Markov Decision Processes and Reinforcement Learning

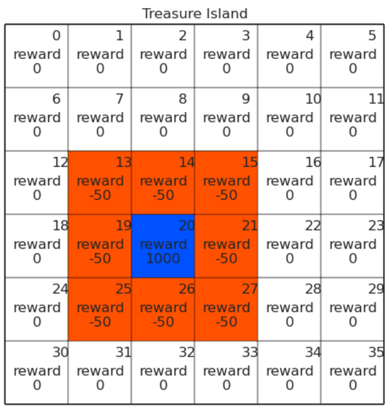
Jeff McGehee, OMSCS 7641

Introduction

Reinforcement learning is an area of machine learning which places an agent into a world with defined rules, and allows the agent to learn based on the stimuli (penalties, rewards) he receives from the world. As an engineer with a background in control design for electro-mechanical systems, this seems ideal for finding optimal control schemes. In fact, it seemed like an excellent experiment to test this out on my web-enabled Raspberry Pi thermostat[1]. But before I dove into that, I needed to do a bit of work to understand how RL works, and how to implement it. For this, I chose to create a specific instance of the standard Grid World problem, which I have named Treasure Island.

Treasure Island

The Grid World problem is a very simple problem to build and test using MDPs and RL. To make it even simpler to implement, I made the actions completely deterministic, so when the agent chooses an action, he always performs that action. The Treasure Island instance of Grid World that I will explore



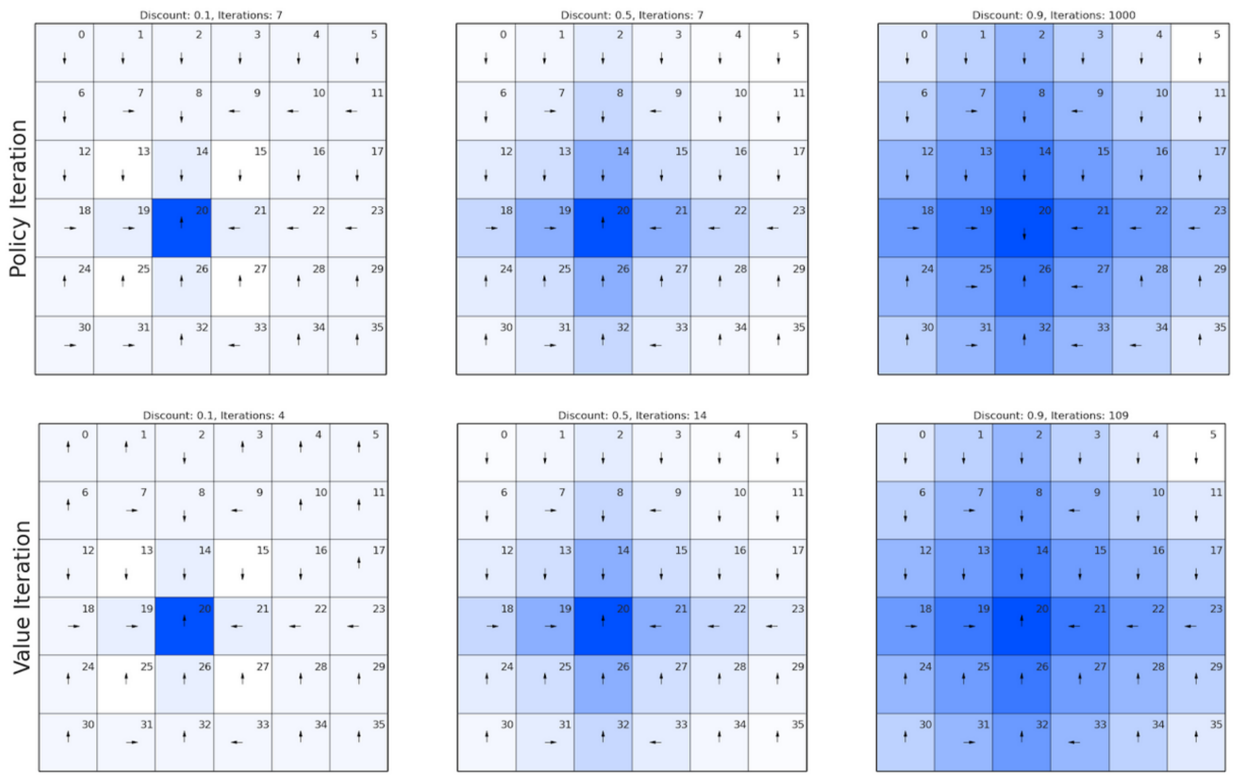
consists of a 6x6 grid, yielding 36 states. As may be deduced by simple arithmetic, the states are merely the positions on the grid. No other information is known to the agent.

[1] http://www.nooganeer.com/his/category/projects/homeautomation/raspberry-pi-thermostat/

Continuing to keep things simple, the rewards are only a function of the state. Every time the agent enters a state, he is given the reward associated with that state, no matter his action to enter. The name "Treasure Island" comes from the shape of the rewards, which can be seen in the figure above.
With this problem, we will let the Policy Iteration, Value Iteration, and Q Learning decide if it is worth the penalty to achieve a greater reward, and how long it takes the algorithms to make their decisions.

Policy and Value Iteration

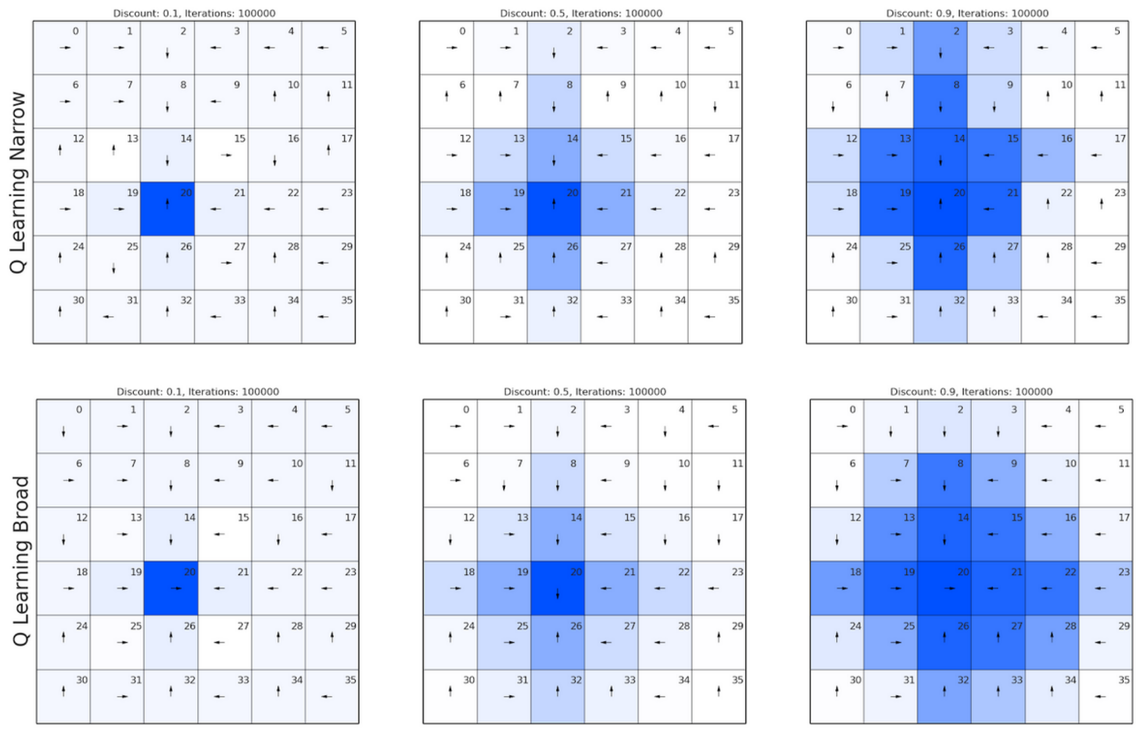
I decided to simply look at how policy and value iteration behaved in a scenario where it is obvious that the reward outweighs the penalty (reward >> penalty). The Treasure Island problem is interesting because it highlights the effects of discount on the algorithms. This can be seen in the figure below, where I varied the discount for each algorithm.



In the figures, I have shown the optimal policy as shown by the given algorithm (keep in mind that square 20 is the "Treasure Island"), where the color of each square represents the relative value of the value function at each location. Looking at the colors, we can see that as the discount increases, the relative values of the positions become closer, showing that as expected, the algorithm is putting greater importance on each state, even if it does not give an immediate reward. We can also notice that as discount increases, so do the iterations. I was surprised to see this has a stronger effect on PI, since PI typically converges quicker than VI . When examining the policies provided by PI and VI, it can be seen that even with a low discount, PI finds a policy that will always land the agent on Treasure Island. However, VI, is not able to do so at the low discount value because in this case where many states have zero value, it is possible that many policies share the same value function. I also found it interesting/exciting that the algorithms were able to determine that they should avoid the corner squares of the penalty area because it would cause a double penalty before reaching Treasure Island.

Q Learning

Q-Learning takes a much different approach to finding optimal policies for the MDP. Where PI and VI must know the model before they begin their *search*, QL has no concept of the model when it begins *learning.* This causes for much more iterations, but can result in more “creative” solutions. For a simple model such as this, it also puts QL at a bit of a disadvantage. In order to analyze QL’s behavior, I performed a similar simulation to the one performed for PI and VI. The results can be seen below.



The top row of plots is a “narrow” search, or a more greedy execution of QL. The bottom row shows an algorithm that continues random searching further into its iteration. The first thing that should be noted for QL is that I had to run it for substantially more iterations than PI and VI to find similarly performing policies, especially for higher discounts. This makes sense because QL does not know the model, which makes it difficult to interpret future rewards (which is necessary at high discounts). The fact that QL performs "poorly" at higher discounts is not really a problem here because, this problem doesn't really require that we perform well initially in order to achieve a high score. You can see that for a low discount, both the broad searching and narrow searching (greedy) examples find policies that are close to equivalent to those found by PI and VI.

Smart Thermostat

When I decided to test MDPs and RL on my Raspberry Pi thermostat, I knew it would be a big problem to tackle, so I contacted a fellow student (Michael Simpson) whom I had worked with before (porting MIMIC to python, mimicry). After receiving approval from the instructors, we worked together to formulate the problem, but carried out our final simulations and analyses individually.

The goal of this problem is to use information that can be made available to a web-connected thermostat as a means to improve the efficiency of a home's HVAC system while maintaining a reasonable climate.

Defining States

The first part of the problem was to settle on the proper way to define the states. The number of states for a real-world physical system like this could be extremely large if we were not careful in our design. Though this was a lengthy process to come to these conclusions, I will just jump to the choices with a bit of explanation of each item. The state array was of the following form:

$$[mode,\ \Delta T_{internal},\ CDD_{3hour},\ CDD_{6hour},\ CDD_{9hour},\ CDD_{12hour},\ \Delta T_{external}]$$

The mode is the most obviously the mode of the thermostat. $\Delta T_{internal}$ is the difference between the desired temperature and the actual internal temperature. CDD stands for "Cooling Degree Days", which is an industry term, but is essentially the integral of the difference between the desired temperature and the outdoor temperature. The subscripts represent hourly bins of CDD as could be calculated using a weather API. These represent the CDD 3, 6, 9, and 12 hours into the *future*. This will allow the thermostat to "plan" for future temperature swings. The final parameter, $\Delta T_{external}$ is the difference between the internal temperature and the external temperature. It should be pointed out that all of the parameters are *relative* numbers, meaning that in the end, the agent should be able to function for any desired temperature set by the user. Still, all of these states had to be discretized judiciously, and this was done as follows:

$$mode \in ['cool',\ 'fan',\ 'off']$$

$$\Delta T_{internal} \in [-5,\ -3,\ -1,\ 0,\ 1]$$
$$CDD_{Xhour} \in [-2,\ 0,\ 2]$$
$$\Delta T_{external} \in [-5,\ 0,\ 10,\ 20]$$

Even with such careful planning, and deciding to only handle cooling scenarios, the total number of states exceeded 4500. This meant it would be impossible to generate transition probabilities and rewards intuitively. This fact makes Q-Learning an appealing approach for this problem because of it's model-free nature, but in order to test PI and VI, we must build a transition/reward model. To do this for such a large problem, we decided to use a monte carlo simulation to build the model for us. However, we also needed a "world" in which the agent would act, which simulates real-world weather/house/thermostat behavior in order to make this possible. Michael has more experience with monte carlo simulations, and I have more experience with physics and engineering models, so he built the monte carlo sim and I built the thermostat "world" model.

The Thermostat World

Not only would we need the thermostat model in order to build the necessary matrices for the MDP, but we would use it to test the performance of our policies. The model consists of a pseudo-random weather generator, a house, and a cooling unit (cool, fan and off modes). Simple linear equations were formulated to describe the behavior of the indoor temperature of the house during all of the cooling unit modes. These equations can be seen in the table below. Notice that the fan cools at ¼ the rate of normal cooling mode. This may not be completely accurate for all homes, but in my home, I have a finished basement area which remains about 8F below the rest of the house. Hence, circulating the air in the home will have the effect of mixing in this cool air with the rest of the house.

Cooling	Fan	Off
$T_i = T_{i-1} - \eta + \alpha T_{dif}$	$T_i = T_{i-1} - \frac{\eta}{4} + \alpha T_{dif}$	$T_i = T_{i-1} + \alpha T_{dif}$

where: η is the cooling rate in degrees per time step, α is the heat loss/gain coefficient of the house, and T_{dif} is the temperature differential between the outside and inside of the house.

An important part of this simulation is that it models the real world, and we just place the agent in it. The temperatures, etc. are continuous, but the agent is only able to know which bins they fall in. In order to keep things simple for the assignment, all simulations used to run the Monte Carlo simulation were performed with a target indoor temperature of 70F.

Reward Function(s)

For this problem, I chose to investigate two reward functions. For the most part, it is difficult to know if any of the many <state, action> tuples is worth *rewarding*, but it is quite easy to know what is worth *penalizing*. For both reward functions, I applied the following power usage penalties, in terms of a "power importance factor, ϱ", based on the agent actions:

$$R_{cool} = -\varrho\,,\ R_{fan} = -\varrho/10\ \ \textit{where: } \varrho = 4$$

This is appropriate because the typical fan in a HVAC unit operates at 1/10 or less the power of the compressor that must be run when cooling.

Of course, the whole purpose of a HVAC unit is to keep the home at a comfortable temperature. So, for both reward functions, I chose to penalize undesirable temperatures, and reward hitting the target temperature. I called this reward (the only reward given to the system) the "Goldilox Reward". The comfort reward/penalties, in terms of a "comfort importance factor, σ", are defined as follows:

$$R_{comfort} = -\sigma * |\Delta T|\,,\ \Delta T \geq 1$$

$$R_{comfort} = -\sigma * |\Delta T|\,,\ \Delta T \leq -5 \qquad \textit{where: } \Delta T = T_{indoor} - T_{desired}\,,\ \sigma = 1$$

$$R_{comfort} = \sigma\,,\ |\Delta T| < 0.5$$

This concludes the first reward function, which I will give the very creative name "Reward A". The second reward function (I'll call this one "Reward B") has an added penalty for switching the compressor on when it is off in the current state. This is because the startup current of the compressor of your HVAC unit is likely 500%-700% higher than it's constant operating current. Spreading this out over the 5 minute timestep of the simulation, we apply the following penalty to "Reward B":

$$R_{startup} = -10 * \varrho\ \ \textit{where: } \varrho = 4$$

Therefore, the two reward functions being evaluated are:

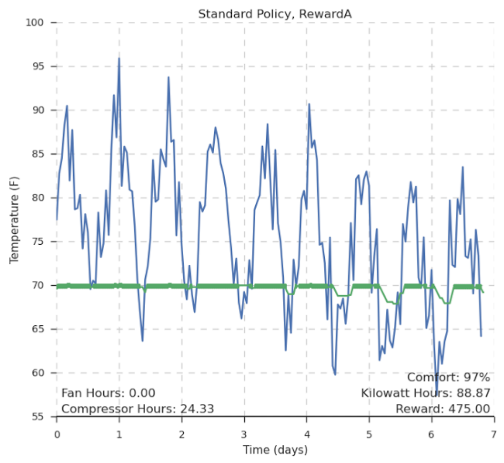
$$R_A = R_{cool} + R_{fan} + R_{comfort} \quad , \quad R_B = R_{cool} + R_{fan} + R_{comfort} + R_{startup}$$

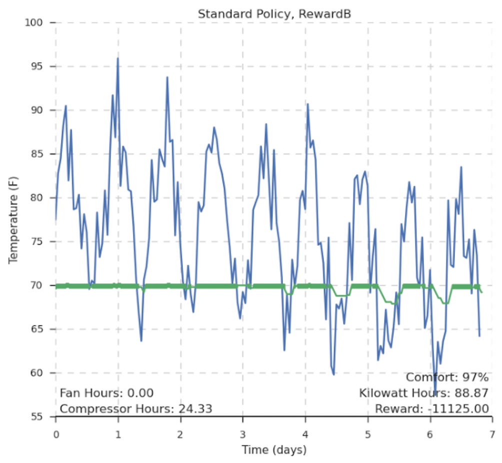
Note that for both equations, a very important factor in the behavior of the algorithms will be the ϱ/σ ratio. This defines the relative importance of power consumption versus comfort. For all experiments, we set this ratio to 4/1, but it would be very interesting to do a study of the solutions for different ratios.

Measuring Real World Performance

The last step before solving the MDP intelligently, is to set a baseline behavior for a typical thermostat. This will allow us to determine the successfulness of our learned solutions. To do so, I just implemented a simple policy that runs in "cool" mode any time the indoor temperature is above the desired temperature. **When evaluating the performance of this control strategy, as well as the others, I will look at the total reward achieved, but I will also look at the simulated KiloWatt Hours and what I will call the "Comfort Factor", which is the percentage of time spent within 2F of the desired temperature.** For the majority of my analysis, I will use the following results as a baseline. This is a simulated "Default Policy" with desired temperature of 70F over a pseudo-random 7 day temperature profile roughly centered around 75F. For all simulations compared to these, I will set the same random seed to eliminate any random variations. Below are the results for RewardA and RewardB. Notice that the temperature profiles are identical (since it has no sense of reward), but the scores are different.

Here, we can already get a sense of how the different reward functions may play out. We can see that when we don't penalize for compressor startup (RewardA) we get a positive total reward because we maintain a Goldilox Temperature, however (RewardB) penalizes this strategy heavily because of the frequent compressor startups. Comfort Percent is less than 100% only because the outdoor temperature cools the house.

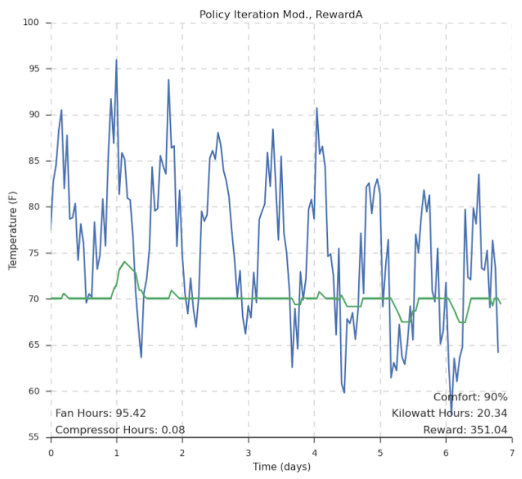


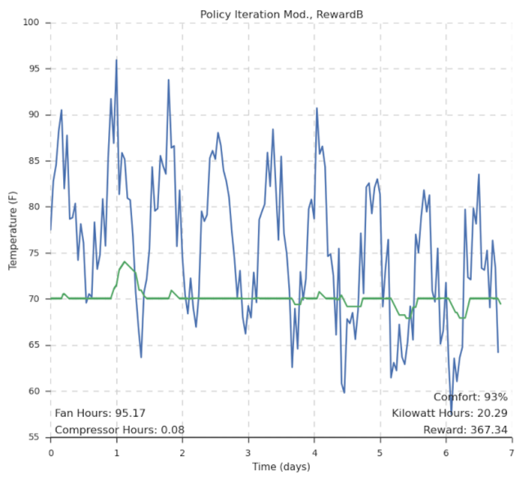


RL Results on Default Simulation

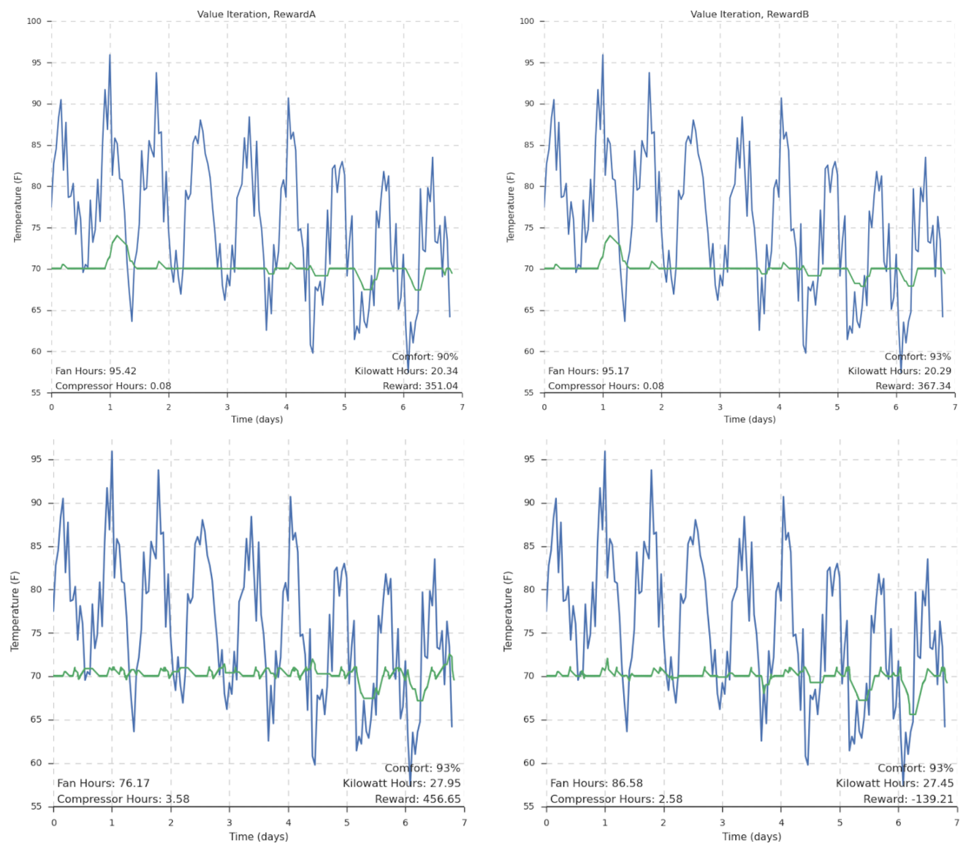
Below are examples of policies found by PI, VI, and QL for the same scenario as the Standard Policy plots above. Some notes about what it took to get these results: First, PI took a prohibitively long time to run on this large state-space problem. It is known to take longer per iteration compared to VI, which seemed to be the problem (minutes per iteration). I solved this by using MDPToolbox's[2] modified PI, which makes a trade-off between speed and effectiveness per iteration[3]. Also due to the large state space, I applied an initial Q for QL that valued 'off' states highest and in order to decrease the iterations to an effective solution for QL, I imparted a very wide initial search and forced no decay on the learning rate. Because QL has no concept of the model, I also noticed that a higher discount improved progress per iteration. Later, I will show results regarding performance vs. discount, but the policies in the plots below are for a discount of 0.9 for all algorithms. This gives them all an equal opportunity to find a solution.

It can be seen that all policies result in an energy reduction. PI and VI actually reach identical policies for both RewardA and RewardB. PI and VI seem to favor energy reduction for both reward schemes. QL is able to achieve a large reduction in energy consumption for RewardA and RewardB, but does so in a decidedly different manner which suits RewardA well, but is not as good for RewardB.



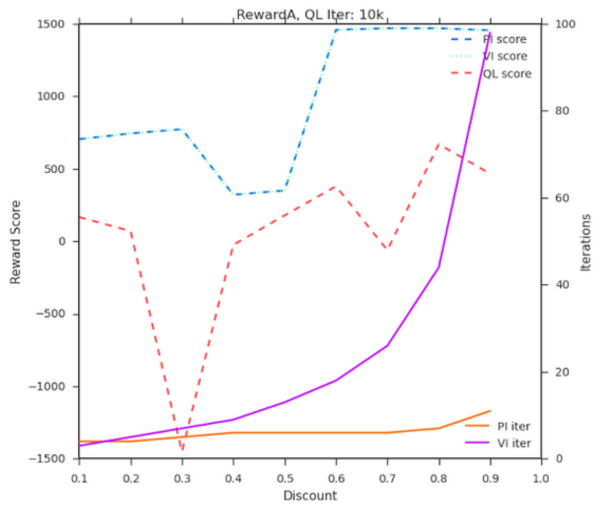


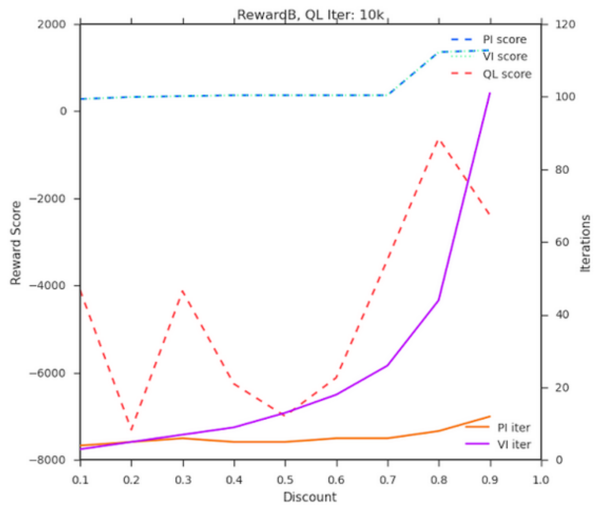
[2] https://github.com/sawcordwell/pymdptoolbox
[3] Reinforcement Learning: A Survey--Kaelbling, Littman, and Moore



What’s Going on Here?

In order to get a better understanding of how these algorithms solve this problem, I plotted performance and convergence iterations versus discount.





We can see that even at high discounts, the iterations of PI and VI don't approach the 10k that QL is running (although VI is approaching quickly). So PI and VI seem to have a clear advantage when the world behaves similar to the model, but as can be seen in the comparative time plots, QL can come up with some strong solutions as well. We can also see that due to the fact that I am forcing QL to perform a wide search, it's behavior is a bit inconsistent. I do not have the room or time to include it in this paper, but it would be interesting to investigate the distribution of scores achieved by QL over a significant number of runs.

This problem is so large with so many possible solutions and ways to evaluate the results. I tried to hit the points here that I thought were important, but unfortunately much had to be left out from this paper. The results show that PI and VI perform best, and all algorithms outperform the standard policy in terms of power consumption. It is common knowledge that if the user is willing to sacrifice his house being at the perfect temperature, we can see great energy savings. This study proves that it is especially true when our thermostat can plan for the future. QL came out looking like the loser in this study, but it really has some advantages. First, it does not require a model in order to learn, which in this case would have saved a large amount of effort. PI and VI are completely reliant on a model and cannot learn the way QL does. Second, it was difficult to build a "perfect" reward function, because there is always a trade-off between power consumption and comfort. It seems that QL follows the "spirit" of the reward function as opposed to the "letter" of the reward function (unlike PI and VI). This means, as can be seen in the time plots, sometimes it will find solutions that perform better than PI and VI.

Real-world Application

I plan on implementing a RL strategy for my thermostat. Based on the understanding I have gained from this project, the strategy I will most likely choose is Q Learning. This is because I can define an initial Q which roughly estimates state values, then let QL learn my home's system without trying to model it using a giant Monte Carlo simulation. Also, for real implementation, I cannot assume that my Monte Carlo simulation of my home would hold in all cases. This would cause PI and VI to break down unless I could have a continuous Monte Carlo sim that's constantly updating the model parameters (this sounds expensive!). I am an an engineer (maybe when I finish OMSCS, I'll be a computer scientist), and engineers like tools that work in real life. My impression of PI and VI are that they are great in theory, but if the real world behaves in a way they don't expect, they will fail. QL seems more robust to this type of world behavior, which is why I would choose it to interact with the real world.