CS4641: Machine Learning
Spring 2019
PS4 Markov Decision Processes

Joel Ye

April 14, 2019

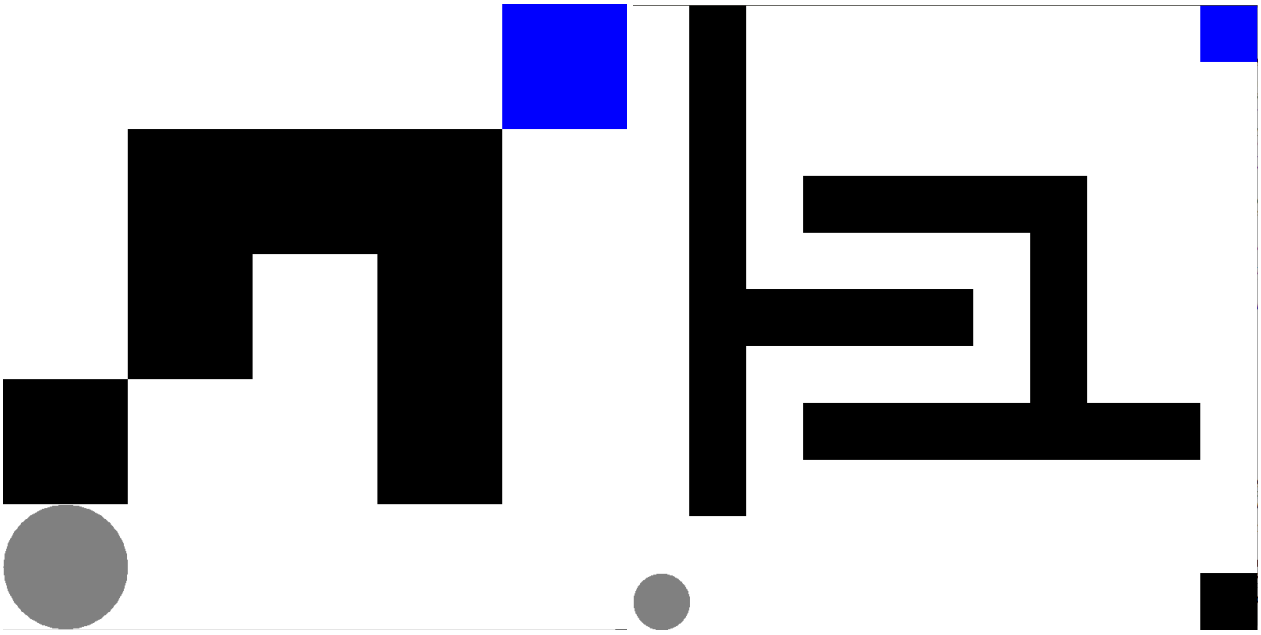
1 Introduction

Markov Decision Processes are a class of problems entirely different from those explored in previous assignments. Generically, we pose agents in an environment that provides rewards for the agents. Agents must choose actions at every timestep or state in order to maximize reward, and their choice of actions per state is known as a policy. Policies can be developed in a variety of manners. Two model-based methods considered in this assignment are Value Iteration (VI) and Policy Iteration (PI), and the model-free method explored is Q-Learning (QL).

The environments posed are a small and large grid world, common but constrained examples typically used for learning experiments (no pun :)). Grid worlds are 2D Cartesian grids, where an agent is initialized at some coordinate. At each timestep the agent steps in the grid to adjacent blocks and can receive some corresponding state-based reward. Most states in grid-world provide nothing except a living reward, save terminal states, which end the episode and provide a terminal reward.

Concretely, this assignment uses Juan Jose's starter code[1], which itself uses the Java Burlap library. Both grid-worlds in consideration initialize the agent at the bottom left corner of a square grid, and have a terminal state. The small grid-world is a 5x5 grid (1a), while the big grid-world is an 11x11 grid (1b). As shown in the diagrams, a certain number of states are illegal, indicated in gray (meaning corresponding attempted transitions will fail and cause the agent to remain the same state). Given the illegal configurations, our small environment specifies a total of 18 states, while our large grid world specifies 92 states.

One big utility of grid worlds is their great extensibility. Transitions are stochastic (actions may be realized differently than intended), so custom transition functions can be defined. In a degenerate case, we can specify a generic MDP, where each state can connect to each other state with non-zero probability, but grid-worlds generally assume states cannot transition to non-adjacent states. In this sense, gridworlds are useful because they can be compactly represented by sparse reward matrices, and provide a convenient model that VI and PI can leverage. In this assignment, transition probabilities are statically defined as .8 towards the intended cardinal direction, and the remaining .2 split evenly among the other directions. As mentioned earlier, if there are impossible transitions, such as moving off the grid, this will result in staying in the same state. Additionally, we can provide custom reward functions for individual states, even if they aren't terminal states, such that an agent may reason that it's beneficial to keep revisiting a planted fountain of reward, or avoid a row of negative rewarding states and the associated region around it. Finally, there are various other more high-level parameters, such as iterations for the respective algorithms, as well as living



(a) Small Grid (b) Big Grid

reward and terminal rewards/states. Here, living reward is set to -1, while the blue terminal state provides a reward of 100. There is no penalty for attempting an illegal action (though this can also be designed). For the sake of simplifying experiments and isolating variables, only a few of these are considered, dedicating the main portion of the paper to analyzing the individual algorithms.

We can make some observations about the design of these grid worlds. Given the initial state in the small world, it is actually impossible for the agent to reach the top left corner states, since the terminal state is a choke point. It is interesting to see what can be deduced about impossible states. In the large world, it is pointless for the agent to go along the left, but not pointless for the agent to travel along the winding mazelike path.

2 Value Iteration and Policy Iteration

As a model based method, VI allows learning about the environment without any simulation. In an infinite setting the value iteration algorithm assigns a value to each state equal to its maximal expected future reward. However, agents don't live forever, and to account for non-convergent sums a finite horizon or discount factor is usually introduced. In this experiment, finite horizons were employed, by capping the number of iterations each algorithm could run (thus information from one state only propagates at most a set number of transitions away), defaulting to 100.

Suppose we had some policy π that was simply to move to the state with optimal value. Formally, state values are uniformly initialized at 0, and for each 'iteration' in VI uses one Bellman update at time t to create new values per state s, as follows:

$$\begin{aligned} v_{\pi'}(s) &= \max_a \mathbb{E}\Big[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) \mid S_t = s, A_t = a\Big] \\ &= \max_a \sum_{s'} p(s' \mid s, a)\Big[r(s, a, s') + \gamma v_\pi(s')\Big] \end{aligned}$$

$\gamma = 1$ since we don't use discount factors here. We use π here for notational convenience instead

of a max operator, but we can repeat this update until either we reach our iteration cap or our maximum state value delta is beneath a small threshold.

Policy Iteration is also heavily based on the Bellman update. A policy π is initialized randomly. In every step, we still update values per state, but instead of using the best possible action, we use the action and update according to the transition dictated by the policy. We alternate between value updates and policy updates, where we course correct our policy π to $\pi*$, which evaluates the best action per state based on current values. (The exact alternation between value updates and policy updates is tunable, in fact faster convergence may be seen by allowing a few steps of value iteration per policy update).

Formally, policy iteration is described as:

$$v_\pi^{t+1}(s) = \sum_{s'} p(s' \mid s, \pi(s)) \Big[r(s, \pi(s'), s') + \gamma v_\pi^t(s') \Big]$$

$$\pi^{t+1} = \max_{a \in A} \sum_{s'} p(s'|s, a)(r(s', a, s) + \gamma v_\pi^t(s'))$$

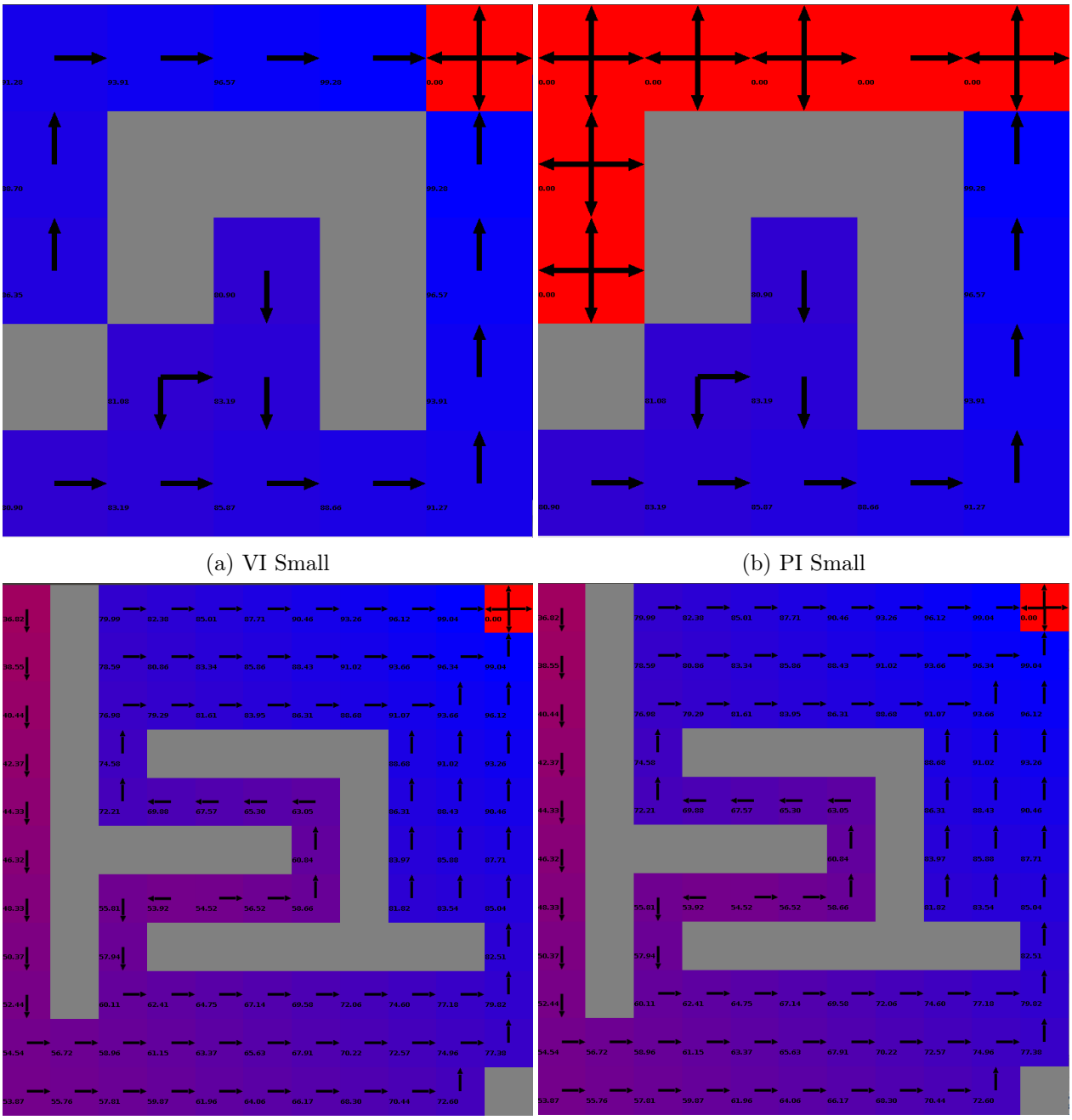
Consider the results of converged value and policy iterations in 2. We have a curious result here - both of these algorithms operate on a model basis, regardless of which states an agent can feasibly explore and not explore. For example, in the big world, the resulting policies are identical, moreover, each states' values are also the same. This is a result expected from class, namely, VI and PI both converge to the same exact policy and values, because they converge to the optimal policies. They aren't used in practice because they are only applicable to fairly constrained environments.

Some interesting things to note in the big world is that the actions all tend to make sense, and it's also interesting that squares with the same manhattan distance away from a particular goal point (termination state, or choke point on the right), tend to prefer items aligned on the positively sloped diagonal. This is an artifact of the stochasticity, as given noise in movements, we prefer squares that allow missteps and still result in progress towards the goal. Additionally, in the tunnel portion, the agent decides to "commit" to the maze only a few steps in (rather than turning at the point in the maze that has the same manhattan distance to the goal either way), as it's easier to try to escape and make forward progress, rather than try to back out. Note that in this tunnel, actions have relatively high failure rate, staying unproductively in the tunnel with 20% chance.

Meanwhile in the small world, we see something even more peculiar. Value iteration behaves as expected, dictating the optimal steps to take in each state. Note a square that has symmetric actions, thus indicating two arrows because each has identical expected reward. However, in policy iteration, the unreachable portion of the map fails to provide informative reward about the top portion of the map. Most of the squares say each direction is identical mediocre, since expected returned reward is 0. This is likely a bug in Burlap, since no reward is propagated to the square to the left of the terminal state (Identical initializers, and reachable state visualizers are used in VI and PI in Emeterio's code). Therefore, its value stays at 0, as opposed to updating. If π_0 didn't indicate that square to go to the left, there is non-zero expected return from wandering to the terminal state. The visual also arbitrarily indicates that said square should go to the right, rather than the any direction.

It is important to note that policy iteration and value iteration should result in identical results, as in the large grid world.

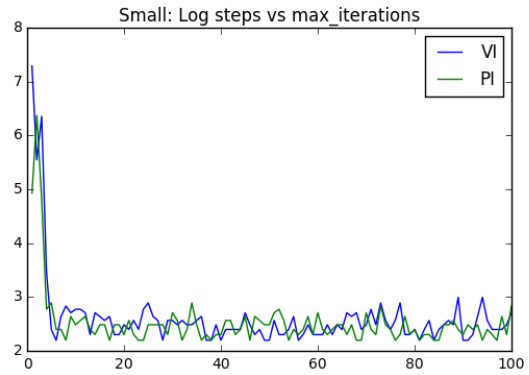
Let's compare iterations and runtimes in these algorithms. Graphs in 3 are shown in log scale. To clarify terminology, iterations here are done after the fact - for actual convergence calculations, we refer to the time graph to compare the performance of these algorithms (all algorithms ran the full specified number of iterations, in order to compare their final states consistently).



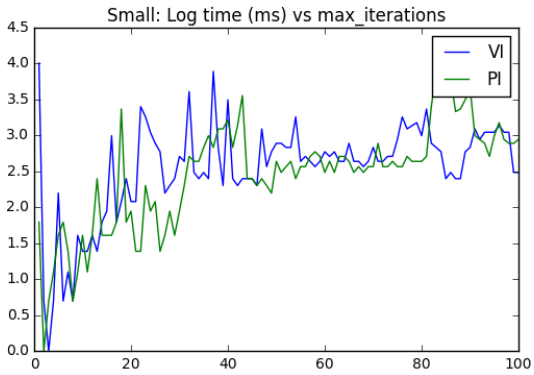
(a) VI Small (b) PI Small

(c) VI Big (d) PI Big

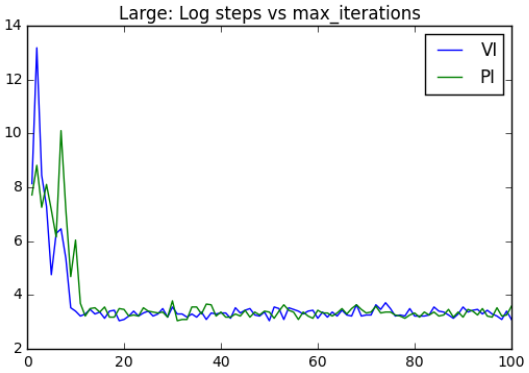
Figure 2: Value and Policy Iteration, Converged Results



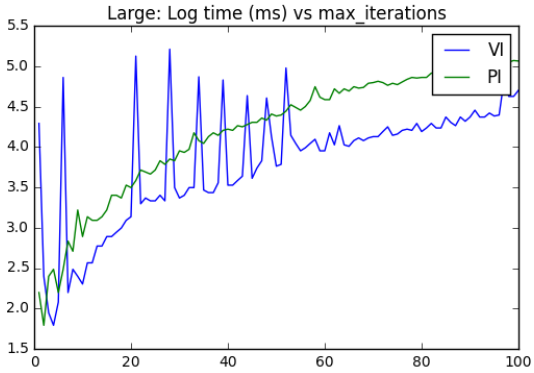
(a) Steps: Small



(b) Time: Small



(c) Steps: Big



(d) Time: Big

Figure 3: Value and Policy Iteration, Converged Results

As seen in the time graphs, in the small world, time differences in algorithms are negligible. However, in the large world, value iteration is a consistent factor faster than policy iteration. The max operator is not intractable due to transition sparsity, and it is worth the slight overhead in order to consistently improve policy in the same step as value update.

The iteration graphs depict the number of steps it takes for an agent to reach termination following the final policy, thus the point of flatline is the number of iterations required to reach convergence. Note that in the small world, difference between in VI and PI is negligible, but as the world gets bigger, value iterations converges a few iterations quicker. As mentioned earlier, they both converge to the optimal policy (around $e^3 = 20$ odd steps to reach terminal state in the large world).

3 Q Learning

The last method studied is Q-Learning, which is a model-free reinforcement learning algorithm, in that it doesn't build a transition model nor values for states. Instead, it has an internal map of Q-values, which evaluate state-action pairs, $Q(s, a)$. Agents explore in episodes, and update their Q-values based on feedback from the environment. During exploration, agents balance exploration (non-optimal) action to discover the environment and exploitation (optimal) action to gather more

information about best paths. During the deployment of a Q-Learning agent it's typical to use entirely exploitative strategies. In Emeterio's implementation, the exploration factor is $\varepsilon = 0.1$, a static exploration policy. We also have a learning rate, which represents how quickly reinforcement is incorporated into a Q-value.

Formally, the Q update for a given state transition $s - a - s'$, learning rate α is as follows:

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

The old Q-value essentially moves closer to the following states Q-value given the learning rate. We use learning rate of .99 to indicate to instability of learning in most experiments. Also note, we assume optimal action in our policy, we can also use observed following action, which results in SARSA. We see some Q-values in 4.

The first item we might notice is that values are a lot less stable. The values indicated in the states are the Q-values of the optimal actions, shown as the arrows. In the small state, our values are very intuitive and familiar, and here the top unreachable states all having no value makes sense, since our agent will never receive any reinforcement on those states. The more interesting patterns results in the large world. In the larger world, there are significantly more states to explore, and each episode explores a smaller and smaller fraction of the states. Due to the smaller greedy policy, edge states are barely explored. Additionally, states that are difficult to wander out of uninformedly (i.e.) the tunnel are heavily penalized, given the number of agents that wandered in and suffered many iterations of living reward before making it out. We notice this pattern is even stronger the more iterations we do – it does not seem like the agent has corrected its negative experience with an informed traversal yet.

However at around 50000 iterations, the values begin to stabilize, to where they seem like earlier, model based numbers. Even then, however, policies and Q-values are not extremely stable.

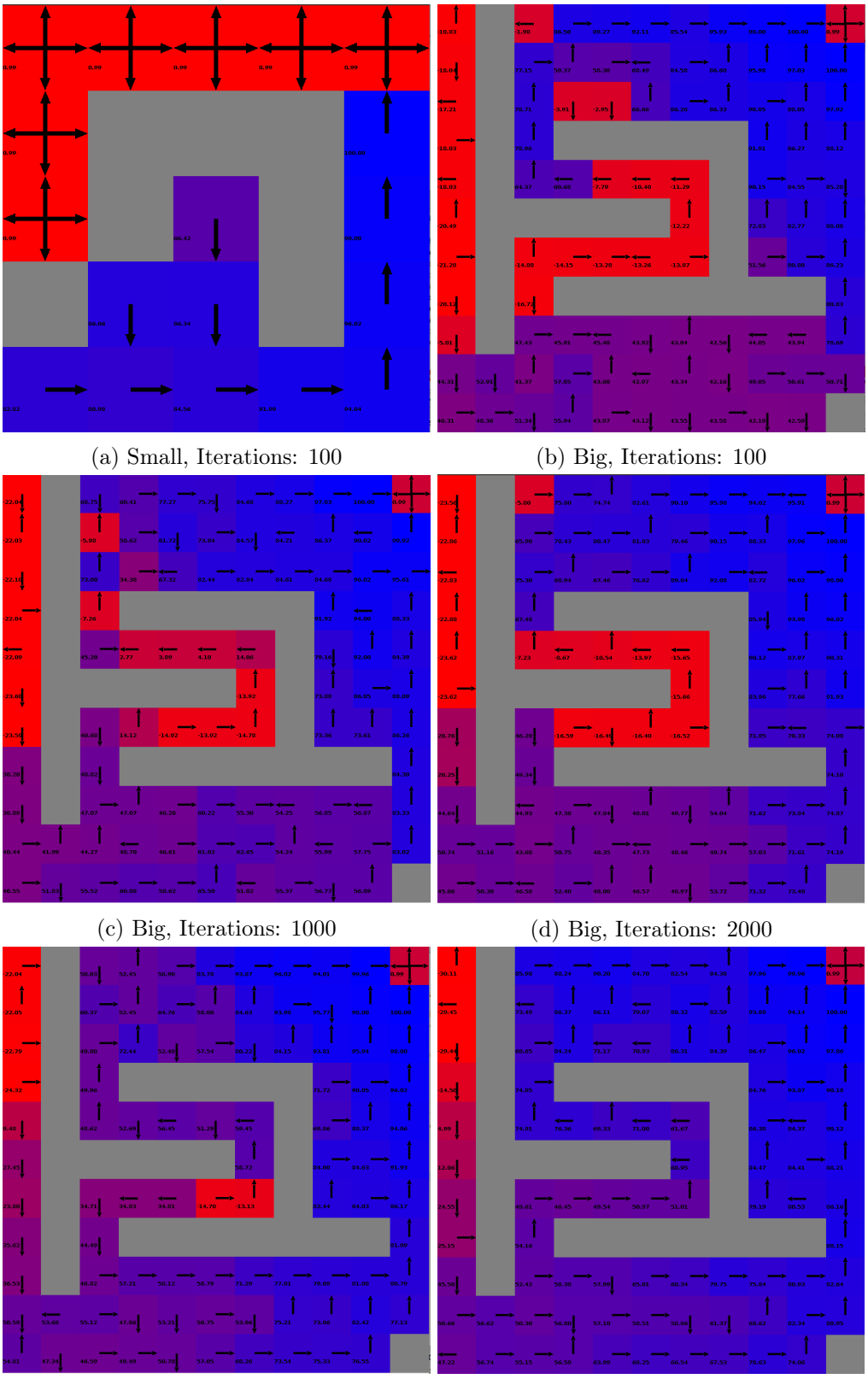
We can experiment with smaller learning values to see how critical it is to not take all your experiences are initial value (.1), as shown in 5. We note the more explored regions immediately stabilize, but curiously the tunnel does not get explored nearly as often for exactly that reason (agents don't bother exploring there since it's always more reasonable to head towards the chokepoint, and they learn this easy route much quicker).

4 Conclusion

Overall Q-Learning appears to learn much worse than model-based methods, but it tends to still be the best we can do in many real-world applications, where building state models is inapplicable. Value iteration tends to converge slightly faster than policy iteration, but both are very quick and optimal in grid world settings. Designed grid-worlds enable us to explore various different properties of these algorithms, and highlight that the difficulty and ultimate problem of exploration and exploitation, in learning about rare states.

References

[1] San Emeterio, Juan J. Open-sourced Burlap Runner. `https://github.com/juanjose49/omscs-cs7641-machine-learning-assignment-4`. 2017.



(a) Small, Iterations: 100

(b) Big, Iterations: 100

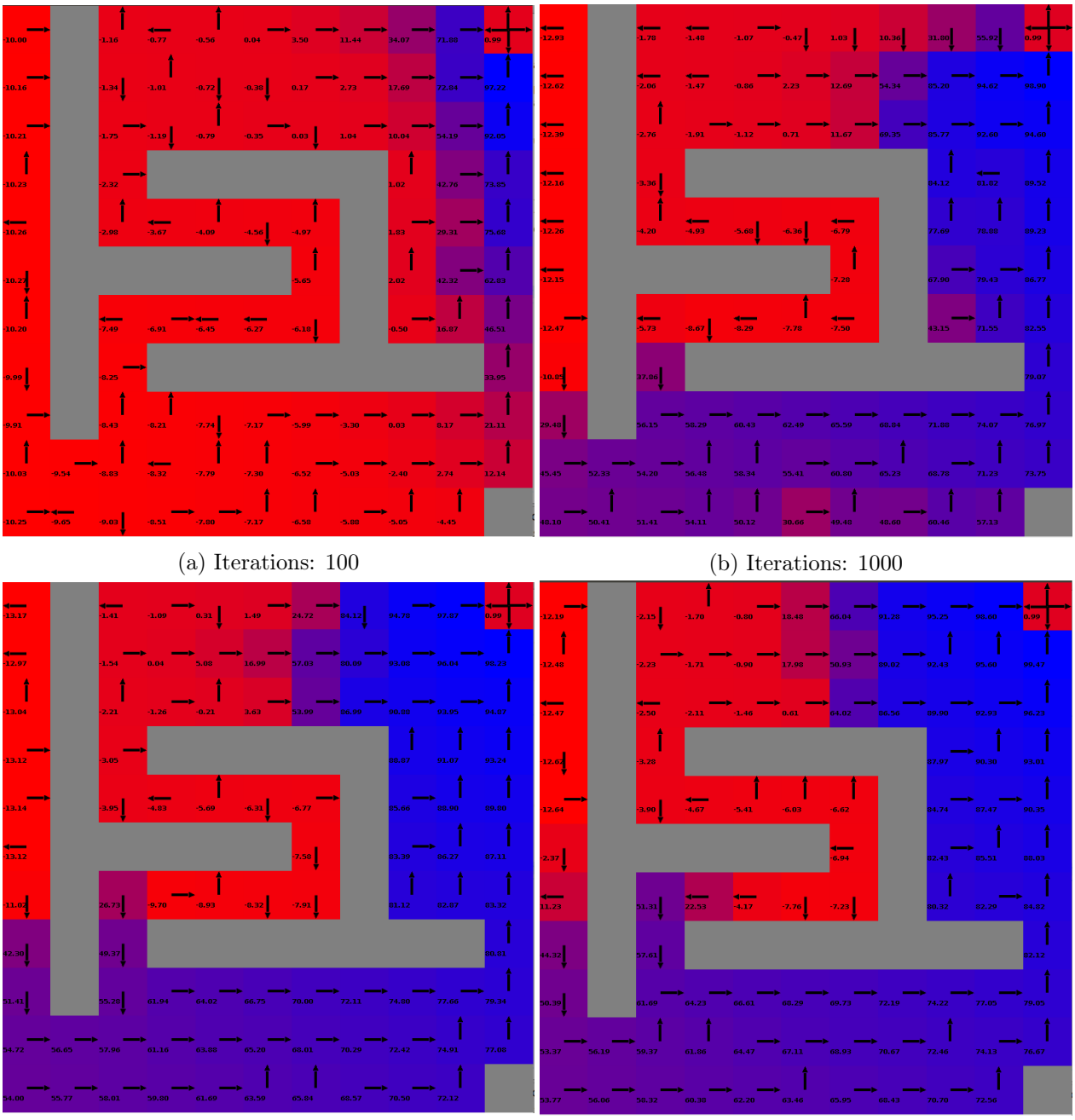
(c) Big, Iterations: 1000

(d) Big, Iterations: 2000

(e) Big, Iterations: 20000

(f) Big, Iterations: 50000

Figure 4: Q-Value Final Policies and Optimal Action Values



(a) Iterations: 100

(b) Iterations: 1000

(c) Iterations: 10000

(d) Iterations: 50000

Figure 5: Q-Values, Learning Rate 0.1