score higher than old stock.

**What a good policy learns to do:**

* Bake croissants a bit before the usual morning rush so they’re fresh at 8:30-9:00.
* If the counter already has plenty of cookies, skip cookies now and bake rolls for the lunchtime sandwich crowd.
* On unexpectedly slow days, avoid overbaking to prevent waste, on busy days, keep the one cycling with the fastest sellers.

**Why MDP suits this:**

It balances now vs. later, freshness and availably today vs. waste if demand doesn’t show up. The uncertainty in arrivals fits naturally, and the reward can reflect the bakery’s real priorities (freshness, minimal waste, steady sales). IT’s simple to explain to a non-technical owner and easy to pilot using the bakery’s own sales logs.

**8. Feedback**

I liked this assignment because it was hands-on and made the MDP ideas feel real. Setting up Gymnasium and running long experiments taught me how much randomness and sample size matter. Tiny code changes or different seeds can move the averages a lot. Comparing fixed random policies to Value Iteration made the planning benefit very clear. Seeing the V(s) heatmap and the policy arrows was especially helpful. Generating the required artifacts pushed me to be careful about reproducibility and record-keeping. The biggest challenge was the runtime and waiting for large batches to finish, but the parallel runs helped. Overall, it was a solid mix of coding, stats, and interpretation, and I came away with a better intuition for slip-robust policies and why model-based methods win here.

**9. Conclusion**

This assignment helped me connect the theory of MDPs to concrete results. On the slippery 8x8 FrozenLake, I saw how planning with a model (Value Iteration) clearly beats fixed random rules. MY best random policy (seed 13) averaged ~253 goals per 10, 000 episodes(~2.53%), while the optimal policy from Value Iteration averaged ~5088 per 10,000 (~50.9%). That ~20x gap made the value of solving the Bellman equations very tangible.

I also learned how stochastic transitions shape good behavior. The best policies don’t just head straight for the goal, they trace slip-tolerant corridors where sideways moves are still safe. The V(s) heatmap and policy grid made this visible: high values and arrows concentrate along paths that keep perpendicular slips away from holes.

From the experimentation side, I saw the importance of large sample sizes, variance, and reproducibility. Running 100x10,000. Episodes produced stable estimates and showed why we should report mean, std, and CIs. I also noticed small discrepancies between different runs (due to different RNG streams), which reinforced why saving CSVs and figures for the exact runs I report is important.

Finally, I came away with a clearer sense of trade-offs. Manual policy tweaks can help locally, but they don’t scale on stochastic, long-horizon problems. Model-based methods (like Value Iteration) give a principled, global solution and the result’s here back that up.