Markov Decision Processes - Project 4

A. Introduction to the MDPs

Problem 1: Pokémon!

As the semester comes to a close, I have more work and more assignments, but less and less time to do them. Naturally I chose this time to download a Nintendo DS emulator and run through the game that defined my 6th grade: Pokemon HeartGold. HeartGold (and every other pokemon game) has a surprising amount of depth; in particular, the formula for catching wild pokemon looks like it came from a physics thesis. I wanted to model it as an MDP, but I couldn't get my head around it, let alone implement it with BURLAP. If you aren't familiar with catching wild pokemon (the right way, i.e. *not* PokemonGO), then there are X important factors: the pokemon species, the health of the pokemon, the status effects on the pokemon, and the type of pokeball. In general, a more common pokemon, with low health, multiple status effects (paralyzed, asleep, etc.), and using a rare/expensive pokeball will be caught. The reverse (high health, rare pokemon, no status effects, common pokeball) will not be caught. However, it is a stochastic process, so there is always an element of unpredictability. This makes it perfect to model as an MDP. For my own sanity, I simplified the process to only consider health as a fraction of its maximum, and a boolean "has an active status effect". Our agent (the pokemon trainer) can choose to Attack (attempt to lower the wild pokemon's health), Status Attack (attempt to put a status effect on the wild pokemon), or Throw a Pokeball (attempt to capture the pokemon). Attacking will lower the health of the pokemon, making it easier to catch, but

has the chance to knock out the wild pokemon (since the amount of damage done is also random), ending the game and making it impossible to catch. Applying a status effect adds a static bonus to the catch rate. Throwing a pokeball attempts to capture the pokemon and end the game. My “simple” catch formula is below.

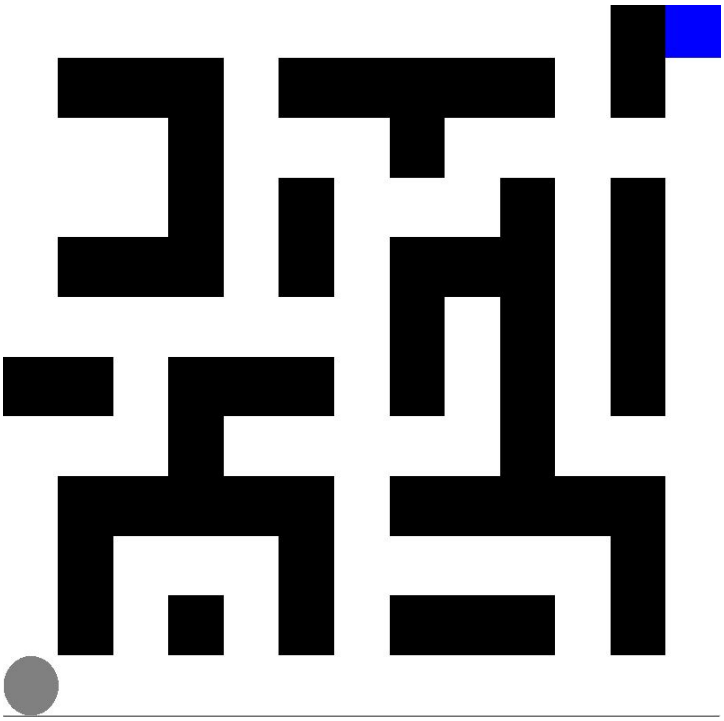
$$P_{catch} = .7 - log_{10}(health) + .3 * hasStatusEffect$$

Although pokemon have health up to the several hundred, attacks also deal damage up to the several hundreds. I assume a wild pokemon has 10 health, and each attack does 0 to 3 damage. With 10 health states, and one boolean status variable, we have 20 possible states. This felt relatively low, so I looked for an MDP with more states for my next problem.

Problem 2: A-maze-ing Gridworld

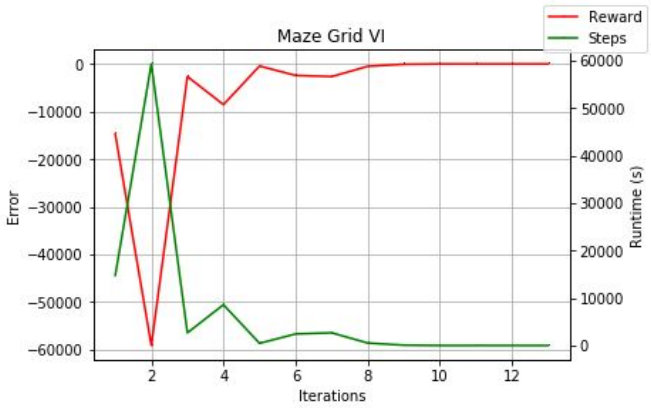
After spending large amounts of time getting my Pokemon MDP working, I also wanted something a little bit simpler to use. I have learned GridWorld about four times now in different classes, so it was an easy choice for my second problem. It has a custom amount of states (just change the map), so making a large problem is fairly easy. I thought about a few layouts for the map. I work with competitive soccer robots, where bumping into other robots counts as a penalty, and getting closer to the enemy goal to score could be modeled as a good reward. This would look like a large map with several small “islands” of negative reward as other robots. I decided not to do this for the problem, since it didn’t turn out that interesting. After some more playing around, I decided instead to build [1] a maze map. The maze has 98 states (98 reachable

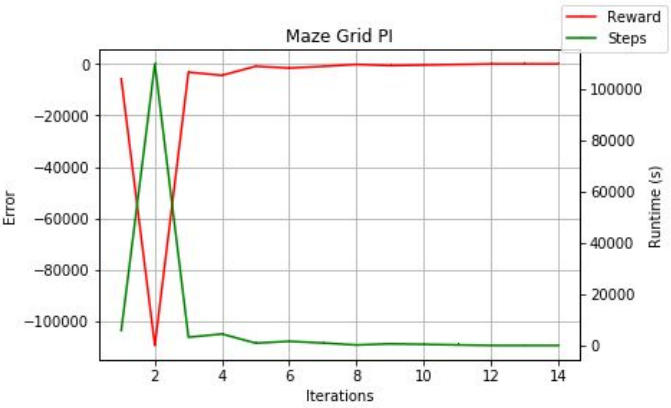
locations). I like it particularly because it rewards exploration more than a large and sparse map. For actions, you can go Up, Down, Left, or Right. There is some probability that you fail and go in a wrong, random direction. I used a .8 probability of success (i.e. going in the direction you would like to go), and then .067 for each other direction. You can see the maze I've set up to the right.



B. Iteration of the Values and Policies

I suspect you will be grading a lot of GridWorld papers, so I'll start Value and Policy iterations with my grid maze to give a good baseline.






```
[>,>,>,>,>,>,>,>,>,>,>,>,v,*,v]
[^,*,*,*,^,*,*,*,*,*,*,v,*,^]
[^,<,<,*,>,>,v,*,>,>,>,>,^]
[^,<,<,*,^,*,>,>,^,*,^,*,^]
[v,*,*,*,^,*,^,*,*,*,^,*,^]
[>,>,>,>,>,>,>,^,*,v,*,^,*,^]
[*,*,^,*,*,*,*,^,*,v,*,^,*,^]
[>,>,^,*,>,>,^,<,<,*,>,>,^]
[^,*,*,*,*,*,^,*,*,*,*,*,^]
[^,*,>,>,v,*,^,<,>,>,v,*,^]
[^,*,v,*,v,*,^,*,*,*,v,*,^]
[>,>,>,>,>,>,>,>,>,>,>,>,>,^]
```

```
[>,>,>,>,>,>,>,>,>,>,>,>,v,*,>]
[^,*,*,*,^,*,*,*,*,*,*,v,*,^]
[^,<,<,*,>,>,v,*,>,>,>,>,^]
[^,<,<,*,^,*,>,>,^,*,^,*,^]
[v,*,*,*,^,*,^,*,*,*,^,*,^]
[>,>,>,>,>,>,>,^,*,v,*,^,*,^]
[*,*,^,*,*,*,*,^,*,v,*,^,*,^]
[>,>,^,*,>,>,^,<,<,*,>,>,^]
[^,*,*,*,*,*,^,*,*,*,*,*,^]
[^,*,>,>,v,*,^,<,>,>,v,*,^]
[^,*,v,*,v,*,^,*,*,*,v,*,^]
[>,>,>,>,>,>,>,>,>,>,>,>,>,^]
```

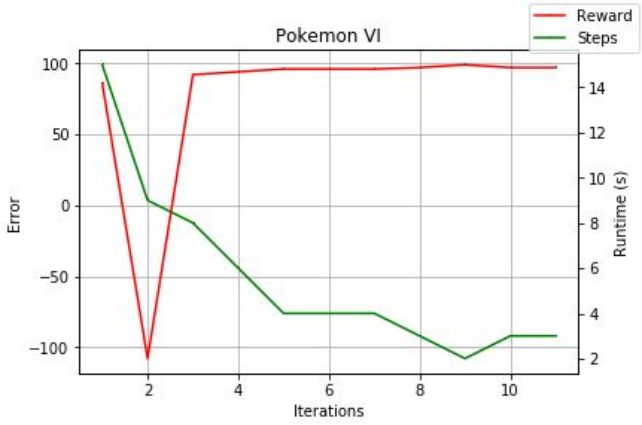
Theoretically, we could expect Policy Iteration to converge faster (in iterations) than Value Iteration, due to its more “macro” approach. Both PI and VI found a pretty decent solution in around 5 iterations. There is some strange bouncing in the second iteration for both algorithms as it produces its “optimal” solution very quickly, but it is a terrible solution. Both algorithms should improve every time, so it might be a bug in the implementation I used. Overall, both converged within 10 iterations. PI found its optimal solution by the third iteration, where VI was still bouncing around until the eighth iteration. This is more or less in line with our expectations.

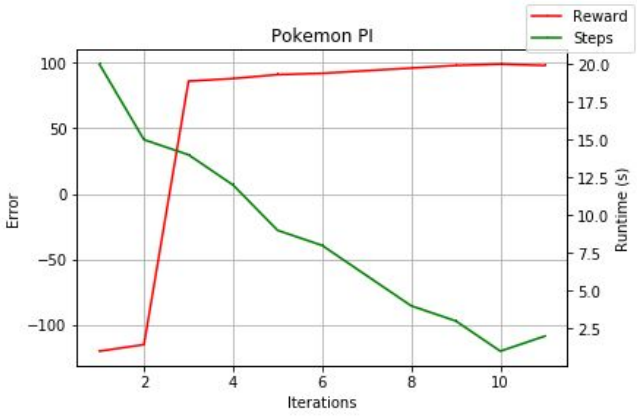
We might expect to see a higher runtime from Policy Iteration, since it is running mini-instances of Value Iteration. In fact this is accurate: PI took 36 ms on its 10th iteration, where VI took only 17 ms. There was some fairly random noise in the runtimes (just occasional very long runtimes with no change in performance), so I’m not going to graph them. But, for the most part, PI took twice as long to run each iteration as VI.

Though it is possible that they could turn up different solutions, it is unlikely. You can see the policies that VI and PI created above, below the utility graphs. They produced the same policy.

So now for Pokemon. With fewer states, we can expect faster convergence. This might be hard to demonstrate since we converged very fast with the maze (perhaps 100 states isn’t quite as large as I expected it to be). My Pokemon MDP has just 20 states, but the situation it represents is more complex (at least for me) than the maze problem. There is no tradeoff with the maze, either you get closer to the solution or you don’t. But

with Pokemon, every move comes at the expense of something else. If you attack or status attack, you didn't attempt to win and end the game. If you attempt to catch and fail, then you didn't improve your situation. With this in mind, I think using Reinforcement Learning is much more realistically helpful and interesting.





We see pretty similar results here. We have the strange "bounce" again from Value Iteration. This is at least partially explainable, we have a terminal state with large negative reward, which was not present in the maze. So if the "optimal" solution for that iteration knocks out the poor wild pokemon (the negative terminal reward), then it will end with large negative reward. We see that Policy Iteration initialized with a policy that knocked out the pokemon, and then learned not to do that. This might explain why we don't see the bounce on the second iteration. VI converged very quickly again, reaching near peak performance by the 3rd or 4th iteration. There is still some noise from the stochastic results (I think there is more "randomness" in the pokemon MDP than the maze). PI took until the 8th or 9th iteration to reach similar performance, it was

capturing the pokemon early, but taking too long to get there. Perhaps it was performing useless actions, like status attacking when there was already an applied status.

Value Iteration - Pokemon Catching										
Health	1	2	3	4	5	6	7	8	9	10
No Status	C	C	C	S	S	S	S	S	S	S
Status	C	C	C	A	A	A	A	A	A	A

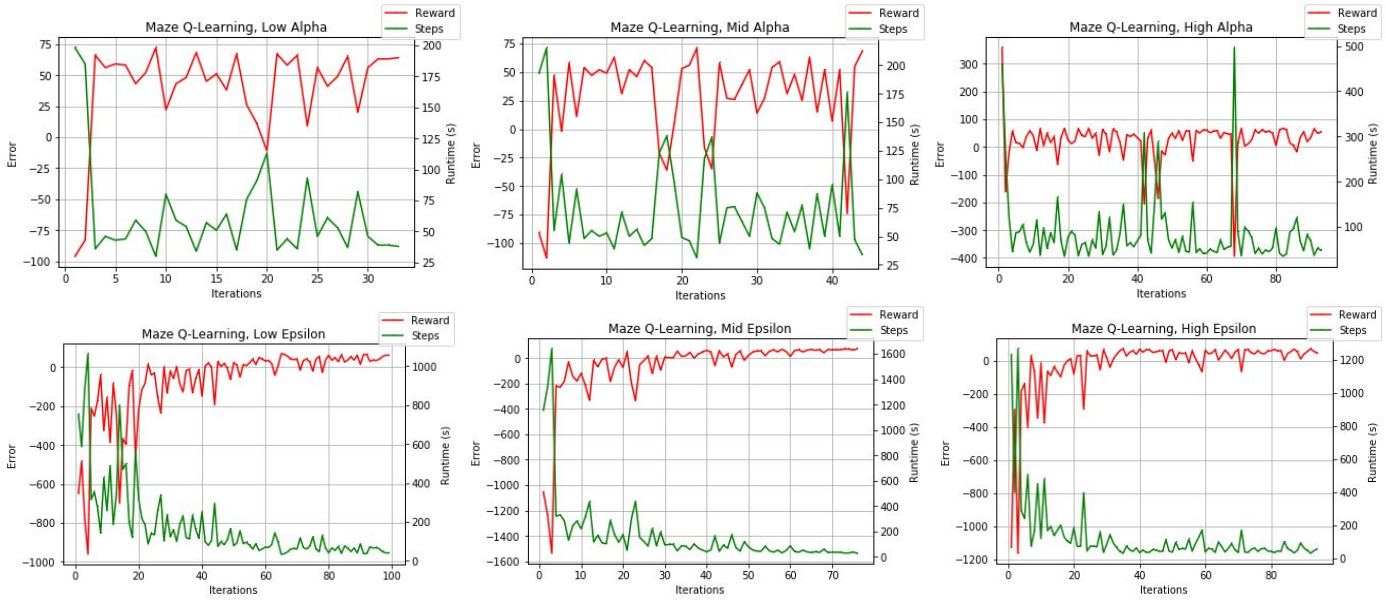
Policy Iteration - Pokemon Catching										
Health	1	2	3	4	5	6	7	8	9	10
No Status	C	C	S	S	S	S	S	S	S	S
Status	C	C	C	A	A	A	A	A	A	A

The above tables detail the policy produced by each algorithm. “C” represents attempting to catch, “S” represents a status attack, and “A” represents a normal attack. The general strategies are the same: apply a status effect, lower health, and then attempt to catch. There was actually a small difference in policy: Policy Iteration chooses to apply a status effect at health 3 where Value Iteration chooses to attempt to catch. I think we can chalk to this up to pokemon catching being up to random chance more than gridworld.

It is hard to compare runtime here, since both ran within 1 to 3 ms because it is such a small size. Comparing to grid world (which has about 5 times as many states), it runs in about 10% of the time. Again, it is hard to draw conclusions here because of the low resolution. I might guess that both VI and PI are running in linear to low polynomial time on the number of states.

C. My Favorite (and only) Reinforcement Algorithm

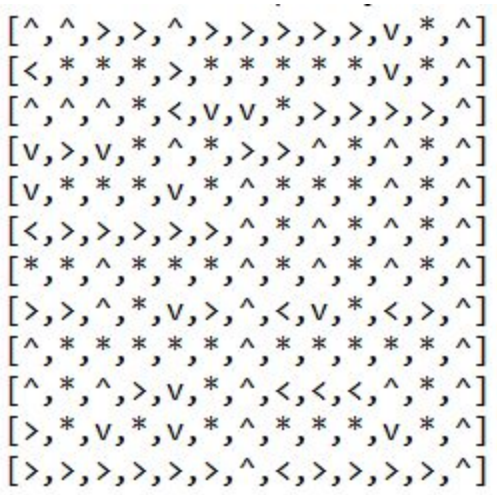
Full disclosure, the only RL algorithm I understand off the top of my head is Q-Learning, so naturally it is my favorite. Some basic parameter tuning is below.



Learning rate (alpha) is the first row. As we can see, it doesn't have a huge effect unless we really take it up high, but even then the best policy from low and mid alpha was about the same as the high alpha. The lower learning rate is, as you would expect, a smoother curve than the other values. It is somewhat interesting that the lower learning rate actually reached its peak faster than the other values.

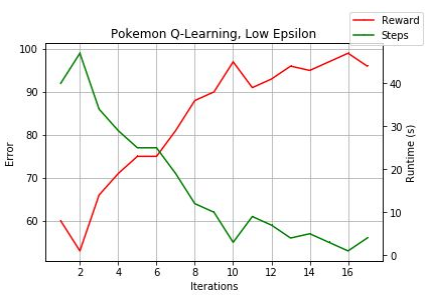
Even more interesting are our different epsilon graphs. A maze is conceptually a great testing bed for exploration, since there is a typically a large penalty for exploration that isn't fruitful - going down a long dead end for example. With different exploration strategies then (willing to explore, semi-willing, and reticent) we should see pretty different reward curves. We did in fact see some differences, but ultimately not as large as I expected. The low-exploration run, predictably, had difficulty actually finding the terminal state and so incurred a lot of penalty for wandering aimlessly. The mid and high exploration runs found the terminal state fairly quickly. Although the high epsilon had trouble repeating its performance early, as it would choose to explore again rather than find the goal. Ultimately the balanced approach with epsilon at .5 resulted in the best policy.

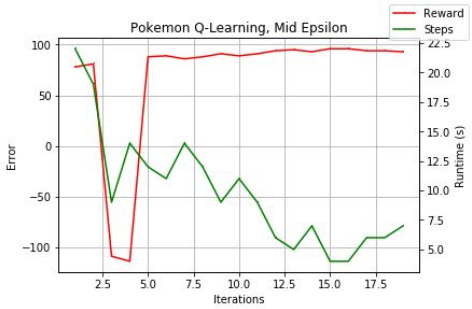
The policy it produced was actually quite different from the policies produced by Value Iteration and Policy iteration. The route from VI and PI was the simple route around the edge. But Q-learning actually takes us through the maze. Technically it is just as fast but it is a more complex solution, in that it has more turns and theoretically the stochastic nature of movement comes into play more. Additionally, there is some noise and nonsense in the policy, such as on the left side of the maze where it tells the agent to run straight into the wall. But, the general

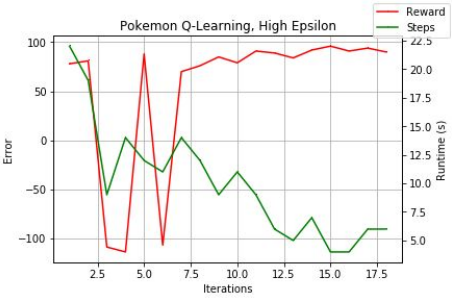


idea of the route is the same length and produces similar utility as our full-knowledge algorithms from earlier.

I like Q-Learning for Pokemon, because I think it's more extensible than earlier approaches. Some pokemon are harder to catch, or more resistant to attacks, or they might even be able to run away. Q-Learning is more adaptable to these circumstances. Let's see how it runs on our standard set up.







First, all of them converged much faster than our Maze runs. The pokemon simulation hit peak utility by around ten iterations, where the maze runs usually took around 60 iterations. Interestingly the low epsilon run never knocked out the pokemon so never reached negative values, likely because it initialized to something very safe and never explored more dangerous territory for attacking. The mid and high epsilon runs *did* knock out the pokemon, and actually did a few times in the name of exploration. Overall the mid epsilon again showed the most reasonable performance. Below is the policy generated by the mid epsilon run. I've also put in the low-epsilon policy, because i think it demonstrates some of the interesting aspects of q-learning.

Pokemon Low Epsilon										
Health	1	2	3	4	5	6	7	8	9	10
No Status	A	C	C	C	S	A	S	S	A	A
Status	C	C	C	A	A	A	A	A	A	A

Pokemon Mid Epsilon										
Health	1	2	3	4	5	6	7	8	9	10
No Status	C	C	C	S	S	S	S	A	S	A
Status	C	C	C	C	A	A	A	A	A	A

The low epsilon is advocating attacking when the pokemon has only 1 health left, which is downright cruel. Likely it never explored that area of the state space and so kept whatever was randomly initialized there. The mid epsilon policy is much more reasonable. It is a little different from the VI and PI policies (it attacks early with no status, and is also willing to catch a little earlier), but produces similar results.

D. Conclusion

I don't think I will be using Reinforcement Learning to pathfind through a maze, but it was surprisingly effective at catching Pokemon. Q-Learning surprised me; although it took a bit longer to converge, it ultimately reached performance equivalent to Value and Policy iteration.