**ECE 6882 – Reinforcement Learning**

**Spring 2023 – The George Washington University**

**Osama Yousuf –** [**osamayousuf@gwu.edu**](mailto:osamayousuf@gwu.edu)

**Project 3 – Solving a Maze using SARSA and Q-learning Algorithms**

1. **Introduction**

In this project, I have implemented two different algorithms, namely SARSA and Q-learning, to solve the provided maze problem. I have carried out experiments to establish a sense of which algorithm performs better. Similar to Project 2, I used environment 4 and edited its functionality to my liking. Supplemental code files are provided as an accompanying archive. Table 1 highlights the purpose of each file.

Table : Description of implemented code.

|  |  |
| --- | --- |
| **File** | **Description** |
| main.py | Entry point. Executing this as is will replicate all plots used in this report. |
| policies.py | Contains my implementations of the SARSA and QLearning policies. Policies from Project 2 are also included. |
| util.py | Contains environment-specific functionality such as classes for the maze and agent, as well as visualization functions. |
| mazes/base.txt | This is the base file for the 18 x 18 maze which we have to solve for this project. This is directly used in main.py. This could be replaced with any other maze directly, and the environment should correctly load and solve it using both policy iteration as well value iteration. |

1. **Environment Description**

As highlighted in Table 1, my environment can load a maze for solving using generic text files, as long as the mapping is respected. Figure 1 shows the base maze, which contains a total of 18 x 18 possible states.

A picture containing square

Description automatically generated

Figure 1: Base maze environment loaded from mazes/base.txt

* 1. *State Space (S):* The state space contains all possible cells of the 18 x 18 matrix with the exception of walls where the agent can be present at any given point. Each state can either be empty, full (walled off), bump, oil, or goal. Accounting for walls, the total number of states are .
  2. *Action Space (A):* An agent can take one of four possible actions at any given state. These are: Up, Down, Left, and Right. In my environment, these are coded as action 0, 1, 2, and 3 respectively.
  3. *Transition probabilities (T):* When an action is chosen, the agent can either move to one of its neighbor states or stay in the same sell depending on – the transition randomness parameter – and whether the anticipated state is a wall or not. In exact terms, after taking any given action, the agent moves to the anticipated state with a probability of and moves to one of the other neighboring cells with a probability of . If the state to transition to is a wall, then the agent simply stays in the current cell. can take on values between 0 and 1 (inclusive), where means that the agent will always transition to its intended state and means that the agent will always transition to an unintended neighboring state with equal probability of .
  4. *Reward Function:* The agent receives a -1 reward for all actions regardless of the state it transitions to. It also receives additional special rewards if it transitions to oil, bump, and goal states, as specified in Table 2.

|  |  |
| --- | --- |
| **State (after transition)** | **Reward** |
| All actions (Up, Down, Left, Right) | -1 |
| Oil | -5 |
| Bump | -10 |
| Goal | +200 |

Table : Reward as a function of transitioned state

1. **Algorithmic Comparison: SARA vs. Q-learning**

SARSA (State-Action-Reward-State-Action) and Q-learning are both reinforcement learning algorithms, but they both have subtleties and differences that differentiate their performance from one another.

* **SARSA** is an **on-policy** learning algorithm, meaning that the agent learns from experiences generated by following the current policy. The same policy is used to generate actions as well as update the policy. **Q-learning** on the other hand is an **off-policy** algorithm, where the agent learns from the experiences generated by following a different policy (greedy) than the one currently being used. Different policies are used to generate actions and update the policy.
* SARSA learns the Q-value based on the next state and action taken by the agent, whereas Q-learning simply learns Q-value based on the action that gets the maximum possible reward from the next state. This is evident in the update step used in these two algorithms. For SARSA, the update is . For Q-learning, the update is . The different hyperparameters in these equations are summarized in Section 4.
* It is known that SARSA, owing to its on-policy nature, can converge to a suboptimal policy in a few cases. On the other hand, Q-learning is guaranteed to converge to an optimal policy.

Compared to policy iteration and value iteration algorithms (which were model-based), both SARSA and Q-learning are model-free algorithms. This means that they do not require explicit knowledge of the underlying transition probabilities for learning the optimal policy.

1. **Algorithmic Hyperparameters**

Table 3 summarizes the exact values used for the different algorithmic hyperparameters in my simulations, followed by an explanation of what each hyperparameter corresponds to.

|  |  |
| --- | --- |
| **Hyperparameter** | **Value** |
|  | 0.02 |
|  | 0.95 |
|  | 0.3 |
|  | 0.1 |
| Episodes (E) | 1000 |
| Episode Steps (ES) | 1000 |
| Policy Steps (PS) | 500 |

Table 3: Summary of chosen algorithmic hyperparameters

* is the transition randomness of the agent (as explained in Section 2.3). The agent transitions to an intended state with a probability of , and a random neighbor with equal probability . Higher the value of (between 0 and 1, inclusive), higher is the randomness in the agent in following an intended trajectory.
* is the discount factor allotted to future rewards. It directly represents the importance of future rewards relative to immediate rewards. The higher the value (between 0 and 1, inclusive), the more important future rewards are considered relative to the immediate reward.
* is the learning rate that is multiplied by the error term in the Q-value update equation for each algorithm (given in Section 3 point 2).
* is the parameter for the -greedy policy that is derived from the Q-values. For SARSA, -greedy is used when the agent has to choose the initial as well as next state actions. For Q-learning, it is used for the initial action. -greedy means that a greedy action (one which maximizes the Q-value) is chosen with a probability of ; otherwise, an action is drawn randomly with equal probability over all possible actions.
* E refers to the total number of episodes for the SARSA and Q-learning algorithms.
* ES refers to the total number of steps an agent is allowed to take within each episode. Once ES steps have been taken, an episode is concluded.
* PS refers to the total number of steps an agent can take to reach the goal state. If an agent does not reach the goal state within PS steps, the policy is classified to have failed in finding a path from the start to a goal state. If an agent reaches the goal state within PS steps, the policy is classified to have successfully found a path from the start to a goal state.

Apart from these hyperparameters, it is also important to note how tie-cases are handled in my implementation. A tie can occur when more than one action leads to the optimal Q-value. In such a case, competing actions are chosen randomly with equal probability during the learning phase. After the Q-values have been learned, the optimal policy is derived directly by choosing the action with the highest Q-value for each state. If there’s a tie in this phase – it is broken simply by choosing the first action with the highest Q-value (in the sequence Up, Down, Left, Right).

**SARSA**

**Base Scenario**

Figures 2, 3, 4 report the optimal policy, V function values, and the optimal path respectively learned by policy iteration under this scenario. In addition, Figure 5 shows an instance of an agent attempting to traverse the maze using the policy with transition randomness in place.

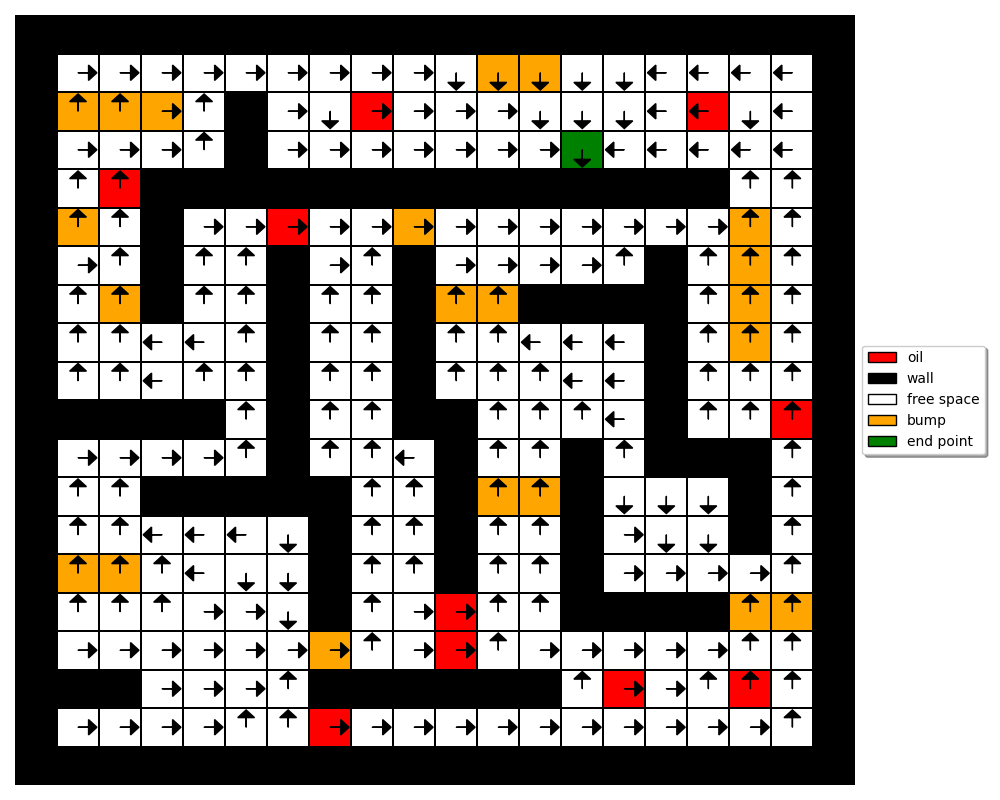


Figure : Optimal policy learned by Policy Iteration under the base scenario

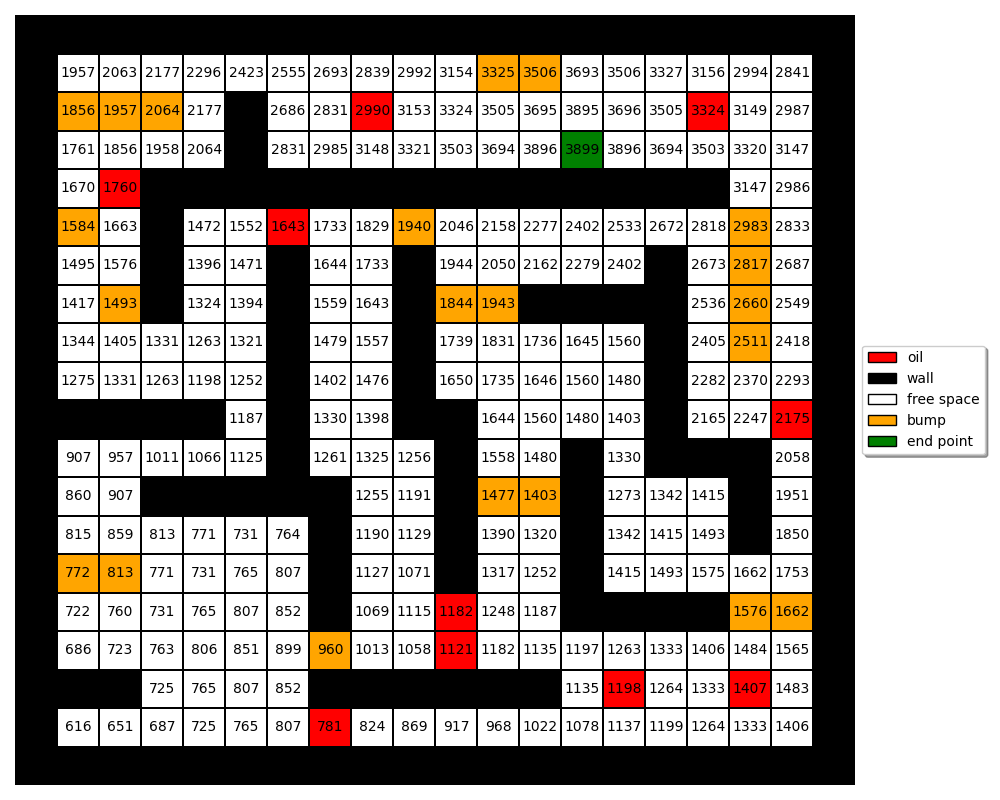


Figure : Optimal V function learned by Policy Iteration under the base scenario

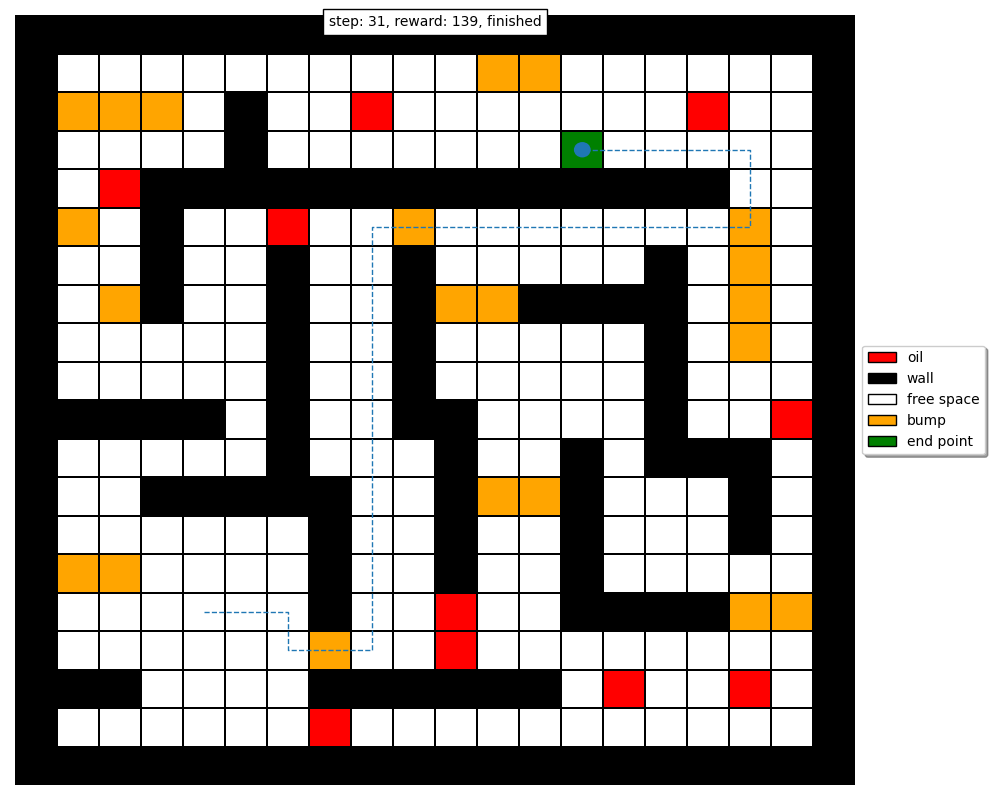


Figure : Optimal path learned by Policy Iteration under the base scenario

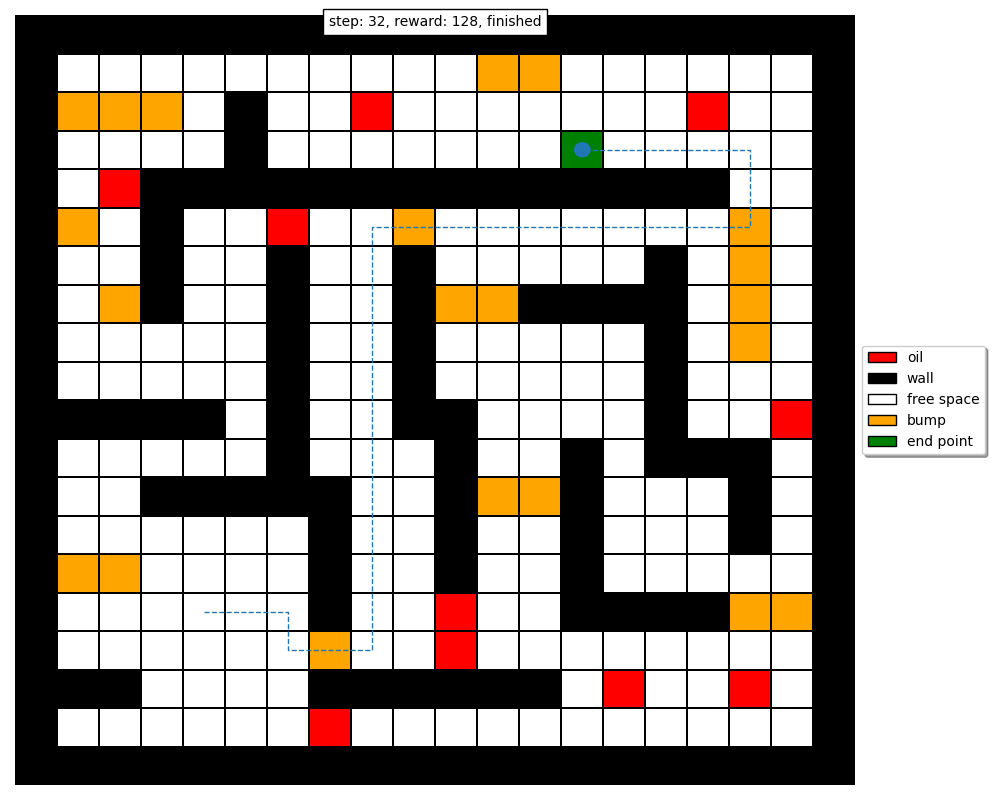


Figure : Instance of an agent solving the maze under the given scenario and learned policy

It can be seen that the trajectories in Figures 4 and 5 are similar, with only a single extra step as the agent is quickly able to recover under the learned policy. This occurs because we have really small transition randomness (p = 0.02), so the agent transitions to an intended state 98% of the time.

The large V function values in Figure 3 can be explained similarly – once the agent reaches the goal state, it can take the optimal action of down, hit a wall, remain in the goal state, and repeatedly receive the goal state reward of +200. The expected reward of each state thus accumulates overtime and increases in magnitude, as seen in the V function values.

**High Stochasticity Scenario**

Figures 6, 7, 8 report the optimal policy, V function values, and the optimal path respectively learned by policy iteration under this scenario. In addition, Figure 9 shows an instance of an agent attempting to traverse the maze using the policy with transition randomness in place.

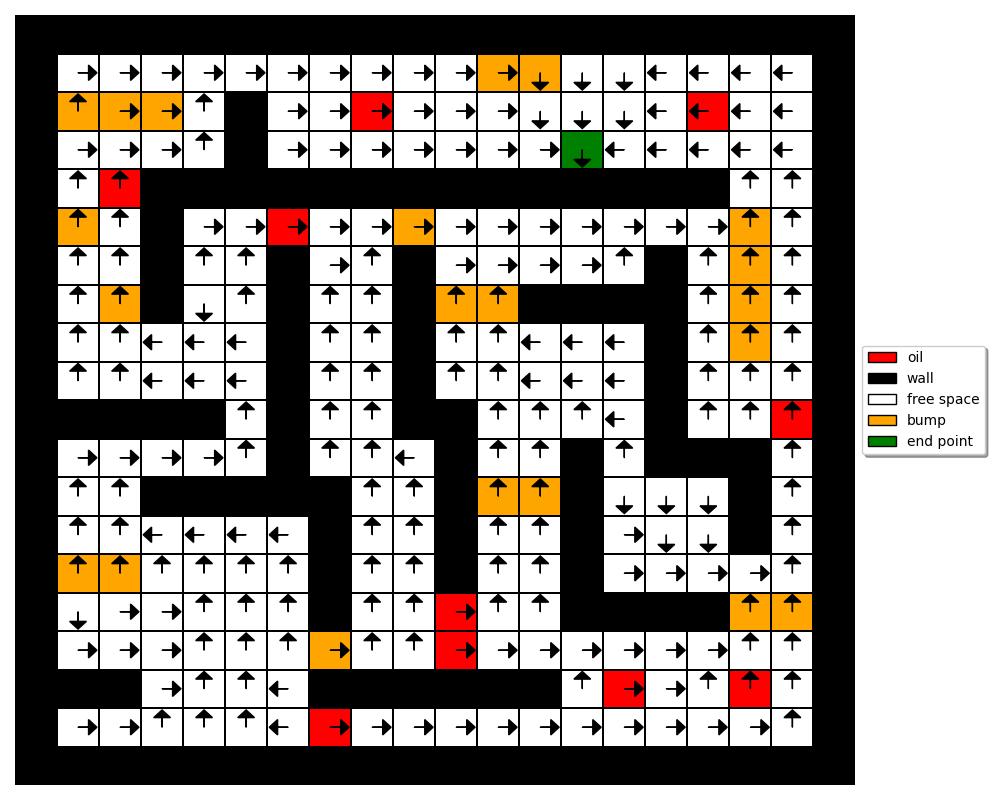


Figure 6: Optimal policy learned by Policy Iteration under the base scenario

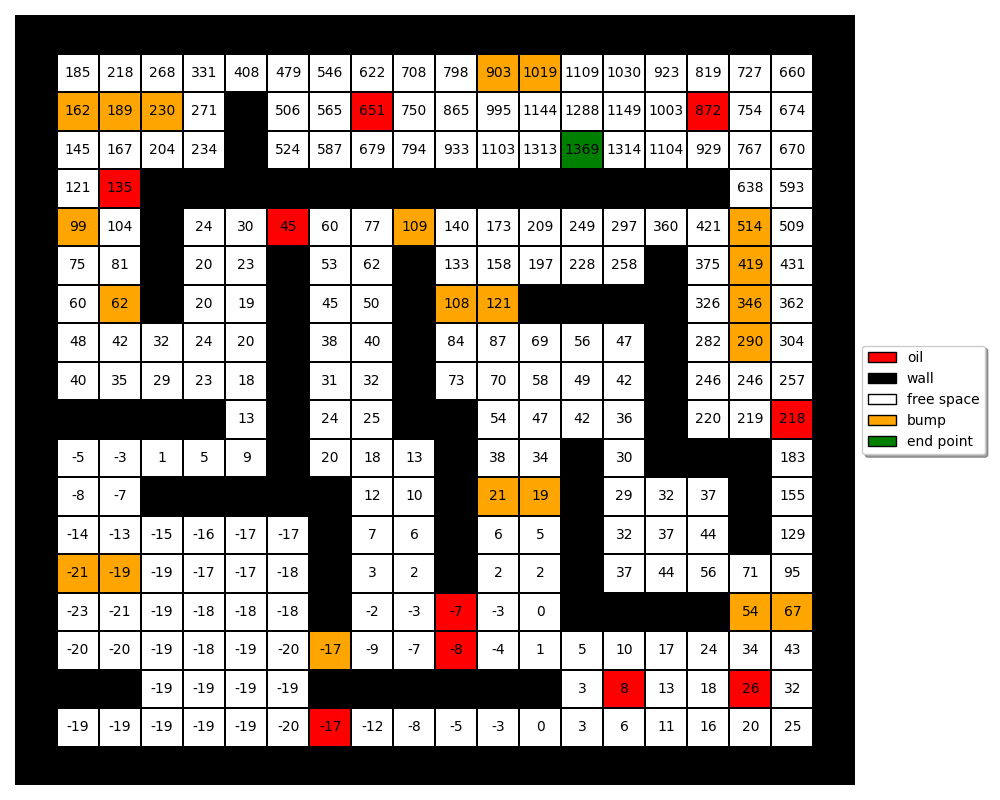


Figure 7: Optimal V function learned by Policy Iteration under the base scenario

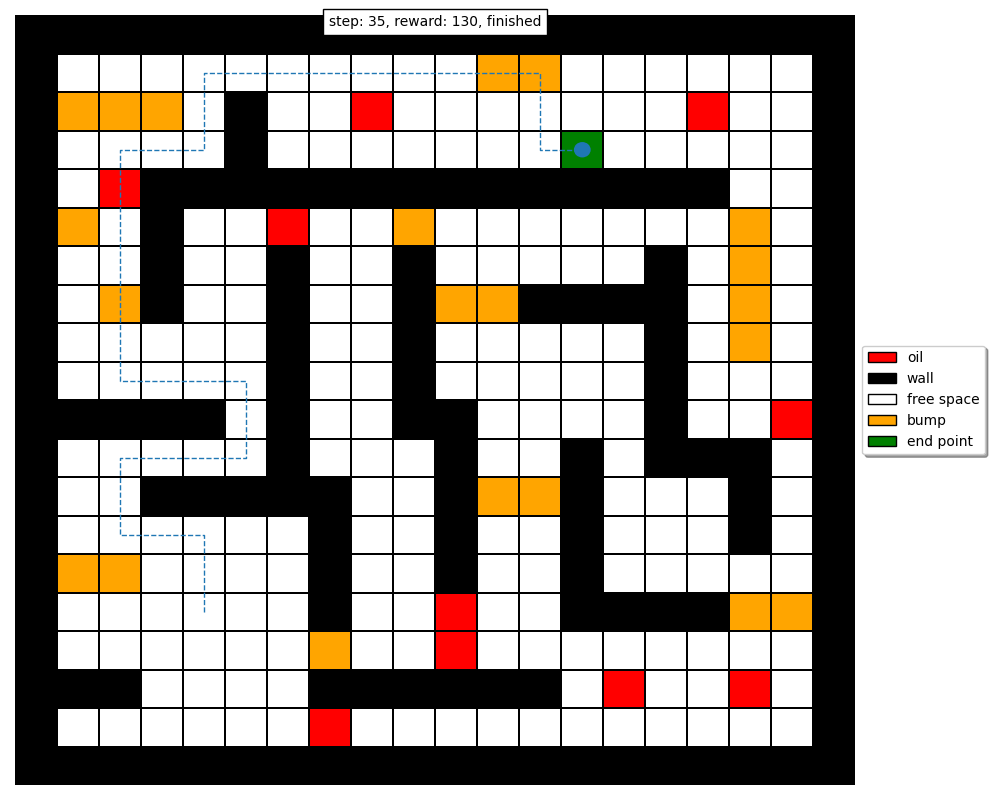


Figure 8: Optimal path learned by Policy Iteration under the base scenario

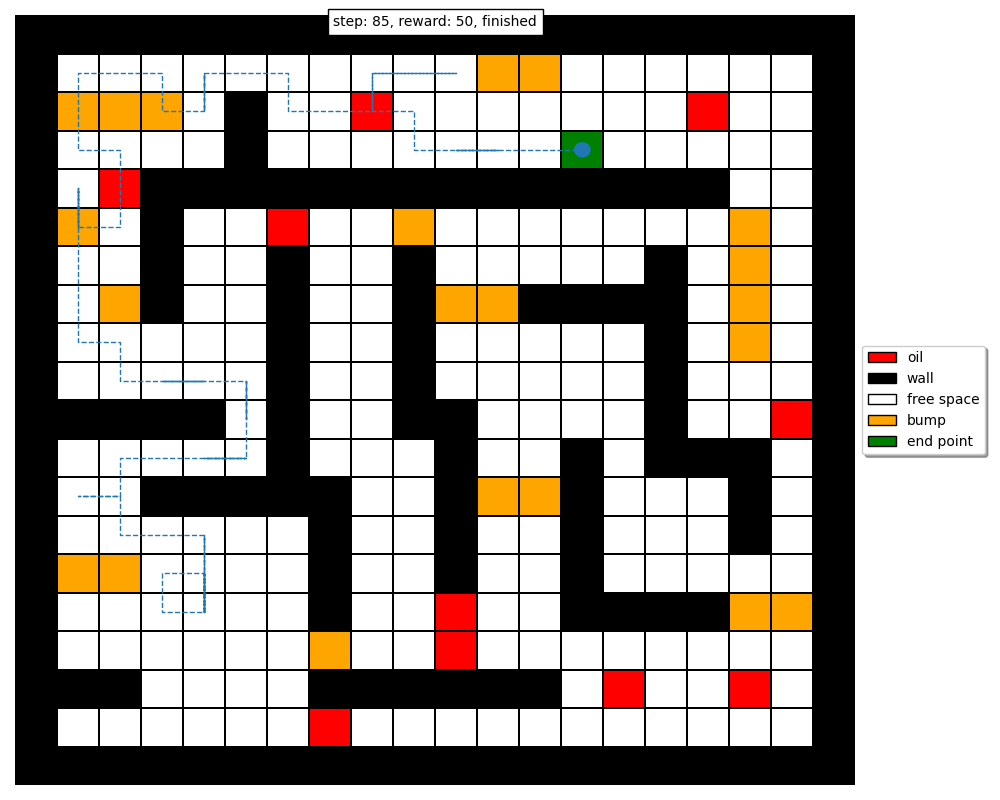


Figure 9: Instance of an agent solving the maze under the given scenario and learned policy

It can be seen that the trajectories in Figures 8 and 9 have a high variation, indicating that the transition randomness is too high for the agent to be able to quickly recover under the learned policy. This occurs because we have a relatively high transition randomness (p = 0.5), so the agent transitions to an intended state only 50% of the time and wanders randomly in an unintended direction the remaining 50% of the time.

Compared to the V function values in the base scenario, the V function values in Figure 7 are much smaller. This can be explained along similar lines as the base scenario – once the agent reaches the goal state, it can take the optimal action with only a probability of 50%, and so there’s always a high chance of transitioning to a state that is not the goal state, thereby reducing the expected accumulated reward.

Finally, it can be seen that under this scenario, the optimal path that is learned is sub-optimal compared to the path which is learned in the base scenario. This can be attributed to the fact that the transition randomness led to a much higher degree of exploration, and the agent was unable to arrive at the same optimal policy. It is possible that other algorithmic parameters such as the discount factor and accuracy of estimation could be altered to learn a much better policy overall.

**Small Discount Factor Scenario**

Figures 10, 11, 12 report the optimal policy, V function values, and the optimal path respectively learned by policy iteration under this scenario. In addition, Figure 13 shows an instance of an agent attempting to traverse the maze using the policy with transition randomness in place.

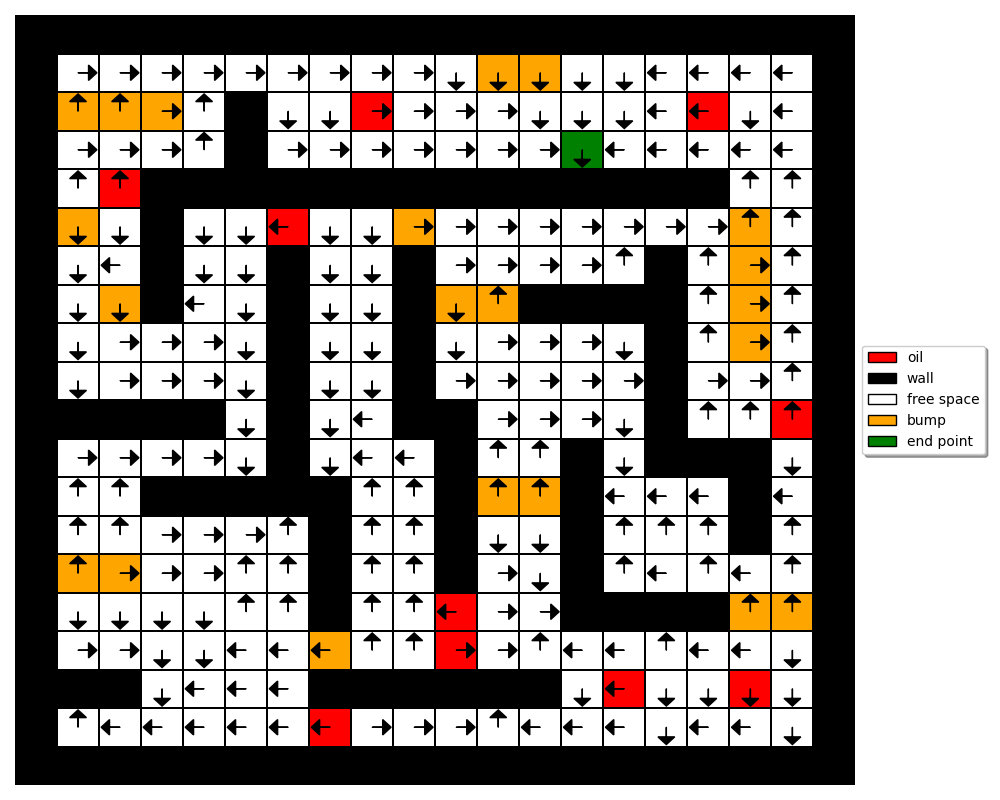


Figure 10: Optimal policy learned by Policy Iteration under the base scenario

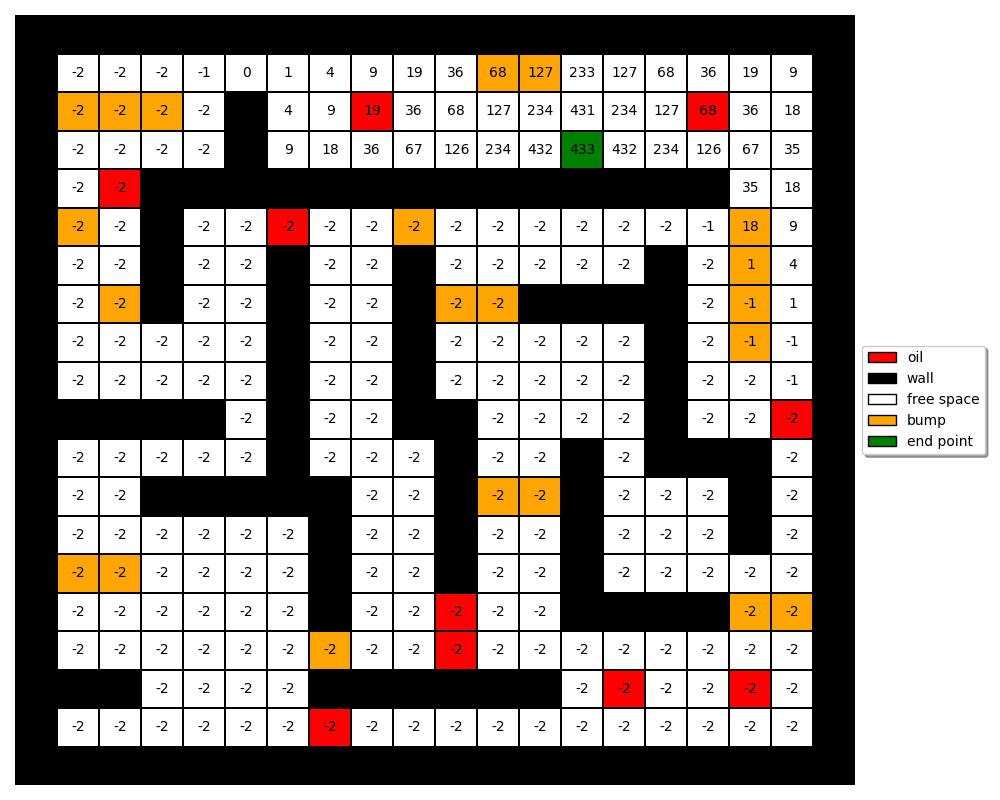


Figure 11: Optimal V function learned by Policy Iteration under the base scenario

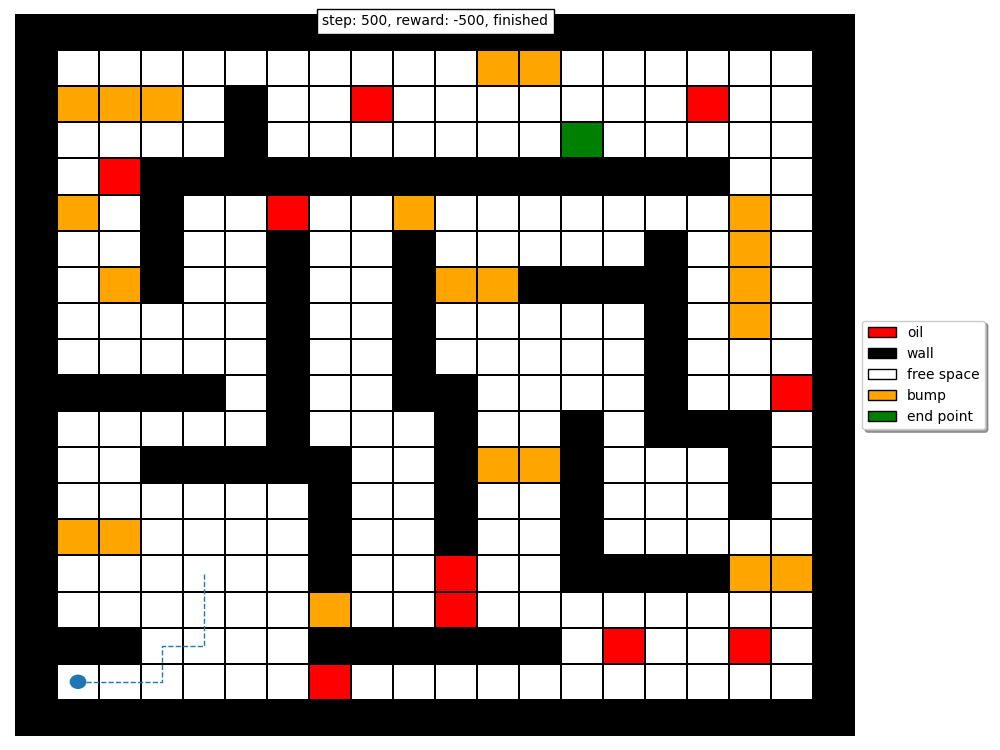


Figure 12: Optimal path learned by Policy Iteration under the base scenario

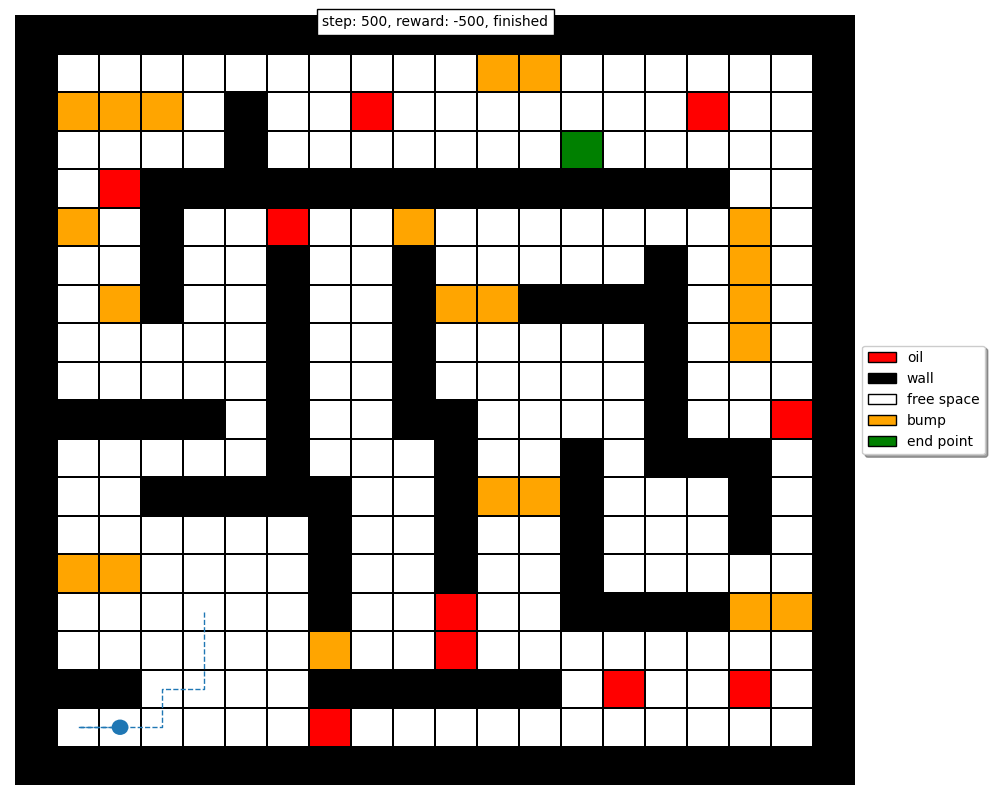


Figure 13: Instance of an agent solving the maze under the given scenario and learned policy

It can be seen that the trajectories in Figures 12 and 13 have only a slight variation, but both are equally bad since the agent is unable to get to the goal state from the starting position. Since we have small transition randomness (2%), the agent remains trapped in the corner as it follows the bad policy shown in Figure 10.

Compared to the V function values in the previous two scenarios, the V function values in Figure 11 are the smallest. In addition, the optimal path that is learned is entirely sub-optimal since the agent remains trapped under the learned policy. Both of these things can be attributed to the smaller discount factor. A higher discount factor encourages the agent to optimize for long-term rewards, while a lower discount factor focuses on immediate rewards. From the V values, it can be seen that the agent has hyper-optimized for immediate rewards, leading to a policy that basically traps the agent if the starting position is sufficiently far from the goal states. From our results in the base scenario, we already know that by increasing the discount factor from 0.55 to 0.95, we can learn a really good optimal policy.

Table 3 offers a comparison of the performance of policy iteration under the three investigated scenarios. It can be seen that the base scenario is the optimal choice since the agent is able to achieve an optimal policy that guarantees the highest reward while minimizing the number of steps taken to get to the goal state. The small discount factor scenario exhibits the worst performance, since the policy fails to solve the maze altogether under the given starting position.

Table : Performance comparison of Policy Iteration under the three investigated scenarios

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Steps** | **Final Reward** |
| **Base** | 31 | 139 |
| **High Stochasticity** | 35 | 130 |
| **Small Discount Factor** | 500 | -500  (agent stuck, maze unsolved) |

**Comparison of Policy Iteration & Value Iteration**

Through my experimental results, I found that both policy iteration and value iteration converge to the same optimal policy under the same algorithmic parameters. For a given scenario, the optimal policy learned by the policy iteration algorithm was the same as the optimal policy learned by the value iteration algorithm. This being said, I did find differences between the two in terms of the number of iterations it took each algorithm to converge under a given scenario, as well as the total computation time. I found that policy iteration required more computation time per iteration but often converged faster than value iteration, while value iteration was slightly simpler to implement than policy iteration and converged slower than policy iteration in most scenarios, as summarized in Table 5.

Table : Comparison of policy iteration and value iteration in terms of average computation time

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Policy Iteration** | **Value Iteration** |
| **Time to learn optimal policy**  **(Averaged over 20 independent runs)** | |
| **Base** | 3.94 s | 4.35 s |
| **High Stochasticity** | 2.56 s | 3.04 s |
| **Small Discount Factor** | 0.609 s | 0.433 s |