A8 - Policy Evaluation

CS 4300
Fall 2016
Leland Stenquist - 1,3,5
Ryan Keepers - 2,4,6

1. Introduction

The purpose here is to introduce a new strategy for generating policies. This strategy focuses on fine tuning a randomly generated policy by finding better actions until that policy stops changing. When the policy stops changing on every update it is assumed that the policy is optimal. Some of the questions that are posed are:

1. Does Policy Evaluation create the same results as Value Iteration?
2. Is Policy Evaluation faster than Value Iteration?

2. Method

The algorithm for MDP policy iteration was implemented according to the pseudocode on page 657, figure 17.7 of the textbook.

CS6380_MDP_policy_iteration takes in as parameters, a state vector, action vector, probabilities for each state-action pair, the gamma hyperparameter, and maximum number of iterations. The policy choice begins off in a randomized state (each value in the vector set to a value in the set {1,2,3,4}). The program then iterates until it finds that the policy has settled and remains unchanged from the last iteration.

At the beginning of each iteration, the function updates a Utilities vector (which is initialized to zero). This is not as robust of an utility set as seen with MDP_value_iteration. Instead, it quickly estimates the likely utilities based on the current policy using the function U’ = gamma*(A*U)*R. Where R is the rewards vector, U is the utilities vector, and A is a square matrix with each row containing the vector from the probability array P (passed in from MDP policy iteration) at each state, where the action matches the current policy action.

After the Utilities are updated, the program checks for changes to the policy based on the new utilities information. If changes occur, the function will cycle through once again. If no changes are observed, the function returns the policy.

A hyper-parameter was updated to check on the extended accuracy of the final policy. While the original algorithm returned the first policy to make no changes between two iterations, the new parameter tells the algorithm to continue iteration until the nth time that no policy changes occur. The count does not need to be sequential, so if the parameter is set to 2, then the algorithm will return the policy on the second time it does not update throughout all iterations.

3. Verification of Program

Policy Evaluation was verified by comparing the policy generated to the policy given in the book examples. This was interesting since we found that Policy Evaluation, written exactly by the pseudocode given in the book, did not generate exactly similar results initially. We found that sometimes a policy would repeat even though the policy had not completely converged on the optimal policy yet. In order to fix this we created a hyper parameter that asked that the policy not stop updating until it repeated itself n times. When we did this we were able to get the exact same results from Policy Evaluation that we got in the book.

Reward = -0.04 | Gamma = 0.999999							
Book Policy				Policy Evaluation(with 10 unchanged policies)			
4	4	4	x	4	4	4	x
1	x	1	x	1	x	1	x
1	2	2	2	1	2	2	2
Reward =-1.7 | Gamma = 0.999999							
Book Policy				Policy Evaluation(with 10 unchanged policies)			
4	4	4	x	4	4	4	x
1	x	4	x	1	x	4	x
4	4	4	1	4	4	4	1

4. Data and Analysis

Results for textbook 3x4 board utilizing the multiple-unchanged-policies hyperparameter.
Reward = -0.04
Gamma = 0.999999

Number of unchanged policies before return: 1
Final Utility

0.6926	0.8477	0.9133	1.0
0.4850	0	0.6471	-1.0
0.1369	0.2916	0.4847	0.1664

Policy

4	4	4	x
1	x	1	x
1	4	1	2

Number of unchanged policies before return: 5
Final Utility

0.8056	0.8673	0.9177	1.0
0.7441	0	0.6599	-1.0
0.6498	0.5442	0.5699	0.3434

Policy

4	4	4	x
1	x	1	x
1	2	1	2

Number of unchanged policies before return: 10
Final Utility

0.8115	0.8678	0.9178	1.0
0.7614	0	0.6603	-1.0
0.7042	0.6523	0.6056	0.3771

Policy

4	4	4	x

1	x	1	x
1	2	2	2

Across various values of R we found a similar result to the above tables. Returning on the very first iteration provides a non-optimal, but still sensible, policy. However, it isn't until we allow the algorithm to continue iterations that it reaches a conclusion which matches the product of assignment 7's value iteration. It's possible to see the utility converge on the value iteration results as the hyper-parameter gets larger.

The premature returns for lower values of the hyperparameter, for example the initial implementation where the very first unchanged policy is returned, seems to be a result of the program hitting minor plateaus as it iterates on the utility values. We noted that when allowing for 10 unchanged iterations, the algorithm would encounter three or four plateaus where the policy would remain unchanged for two or three iterations and then continue with a series of changes before hitting another plateau.

Overall it seems most values of R reached the same policy as assignment 7 at about 6 or 7 passes through unchanged policies. However, the parameter might need to grow relative to the size of the board, since it seems that the driving factor for these plateaus is allowing the utility value changes to spread from the top right of the board (where the the success and death locations are found) to the opposite corner. Logically, the farther the distance between the goal/death locations and the farthest opposite tile, the greater the number of iterations required.

Wumpus Board Trials:

These boards were run with success = 1, and death = -1. Since this is scaled down from 1000, the standard reward for movement is -0.001 rather than -1.

Gamma = 0.999999
The left board breaks on the first unchanged iteration.
The right board utilizes the hyper-parameter to break on the 7th iteration.

R = -0.001

4	4	1	gold		4	4	1	gold
1	2	w	pit		1	2	w	pit
1	1	pit	3		1	2	pit	3
1	1	2	2		1	2	2	2

R = -0.075

4	4	4	gold		4	4	4	gold
1	1	w	pit		1	2	w	pit
1	1	pit	3		1	2	pit	3
1	1	2	2		1	2	2	2

R = -0.25

4	4	4	gold		4	4	4	gold
1	1	w	pit		1	1	w	pit
1	1	pit	1		1	1	pit	1
1	1	1	1		1	1	2	1

R = -0.5

4	4	4	gold		4	4	4	gold
1	1	w	pit		1	1	w	pit
4	4	pit	1		4	4	pit	1
4	1	1	1		1	1	1	1

R = -1.0

4	4	4	gold		4	4	4	gold
4	4	w	pit		4	4	w	pit
4	4	pit	1		4	4	pit	1
4	4	1	1		4	4	1	1

On the wumpus board, the hyper parameter seems to have a lot less functional influence than with the textbook board. It seems to have the most effect when the reward penalty is low, such as at -0.001 (the standard wumpus point setting), and even as high as -0.075. Bumping the hyper-parameter up to 7 cuts out most of the risky circumstances, such as taking the 10% chance of walking into a pit or wumpus when heading north along 2,2 and 2,3. Here or there some minor changes exist, such as not giving up at 1,3 when R = -0.25. But overall the policies look and act relatively similar.

5. Interpretation

A7's interpretation goes into great detail on how the agent responds differently to different reward values. Because of the similarities between A7 and A8 these results will be pretty much identical. The most interesting thing that we believe we found out about Policy Evaluation is the fact that the policy can have multiple iterations where it makes no updates. Yet the utilities are still converging so it may need those updates in order to find the optimal policy.

This was quite the bug for us for a while. We had to use our utility trace in order to really understand what was going on and then figure out why the algorithm was breaking out before it created an optimal policy. This seems to make Policy Evaluation a little less reliable since it may not return the optimal policy if written to follow the pseudocode exactly. The user of the algorithm should be aware of this potential bug. On the other hand, Value Iteration seemed to always find the optimal policy as long as it was given a good gamma value.

On running tests though it seemed that Policy Evaluation was a bit faster than Value iteration and in many circumstances speed trumps exactness. It seems that for these circumstances Policy Evaluation would be the way to go.

6. Critique

Policy iteration seems faster (though it's difficult to tell given the small size of the environment), but according to our data there's a trade-off between total accuracy and optimal policy. The policy that results isn't necessarily bad; on the contrary, in its basic format it very closely represents the policy discovered in assignment 7's value iteration, and the few changes created aren't significantly different from the optimum. If the complete optimization of the policy were a problem, I suppose it could come down to how much additional cost our hyper-parameter introduced.

We had an alternative hyperparameter idea which we ultimately did not attempt: rather than allowing the algorithm to continue iterating on its policies even after they plateaued, it could be possible to force larger utility changes with each iteration. For example, setting the utility policy iteration to update the utilities n times per policy iteration. Rapid change might force the policy to converge on its optimum without hitting the plateaus seen in the standard implementation.

7. Log

Ryan : 6 hours
Sections 1, 3, 5

Leland : 6 hours
Sections 2, 4, 6