Raphael Gontijo Lopes
903062081

CS 4641
Assignment #4

MDPs & Reinforcement Learning Report

Problem Description

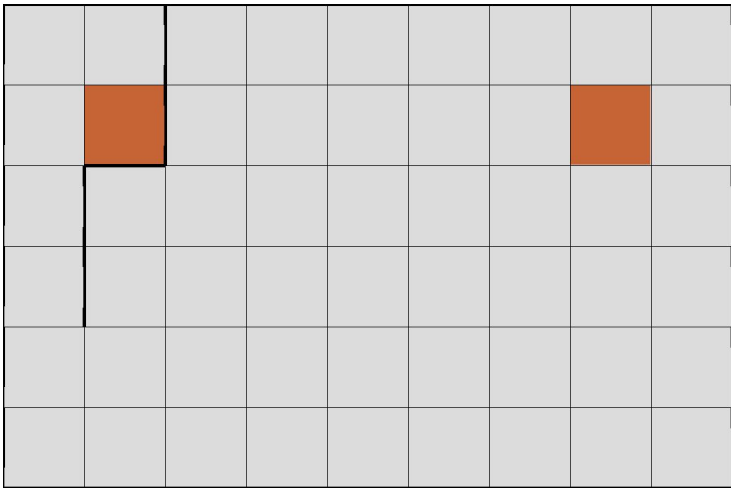
A Markov Decision Process is a discrete, time stochastic control method. They are used to model some process, based on the real-world or not, and provide a framework for decision-making. In this report, the problem being modeled is that of autonomous household cleaning robots (Roombas and the like). More specifically, the problem of needing to return to its charging station before its batteries run out.

To model this problem, I have used the Grid World MDP. In it, an agent can be in one of many discrete physical states. It can take actions by moving up, left, down or right. To motivate the agent to reach the goal (the charging station), each action taken has a constant "path cost". There is no reward given for reaching the goal state, nor is there a path cost for it.

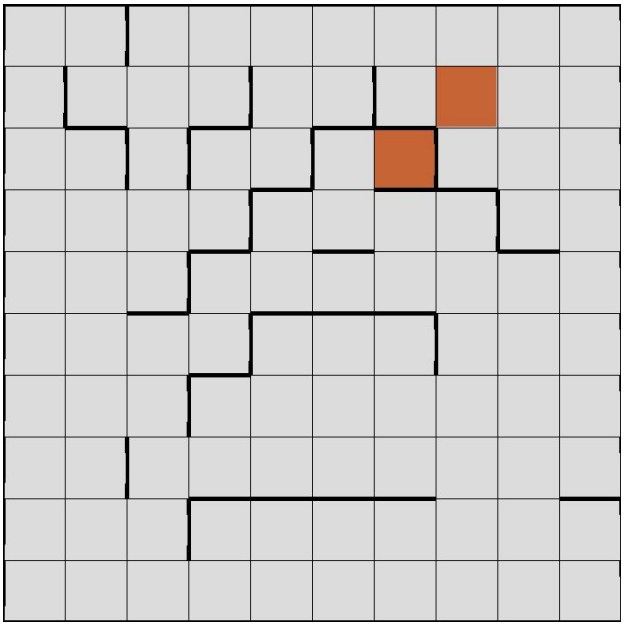
The transition function is, however, not fully deterministic. Given a constant 0 < PJOG < 1, the agent's action is effective with probability (1 - PJOG). The probability of the action resulting in a transition to any of the remaining states is (PJOG / 3) for each state. Moreover, much like in the real world, there is negative reinforcement given for when the agent runs into a wall.

My two MDPs are different variations of this problem. The goal of this paper is to solve these MDPs using different reinforcement learning techniques, in order to compare the algorithms' performances.

Map 1 (4_multipleGoals2.maze) – This is a relatively small map that I used to run the initial experiments. In all experiments ran using this, I chose to use PJOG = 0.1. That is to say all actions taken by the agent were fairly deterministic (being effective 90% of the time).



Map 2 (7_medium_complex_multigoal.maze) – To make things more interesting, I also used a much larger map, with a higher number of states, because it helped make comparisons of the algorithms and point out their relative strengths/weaknesses. In this problem, I used PJOG = 0.3, which means the agent would have to learn optimal policies despite having non-deterministic actions.



I believe these problems are interesting not only for their real world application but also because their differences should be able to highlight differences in the algorithms we discuss here.

Algorithms

Value Iteration

This algorithm works by trying to estimate the value (goodness) of being in a certain state. This value (which depends on the state being considered) is assumed to be dependent on future rewards, expected for actions subsequent to it.

This “value” equation, V_k(S_i) is defined as “maximum possible future sum of rewards that can be obtained, starting at state S_i, in k time steps”. This means that V1(S1) is the immediate reward of being in state S1, while V2(S1) is a function of the immediate reward and of V1(S1).

Value iteration is an example of dynamic programming, which means that the values of all states are computed, stored, and updated according to the following equation, until V

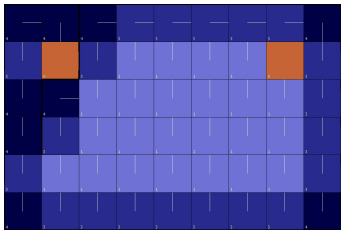
$$V_{t+1}(s) = \min_{a \in A} \left(PCost + \sum_{s'=S} (P(s'|s,a) \cdot x_{ss'}) \right)$$

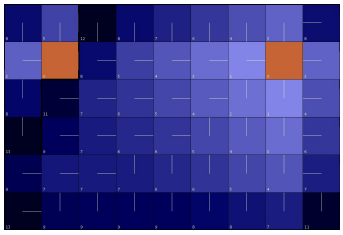
where,
PCost = PathCost
$P(s'|s,a)$ = Prob. of transition from s to s' after action a.
$x_{ss'} = V_t(s')$, if transition from s to s' is safe
$= Penalty + V_t(s)$, if transition from s to s' is not safe

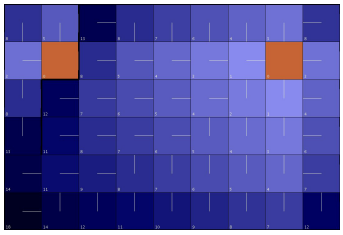
converges.
In our problem domains, all reinforcement is negative (the goal state has 0 reward value). Therefore, the goal of the agent is to minimize reinforcement. Thus we can define the optimal policy to be:

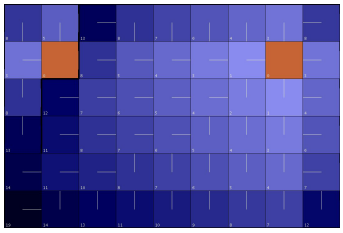
$$\pi^*(s) = \arg\min_{a \in A} \left(\sum_{s' \in S} (P(s'|s,a) \cdot x_{ss'}) \right)$$

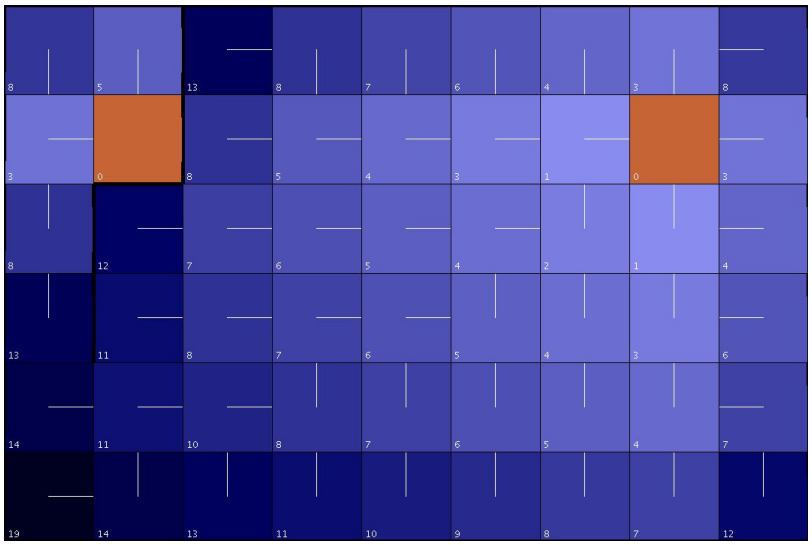
The first experiment I ran was value iteration on the small map. It took a very small time, and number of iterations to converge. Here are the results:


Iteration 1


iteration 7

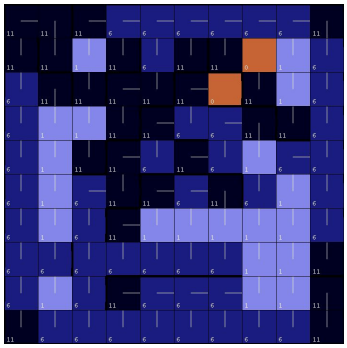

iteration 13


iteration 19

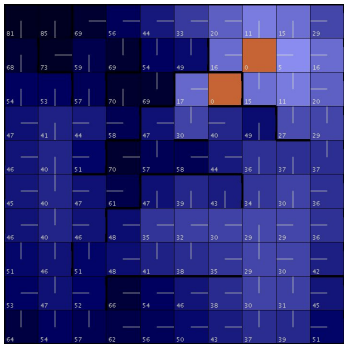

Iteration 24
(Converged)
2 ms

It is interesting to note that, even with a small noise in the transition model (small PJOG), many states prefer taking a longer path if it means being able to avoid walls (which, as mentioned, can provide negative reinforcement if ran into). An example of this is the four states at the very bottom left corner. They are much closer to the leftmost goal than the rightmost one, but they prefer going the long route than trying to go through the narrow path up. If this behavior is not desirable, one can simply set the wall penalty to be smaller (it is currently 50).

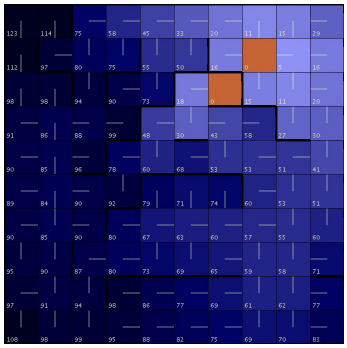
The second problem also showed some interesting results:



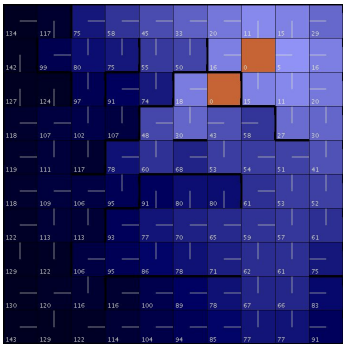
Iteration 1



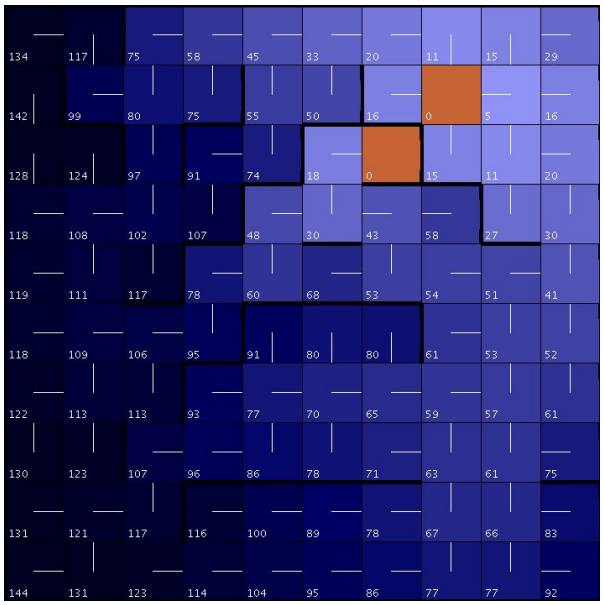
iteration 17



iteration 35



iteration 56



Iteration 84
(Converged)
52 ms

Here, it is interesting to see the growth in number of iterations and time until convergence, when compared to the first problem. This is expected, as the large map is about twice as big as the small map, but number of iterations/time grew at a much higher rate than 2. This is probably because, as the number of states grow, so does the number of expected rewards that affect

other states' values. Furthermore, the time until convergence grew as much as it did mostly because, for every state, value iteration must consider all four actions. So value iteration grows nonlinearly for both time and number of iterations.

Policy Iteration

Policy iteration differs from value iteration in that. With the latter, we try to estimate the value of every state, based on the value of their neighboring states. With the former, you estimate the reward of an action at a state, not of the state itself. Thus, with Policy Iteration, you directly manipulate the policy

Once again, we must compute the value of every state on the board, but this time, instead of assuming the action taken at every state will be the one that leads to the highest valued neighboring state, we instead use the current policy to make decisions:

$$V_{t+1}(s) = PCost + \sum_{s' \in S} (P(s'|s, \pi(s)) \cdot x_{ss'})$$

where,

$P(s'|s,a)$ = Prob. of transition from s to s' after action a.

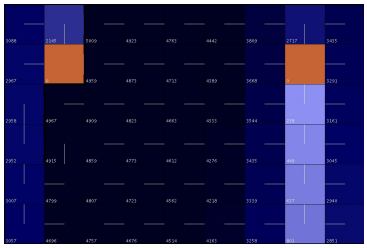
$x_{ss'} = V_t(s')$, if transition from s to s' is safe

$= Penalty + V_t(s)$, if transition from s to s' is not safe

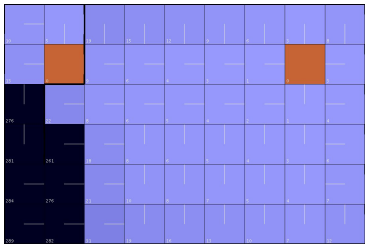
After the board has converged, we update the policy based on what we know about the expected future rewards:

$$\pi_{t+1}(s) = \arg\min_{a \in A} (\textstyle\sum_{s'=S} (P(s'|s,a) \cdot x_{ss'}))$$

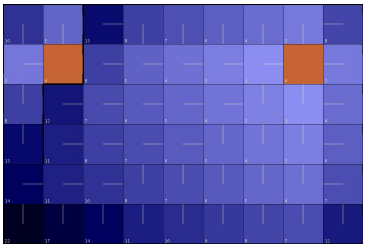
Policy iteration was the second experiment run on the problems. Results for the small map:



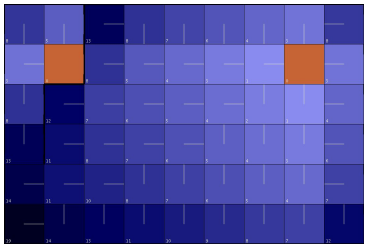
Iteration 1



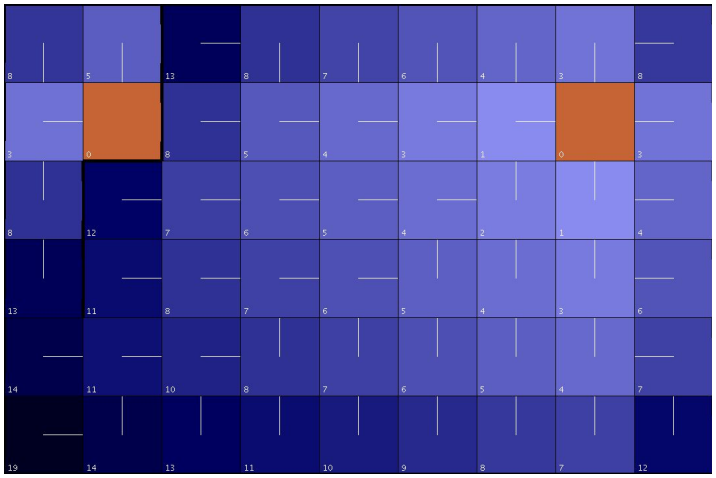
iteration 2



iteration 3



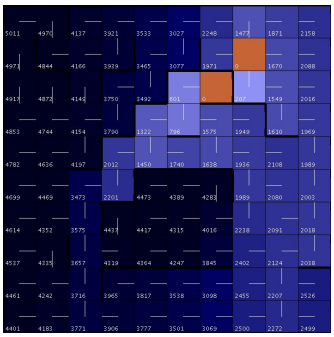
iteration 4



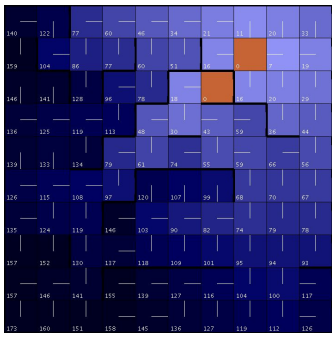
Iteration 5
(Policy Converged)
14 ms

These initial results seem very impressive, considering policy iteration was able to converge in 1/5 of the number of iterations, when compared to value iteration. We note, however, that it actually took longer to converge, which is expected because each iteration of policy iteration involves 1 convergence of the state values, and 1 policy update. This is the reason why there are modified versions of policy iteration in which the value update step is performed, but not until convergence.

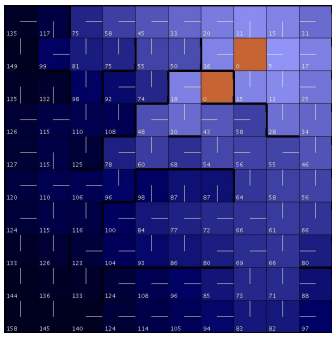
For the larger map, the results are as follow:



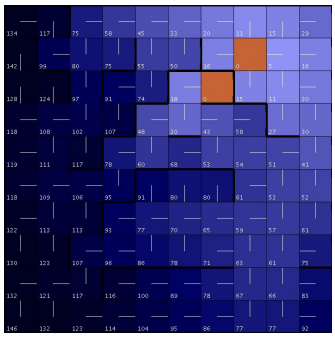
Iteration 1



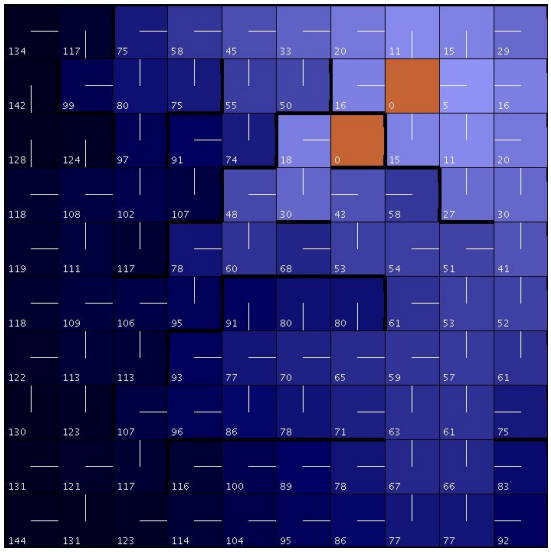
iteration 2



iteration 3



iteration 5

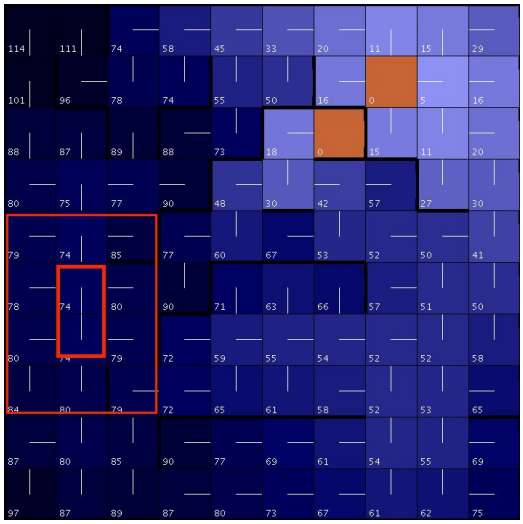


Iteration 7
(Converged)
36 ms

For the larger map, we see that the drop in number of iterations required to converge was so significant, that it was able to achieve a better time than by using value iteration. This result shows that, if the problem at hand has a large number of states, it is valuable to use policy iteration over value iteration.

We also note that in both small and large maps, both policy iteration and value iteration converged to the same answer. This is because both algorithms are attempting to model the same thing (the values of each state). The only difference is that one of them models the values directly, and the other indirectly (by directly modeling the policy, which depends on the value function).

But this is only true if one is using the non-modified version of policy iteration. In the modified version (the one that does not update values until their convergence), policy iteration not converge to the same answer as value iteration. In fact, by limiting the number of value updates, the policy reached can be non-optimal. In the following image, note the non-optimality of the highlighted parts of the map.The idea behind the modified version is achieving faster convergence of policy, so there's a tradeoff to be made here.



[modified policy iteration - max 2 value updates]

Iteration 30
(Converged to non-optimal policy)
8ms <<< very short time!

Q-Learning

One of the biggest drawbacks of the algorithms seen so far, is that they require that the model (the transition model and the reward function) be known by our agent. However, it is easy to imagine a scenario in which this is not the case. Say, for instance, that our cleaning robot functions perfectly for a year, but then it a malfunction occurs and now, every time it tries to move up, the chance of it moving left becomes 20%.

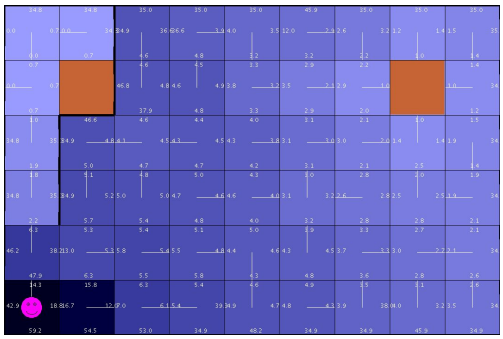
Q-Learning is a reinforcement learning method that does not assume a known model. This *model-free* approach attempts to learn a policy directly, without learning the model at all. It work by estimating the value of each state-action pair Q(s, a). This value is the expected sum of future reinforcement after (s + a). Once the Q-values are known, the optimal policy is that which takes the action with lowest Q-value (recall that in our problem domains, all reinforcement is bad, so we should minimize it). To obtain the Q-values, our agent updates the initial guesses with by experiencing reward and using the equation to the right:

$$Q(s,a) = Q(s,a) + \alpha \left(X + \min_{a' \in A} (Q(s',a') - Q(s,a)) \right)$$

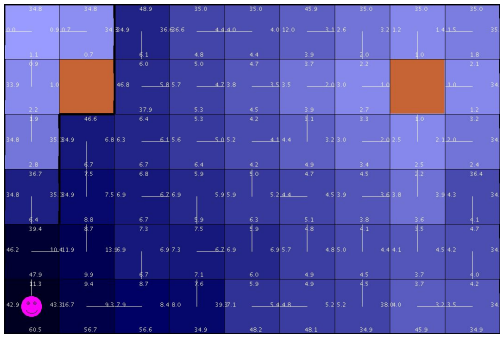
$$\pi^*(s) = \arg\min_{a \in A} (Q^*(s, a))$$

One of the things to note in the second equation is the learning rate alpha. It determined by how much the Q-values are updated. In noisy environments (ones with high PJOG, like the large map), high learning rates make it hard for the system to stabilize, but small learning rates make learning very slow. The alternative to either of these is to use a decaying learning rate, which is what was done.

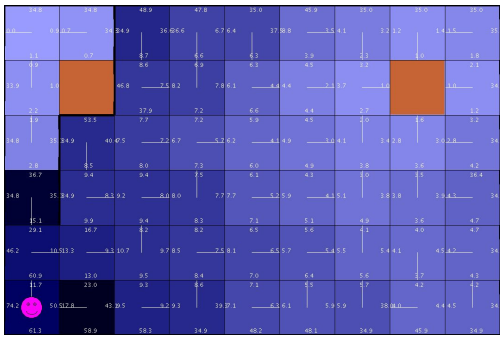
One of the biggest problems with this approach is the fact that, to converge to optimal Q-values, each state action pair needs to be explored a considerable number of times (ideally infinite many times each). If our agent gets stuck in local optima, exploration becomes a problem. Part of it is solved by the non-determinism of the transition model (PJOG > 0), which forces exploration. But it is not enough, especially for cases like our small map, which have small PJOGs. To solve this, we use an exploration policy that takes the highest Q-valued with probability (1 - E), and takes a random action with probability E. For our experiments, I have chosen E to be 10% and 25%.



25 episodes, E = 0.1

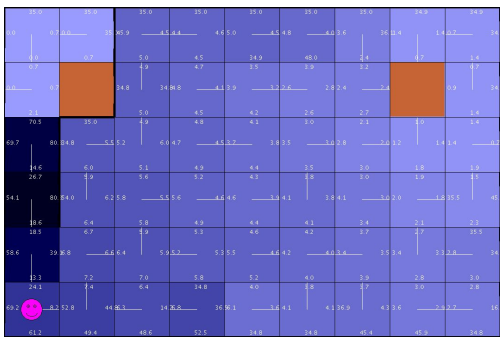


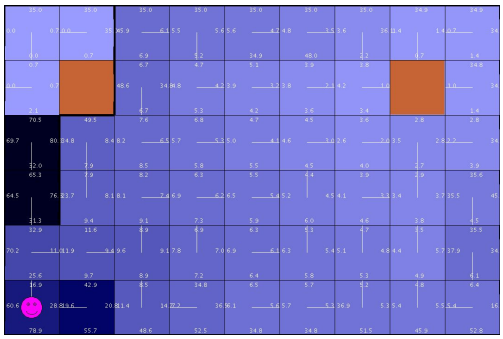
50 episodes, E = 0.1

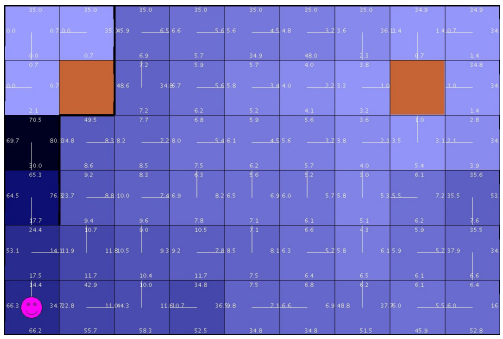


100 episodes, E = 0.1

We see that even with a small number of episodes (each of which run faster than either value or policy iteration), Q-Learning was able to learn very important features of the map, such as the fact that taking the path up the narrow hallway is non-optimal, because of the chance of running into a wall. What this also means is that the narrow hall was very little explored, even with our exploration strategy. The result is a non-optimal policy for the top leftmost states, even after 100 iterations. So we try an exploration rate of E = 25%.

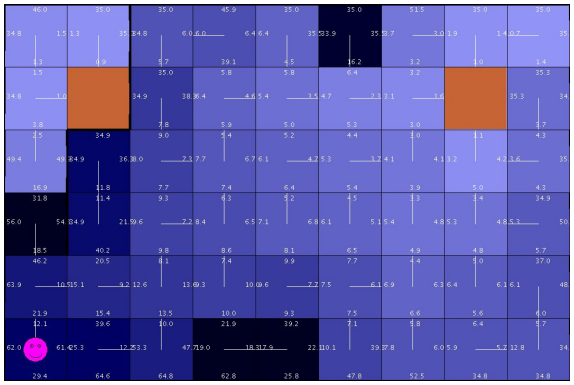




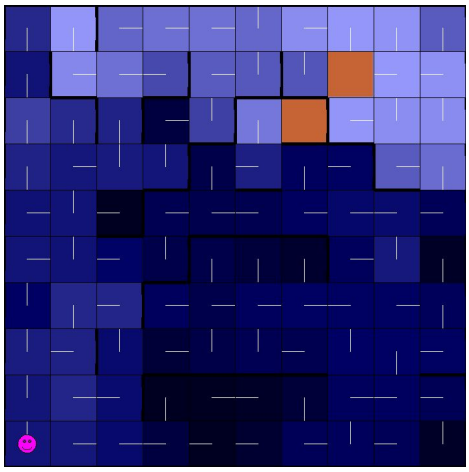


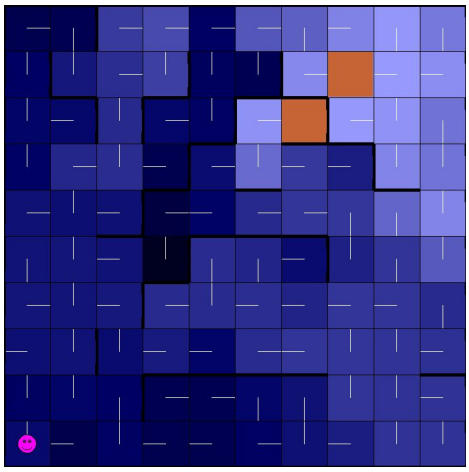
25 episodes, E = 0.25 50 episodes, E = 0.25 100 episodes, E = 0.25

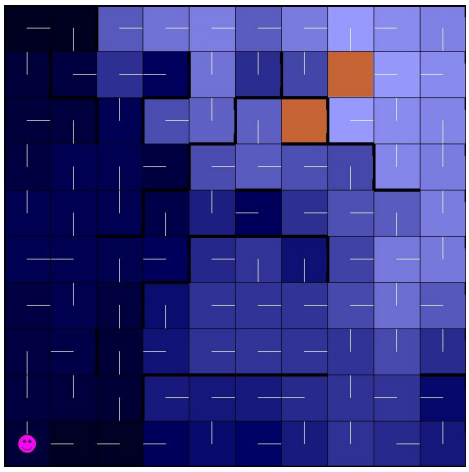
It's very clear that this was too high an exploration rate. It essentially added to PJOG, making it harder for the agent to make informed decisions about which action to take. So perhaps the best way of solving our initial non-optimality problem was simply to run the experiment for longer episodes:



250 episodes
E = 0.1
[Top left optimal policy]
[some non-optimalities still exist]
Now, for the large map, we see that with a small number of iterations, the agent becomes very wary of going near the middle part of the map, simply because of the sheer number of walls present there. When we compare this to the results in value/policy iteration, we see that this is a non-optimal strategy. It is corrected with subsequent episodes of Q-Learning, something which would be nearly impossible without the exploration method.







25 episodes E = 0.1 50 episodes E = 0.1 100 episodes E = 0.1

Once again, we run the algorithm for more episodes and obtain policies much closer to the optimal:

350 episodes
E = 0.1

We have to keep in mind that the results obtained with Q-Learning were obtained much faster than the other algorithms (because computing an episode is much faster than iterating over all state values until convergence). Moreover, Q-Learning is, as mentioned, model-free. With these considerations, it is remarkable how well Q-Learning performed, and in such a short amount of time: despite displaying some non-optimalities, one can clearly see that the policy learned is quite good.

References

[0] MEHTA, Vivek. et Al. *Simulation and Animation of Reinforcement Learning Algorithms*. http://read.pudn.com/downloads93/sourcecode/book/365930/RL_sim/reports/vivek_report.pdf