Assignment 4: CS7641 - Machine Learning

Saad Khan

November 29, 2015

1 Introduction

The purpose of this assignment is to apply some of the techniques learned from reinforcement learning to make decisions i.e. to analyze work of an agent from a machine learning perspective. To be specific, the task is to explore Markov Decision Processes (MDPs) that are different from one another in the sense that first problem consists of large number of states (in this case more than a thousand) while the other has very few states (less than hundred). The analysis performed in this assignment is based on two planning algorithms and one learning algorithm. Details of these are in the following sections.

2 Implementation

The implementation of the MDPs along with the analysis was done using code written in Java that used the BURLAP Reinforcement Learning package. The three algorithms used were value iteration and policy iteration from the planning section of BURLAP along with Q-learning from the learning section. Computation times measurements all throughout this report are average of three runs for each algorithmic configuration.

3 Problems

The names I have given to the problems covered in this assignment are 'Maze Solving Problem' and the 'Sherpa Rescue Problem'. Following sections will briefly introduce these MDPs and analysis will come later.

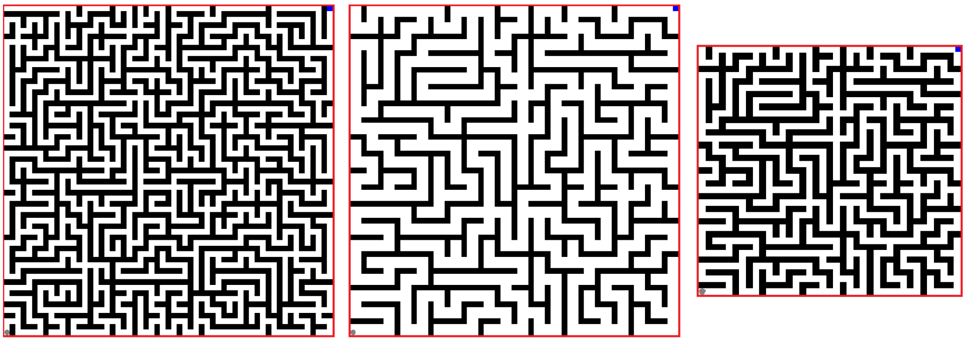
3.1 Maze Solving Problem

3.1.1 Introduction

This MDP is a simple maze solution implementation which can practically be applied to bots in computer games, specifically first person shooting (FPS) games. FPS bot artificial intelligence (AI) generally consists of optimal path-finding, picking up and using objects as clues to solve missions. For simplicity, we will only focus on path-finding aspect of these games in this assignment by making an agent find the solution to a maze and get to the end of a mission as soon as possible.

3.1.2 Dimensions

For this problem, to demonstrate the performance of planning and learning algorithms, I have considered a fairly large maze (Maze A) with dimensions of 59 x 59 cells and consisting of 1799 unique states. To analyze performance of the algorithms for different number of states, along with this primary maze (Maze A) I have used two other variants as shown in Figure 1. Maze B has the same dimensions as Maze A but has more wider maze paths and as result has 2398 states. On the other hand, Maze C has around 30% lesser states, at 799, but has the same path width as Maze A. Although, performance of these mazes cannot be compared directly but we will see how they fair against each other.



Maze A [59 x 59] , 1799 States

Maze B [59 x 59] , 2398 States

Maze C [39 X 39] , 799 States

Figure 1: [Maze] L to R: Maze A, Maze B and Maze C

3.1.3 Implementation

For each maze and the algorithm tested, the bot is initialized at [0,0] at the bottom left corner of the maze and termination condition is set at top right corner, i.e. [58, 58] for the two larger mazes and [38, 38] for the smaller one. The bot has to make decisions when it reaches an intersection, i.e. a point where it has a choice of going north, south, east or west (the four possible actions the robot might take). These directions are probabilistically set with the probability of success in the intended direction at 0.8 and 0.2 otherwise. GridWorldDomain, available in BURLAP, was used to set the domain as Map and a simple reward function was implemented with goal/termination state reward as 0 while a small negative reward of -0.1 for all other states. Setting up the reward function this way along with the terminal function of single goal (TFGoalCondition) encourages the bot to perform path-finding in a way so that it tries to reach the end of the maze as quickly as possible.

All of the above mazes were generated using the link : http://www.delorie.com/game-room/mazes/genmaze.cgi which was suggested by a class mate on piazza. The generated mazes were than manipulated in Excel (to convert to either 1s for walls) or 0s (for paths) before being passed to the BURLAP domain constructor.

3.2 Sherpa Rescue Problem

3.2.1 Introduction

Most deaths on Mt.Everest and other high peaks are attributed to avalanches, injuries from falls or ice collapse into crevasses. Search and rescue missions to find the casualties are sent but not all of the bodies are located and brought back down, in fact on these missions even the rescue team members have a greater probability of not returning back safely. These search and rescue teams mostly consist of local Tibetans/Nepalese known as Sherpa people. As an avid hiker and a mountain lover, I thought of a scenario where an agent(s) could replace the search and rescue teams for recovering the dead bodies from up there rather than sending human teams with greater risk of loosing lives again. Here an MDP problem can be thought of by considering one such robot which would perform this task. For simplicity, we will only focus on the search part of the team where an agent climbs up the mountain from the south eastern part up to the summit in search of casualties.

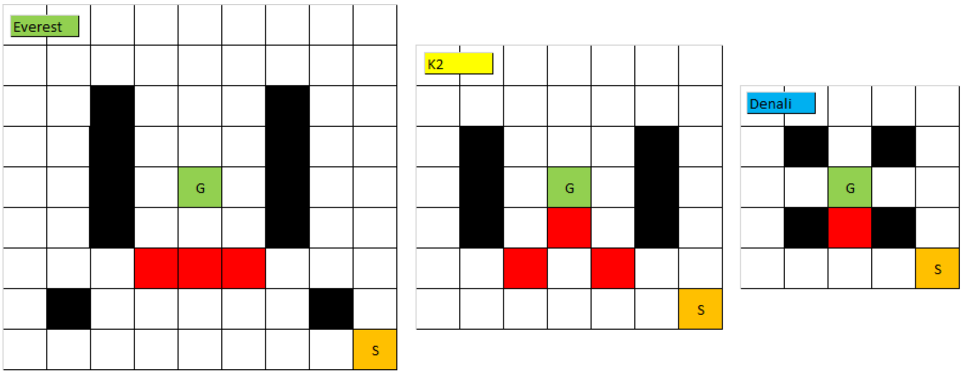


Figure 2: [Birds eye view] L to R: Everest, K2 and Denali

3.2.2 Dimensions

To make this a little interesting, I have tried to implement this as a simple grid seen from a birds eye view of a mountain with summit [G] in the center along with few crevasses [red cells] and rock faces [no go black cells]. The primary grid and its variants are shown in Figure 2. To explore different states, I have named them Everest (as the primary grid) with 9 x 9 grid and 71 states (goal state at [4, 4]), K2 with 7 x 7 grid and 43 states (goal state at [3, 3]) and Denali with 5 x 5 grid and 21 states (goal state at [2,2]). Again for simplicity, I have considered these peaks to be in 2D excluding effects of altitude variation.

3.2.3 Implementation

For this problem the bot will have a choice of going north, south, east or west to reach the summit. These moves are probabilistically set at 0.8 in the desired direction and 0.2 in any undesired direction. A reward function with goal reward of 0 is set up along with small negative reward of -0.04 for all other states except states which are marked red as crevasses. For these states, a harsh negative reward of -2 is set in order for the agent to avoid slipping into a crevasse. Goal is to safely search up till the summit as quickly as possible without falling into any crevasses.

All of these grids representing the three peaks were created manually and the crevasses [red cells] and rock face cliffs [black cells] were placed arbitrarily.

4 Algorithms

4.1 Value Iteration

Value iteration (sometimes called a 'backward function') is a planning algorithm that computes an optimal MDP policy in a reverse fashion as it starts from the 'goal' state and works backward to the initial state refining the estimate of the best policy and value. The process of propagating backwards is never ending so an arbitrary end point is chosen to stop the process when the algorithm has approximated well enough. The stopping thresholds that can be used are maximum no. of iterations (when the no. of iterations exceed this value the algorithms stops) and maximum delta threshold (when maximum change in the value function is less than this value the algorithm will terminate). In BURLAP it is implemented as ValueIteration and will be explored in later sections.

4.2 Policy Iteration

Policy iteration is another type of planning algorithm that starts with a policy and iteratively tries to improve it in order to converge. It tries to computes the optimal policy via a two step process. Step one is the inner value iteration which is performed first followed by step two which the policy iteration. The policy iteration is performed until it converges to the best possible solution and then this two step process is repeated until it completely converges. Here we can also set the maximum delta threshold for the inner value iterations which will be explored later. We will be using BURLAP implementation of policy iteration for our analysis.

4.3 Q-Learning (learning algorithm of choice)

Q-Learning is a reinforcement learning method which tries to find an optimal action/selection policy for an MDP. It starts by learning an action-value function that gives an expected utility by taking that action when in a particular state and following an optimal policy from that point onward. The main advantage of this algorithm is that it is able to make a comparison between the expected utilities for the available actions without even requiring a model of the environment. BURLAP has a function called QLearing which is used for this purpose in the analysis section.

5 Algorithmic Analysis

5.1 Value Iteration

For value iteration, I started off by setting the discount factor γ to the maximum value of 0.99 (so that the algorithm gives most importance to the future rewards) and set the max delta iteration threshold to 0.001 so that I could obtain the maximum number of iterations required in order for the algorithm to converge. The 'ValueFunctionVisualizerGUI' class in BURLAP came in very handy in order to identify the number of iterations required for convergence. The state value map that this class generates visually, shows the values for all the states at time step t. Setting up iteration intervals of 50 I ran the algorithm recording the back propagated state value for the initial state, i.e. the value of the initial state for 50 iterations then for 100 iterations going up till the point there was no further change in the initial state value indicating that the algorithm had converged.

This seemed to be tedious work so I went on exploring the BURLAP library and found an easier method to relate number of iterations required for convergence to computation time and max delta threshold. With some minor tweaking, in the ValueIteration.java class in BURLAP, I was able to record cumulative time along with change in max delta and perform a comparative analysis. I kept the max delta threshold on the primary vertical axis and for comparison kept computation time per iteration on secondary vertical axis.

5.1.1 Maze Problem

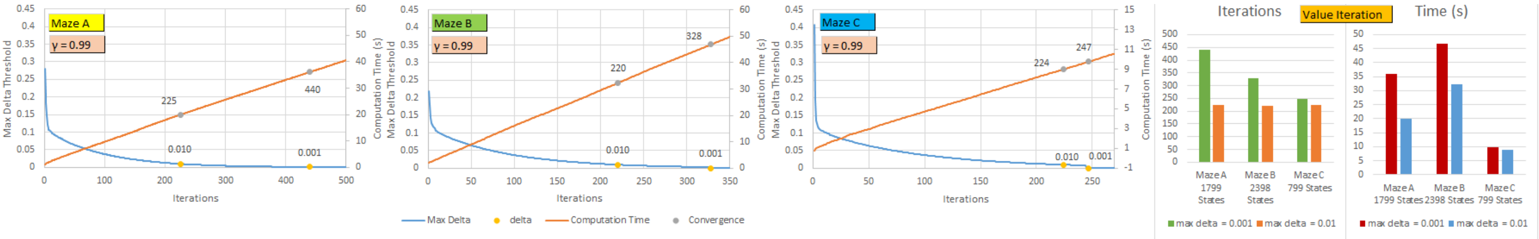


Figure 3: [Value Iteration - Maze Problem] LEFT : max delta and time w.r.t iterations, RIGHT : iterations and time comparison

For all three mazes, Figure 3 [LEFT] shows that the cumulative computation time increased linearly w.r.t no. of iterations while the max delta threshold decreased exponentially settling almost parallel to the x-axis for lower values of the max delta. The two points on each plot highlight the no. of iterations required to converge for a given max delta threshold for all 3 mazes, in this case I have chosen [0.01 and 0.001]. For Maze A and B the no. of iterations required for convergence seem far apart, in fact for a lower threshold of 0.001 the algorithm required more than hundred extra iterations to converge when compared to the higher threshold of 0.01. While for Maze C, there does not seem to be much of a difference in the number of iterations required to converge. I suppose this can be attributed to the smaller number of states for this maze.

For the two max delta threshold settings highlighted in the plots, the bar plots for iterations and computation time are shown in Figure 3 [RIGHT]. Maze B with larger states should have taken more iterations to converge but here it takes lesser iterations to converge compared to Maze A. This could be due to the larger path width which might have helped value iteration algorithm found an easier route to get to the optimal policy, but in order to do this it required much more time compared to the other 2 mazes as shown in the computation time bar plot. Looking at the time bar plot, it can be seen that it is consistent with the number of states for each maze, i.e. time required for value iteration to converge is showing direct proportionality to the number of states (more the states more is the time required to converge).

NOTE : For plotting these curves, I modified the value iteration class in BURLAP so that it did not break the loop when max delta was below 0.001. It kept on going until I interrupted it manually or 1000 iterations were reached.

5.1.2 Sherpa Problem

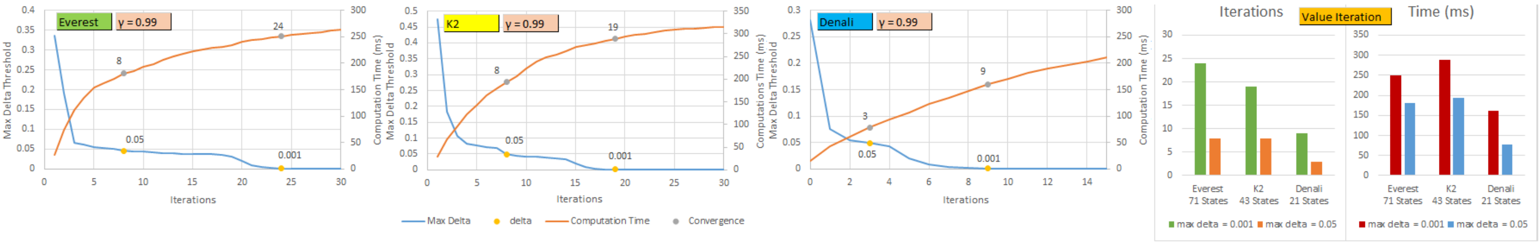


Figure 4: [Value Iteration - Sherpa Problem] LEFT : max delta and time w.r.t iterations, RIGHT : iterations and time comparison

A similar setup was applied to this problem as well and the comparative plots are shown in Figure 4 [LEFT]. Again, keeping γ constant at 0.99, I plotted the trend of changing max delta threshold on primary axis and variation in computation time on the secondary axis. For this smaller MDP problem, things happened pretty quickly and all value iteration algorithm converged well under 50 iterations.

Looking at the max delta variation, just like the maze problem here it also decreases exponentially w.r.t number of iterations. On the other hand, computation time variation is non linear, rather it is exponential where the algorithm takes more time to compute for higher max delta threshold and lesser time for lower values of max delta threshold. In contrast to the maze problem, I suppose this could be attributed to the open grid nature of the problem while for the maze problem the agent had restricted paths. For the two max delta thresholds highlighted with the scatter points, it can be seen that for Everest and K2 the number of iterations required are far apart (Everest : 8 and 24, K2 : 8 and 19) while for Denali in both cases the convergence is under ten iterations (is definitely because of the smaller size of the gird).

The bar plot shows iterations and computation time comparison in Figure 4 [RIGHT]. This clearly shows the trend that for higher peaks (larger grids) more time and iterations are required for the rescue bot to reach the summit and complete the mission (this is in conformity with the grid sizes).

5.1.3 Variation in iterations and time w.r.t. max delta

Now specifically focusing on the primary maze 'Maze A' and primary grid 'Everest', I went in greater detail to see the variation in no. of iterations and time w.r.t to fixed values of max delta to see if there was something interesting. Figure 5 shows that for both of these MDPs, for increasing values of max delta the number of iterations required to converge decreased exponentially and so did the computation time. This highlights that for larger max delta values algorithm breaks early and does not spend more time to converge to a better policy.

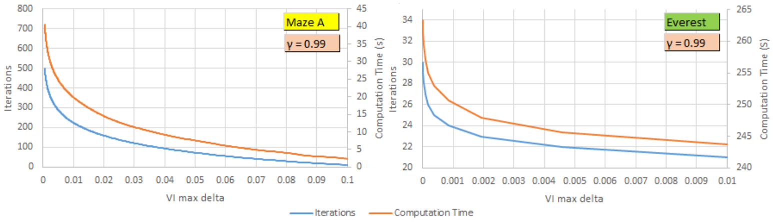


Figure 5: [Value Iteration - max delta variation] LEFT : Maze A, RIGHT : Everest

5.2 Policy Iteration

For policy iteration, I again made some minor modifications to the available PolicyIteration.java class in BURLAP to tap cumulative time for each policy iteration and to highlight computation time for inner value iterations. To further highlight variation in convergence I plotted computation time curves and inner VI bar plots for two different threshold settings of VI max delta [0.001, 0.01].

For the analysis I plotted bar graphs for inner value iterations on the primary axis and computation time variation on the secondary axis as shown in Figures 5 and 6. Again, I set the discount factor γ to the maximum value of 0.99 and ran the algorithm for 500 iterations for both the inner value iterations and policy iterations. The blue scatter points on the computation time curves show the number of policy iterations required to converge. The bar plots in right section of plots show the comparison of no. of iterations and time for the the plots generated pertaining to the two max delta thresholds.

5.2.1 Maze Problem

It can clearly be seen in Figure 6 [LEFT] that in case of the maze problem the policy iteration algorithm takes more inner value iterations in the beginning and lesser iterations when closer to converging. This trend can be seen for both max delta threshold variations explored here. The main difference is that when the max delta threshold is 0.001 the algorithm spends more time on inner value iterations in the beginning and converges in lesser number of policy iterations overall, while for higher max delta of 0.01 the algorithm spends very less time on the inner value iterations as it breaks the loop early and in return takes more policy iterations to converge but takes less overall time to converge. By looking at the curves, it seems that number of policy iterations required to converge for this problem are inversely proportional to VI max delta threshold setting.

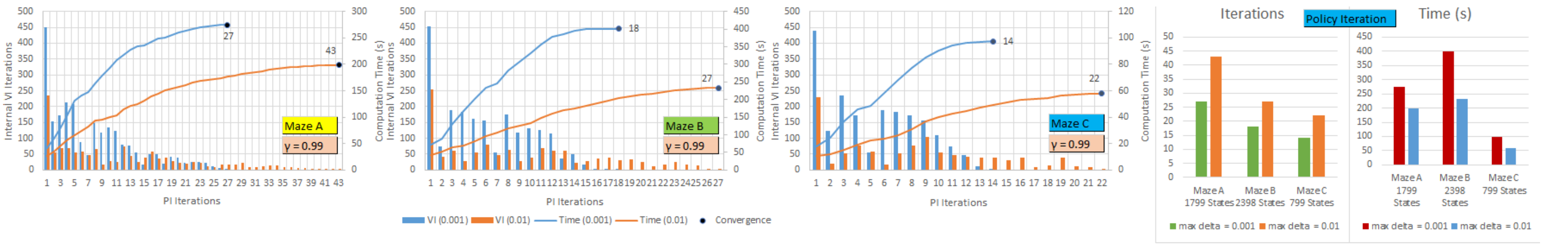


Figure 6: [Policy Iteration - Maze Problem] LEFT : Inner VI and time w.r.t iterations, RIGHT : iterations and time comparison

Figure 6 [RIGHT] shows the comparative bar plots for iterations and computation time for the two max delta setting used. Again similar to the results obtained after applying value iteration, we see that even though Maze A has less number of states it requires more policy iterations to converge compared to Maze B. On the other hand, time required for convergence of the policy iteration algorithm is consistent with the number of states for each maze showing that Maze B required the maximum time to converge.

5.2.2 Sherpa Problem

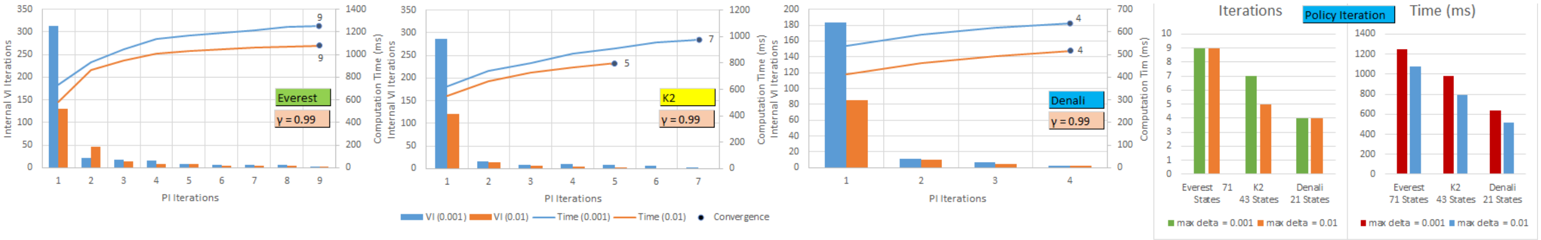


Figure 7: [Policy Iteration - Sherpa Problem] LEFT : Inner VI and time w.r.t iterations, RIGHT : iterations and time comparison

For the Sherpa problem, results were obtained in quick time as shown in Figure 7 [LEFT]. Generally, for both VI max delta threshold settings used [0.01, 0.001], maximum time was spent on the inner value iterations of the first policy iteration. This is consistent with all three grids and after that it takes very less time to compute rest of the iterations indicated by no significant change in computation times (almost flat gradient). Looking at these time curves it seems that changing the max delta threshold does not really effect the number of policy iterations required to converge in case of Everest and Denali, while for K2 (higher max delta value of 0.01) the algorithm requires 2 less policy iterations to converge. If we observe the time required for convergence for each grid, there is a difference of less than 200 ms when comparing the two max delta thresholds used which is also highlighted in the bar plots.

Figure 7 [RIGHT] shows the comparative bar plots for the two inner VI max delta thresholds explored. The general trend seems to be more the states, higher are the number of iterations it requires for the algorithm to converge.

Although, iterations to converge for Everest and Denali were the same for the two max delta values used but less computation time was required for convergence with the max delta value of 0.01, which was as expected. Apart from this, same analogy of policy iterations can be applied to computation times, i.e. larger the grid, more is the time required to converge.

5.2.3 Variation in iterations and time w.r.t. max delta

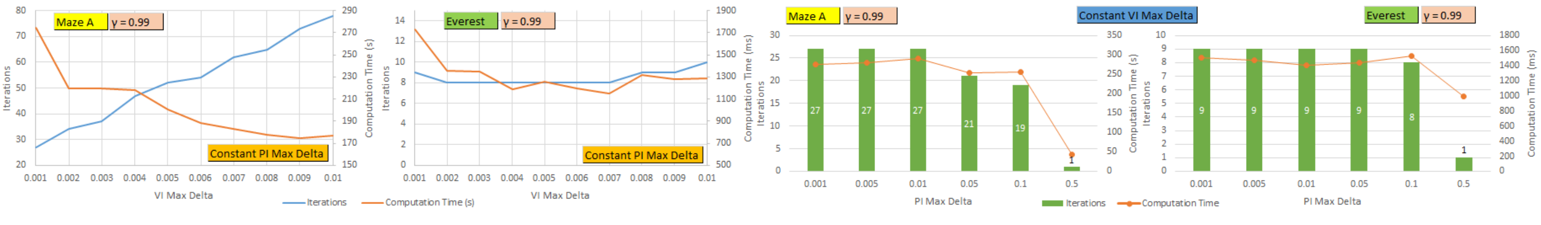


Figure 8: [Policy Iteration - max delta variation] LEFT : varying VI max delta, RIGHT : varying PI max delta

For greater insight, I again considered the primary maze 'Maze A' and grid 'Everest' and experimented with varying both PI and VI max delta values and noted number of policy iterations required to converge along with computation times as shown in Figure 8.

In Figure 8 [LEFT Section] for 'Maze A' the number of iterations required to converge increased with increasing VI max delta while on the contrary the computation time decreased. For Everest grid, it seems like number of iterations to converge and computation time did not really depend upon varying VI max delta which is accordance with the result obtained in Figure 7 [LEFT] for the 'Everest' grid (For these curves, PI max delta was constant at 0.001).

Figure 8[RIGHT Section] highlights the effects of varying PI max delta while keeping VI max delta constant at 0.001. It can clearly be seen that for smaller values of PI max delta, neither the the no. of iterations to converge nor the computation time is effected. However, when the PI max delta threshold is increased above 0.01 that is when we see reduction in no. or iterations to converge and computation time.

5.3 Q Learning

Having applied two planning algorithms to the problems now we move on to applying Q Learning to solves the tasks at hand. For analysis, I ran multiple episodes of Q Learning for the agent to interact with the environment and came up with the best possible solution.

I started off by considering primary maze 'Maze A' and primary grid 'Everest' and plotted two things (learning steps (primary axis) and computation time (secondary axis)) w.r.t. episodes required to learn as shown in Figure 9. By looking at these curves there did not seem to be a clear way to approximate the number of episodes required for learning so I tried a different technique.

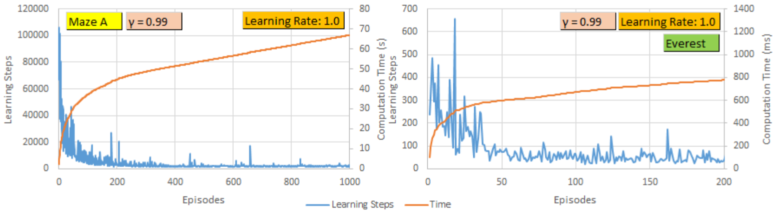


Figure 9: [Q Learning - episode and time comparison] LEFT : Maze A, RIGHT : Everest

After doing some experimentation in MS Excel, I came up with a much better method to average out the randomness of learning steps (for each grid maze and grid variant). I took the difference of max step value at time t_i and t_{i-1} and averaged it over three separate runs. The result was a pulsating graph as shown in Figures 10 and 11.

For performance evaluation, one of the parameter that can be tuned for the learning algorithm is the learning rate. For comparison purposes I used two values of learning rate [0.5 and 1.0] with plots generated showing step variation and computation w.r.t. to learning episodes as shown in Figures 10 and 11. I roughly marked points on the computation time curves for the no of episodes where I thought that the algorithm had learned enough and the computation time no longer increased drastically, i.e. a point after which the time only increases linearly.

5.3.1 Maze Problem

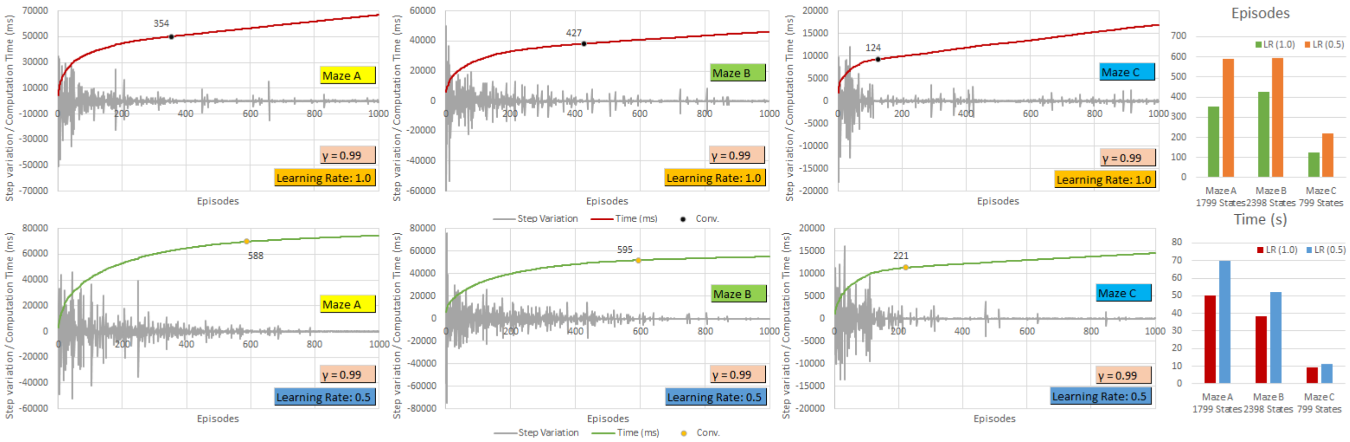


Figure 10: [Q Learning - Maze Problem] LEFT : learning steps and time w.r.t episodes, RIGHT : episodes and time comparison

Figure 10 [LEFT] shows the variation in average learning step and computation time w.r.t. to the number of training episodes for each maze variant. Running these scenarios of 2 different learning rates [0.5, 1.0] for thousand iterations it can be seen that the pulse diminishes after a while, i.e. the randomness of the learning steps smooths out indicating the algorithm has learned enough. For larger mazes, Maze A and Maze B this happens after at least three hundred iterations while for the smaller maze learning happens much more quickly. When comparing these pulse graphs and computation time curves for two different learning rates, we see that the algorithms spends much more time learning when learning rate (LR) is 0.5

Bar plots in Figure 10 [RIGHT] clearly show the comparison that for the two learning rates explored, the no. of episodes required for learning are much less for the higher learning rate of 1.0. Generally, for all maze variants, more time and episodes are required to learn in case of LR : 0.5.

5.3.2 Sherpa Problem

Same methodology was applied on to the second MDP problem and similar pulsating graphs were plotted as shown in Figure 11 [LEFT]. For the smaller no. of states attributed to the three grids variants I only focused on first two hundred episodes. It can clearly be seen that for the episodes where there is a lot of randomness in the learning steps, the computation time increases drastically. Then when most of the learning is done, the computation time curve starts to settle down and ultimately becomes linear. I have marked these points as the approximate no. of episodes required for learning. Just like the planning algorithms, episodes required to learn are well under hundred.

Bar plots for episodes and computation time in Figure 11 [RIGHT] show the trend for all three grid variants. It can be observed that, more the number of states, more is the computation time and episodes required to learn. Again, for the two learning rates tried, algorithm requires more time to learn in case of the lower LR of 0.5 which is understandable.

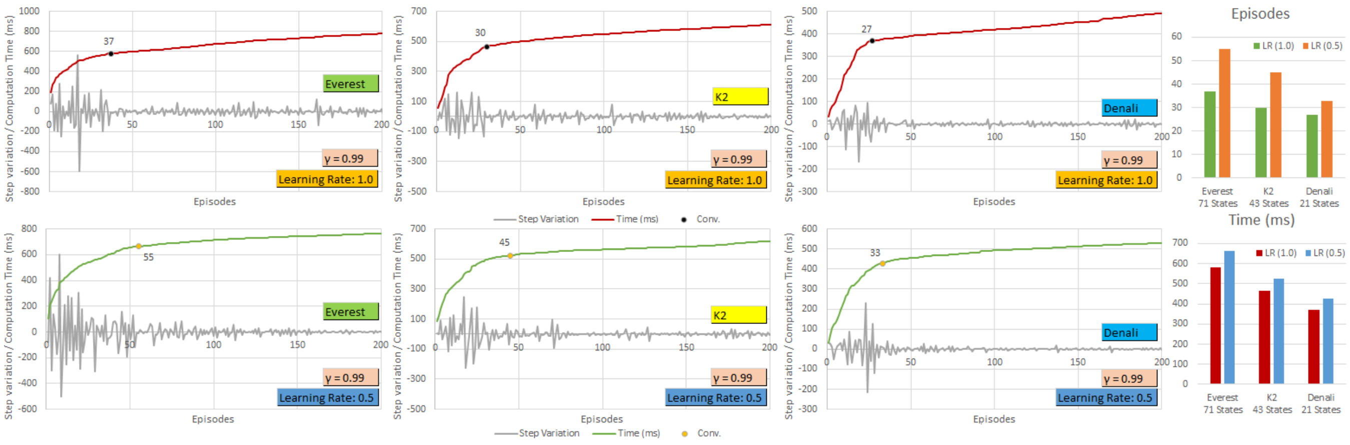


Figure 11: Q Learning - Sherpa Problem] LEFT : learning steps and time w.r.t episodes, RIGHT : episodes and time comparison

5.3.3 Variation in learning episodes and time w.r.t. learning rate and Epsilon

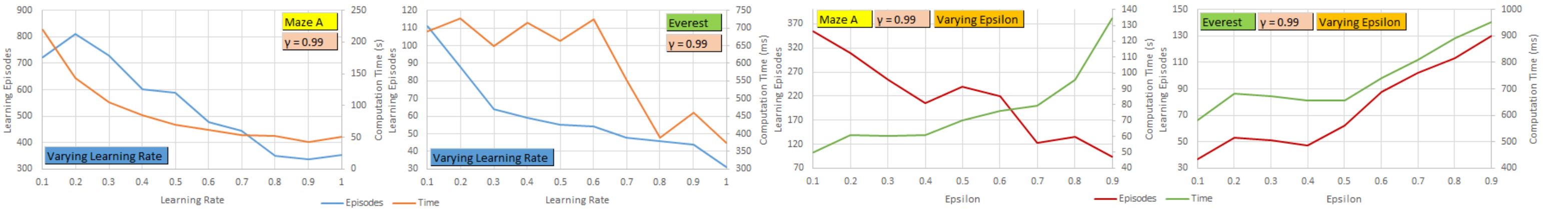


Figure 12: [Q Learning - learning rate and epsilon variation] LEFT : learning rate , RIGHT : epsilon

In this section I have applied the algorithm, with changing learning rate in one case and changing epsilon in another case, to the primary maze 'Maze A' and primary grid 'Everest' as shown in Figure 12.

In Figure 12 [LEFT Section], I have compared episodes required to learn (primary axis) and computation time required for learning (secondary axis) to the varying learning rate. It can clearly be seen that for both these cases, as the learning rate approaches 1.0, the algorithm becomes more confident and less and less episodes are needed to learn with least computation time required to do so.

In Figure 12 [RIGHT Section], I have highlighted the variation of greediness by plotting learning episodes and computation time w.r.t to changing epsilon. This gave contrasting results for the two problems. With 'Maze A', it was observed that for increasing epsilon, i.e. decreasing greediness the algorithm required lesser episodes to learn while sacrificing computation time. On the contrary, for the grid problem 'Everest', increasing epsilon resulted in increase for both computation time and no. of episodes required to learn.

6 Comparative Analysis

6.1 Discount Factor Comparison

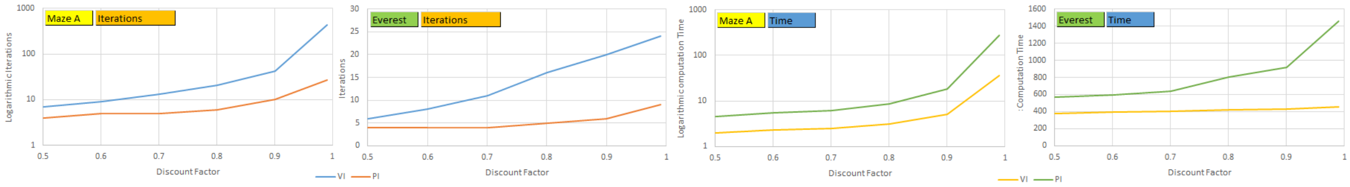


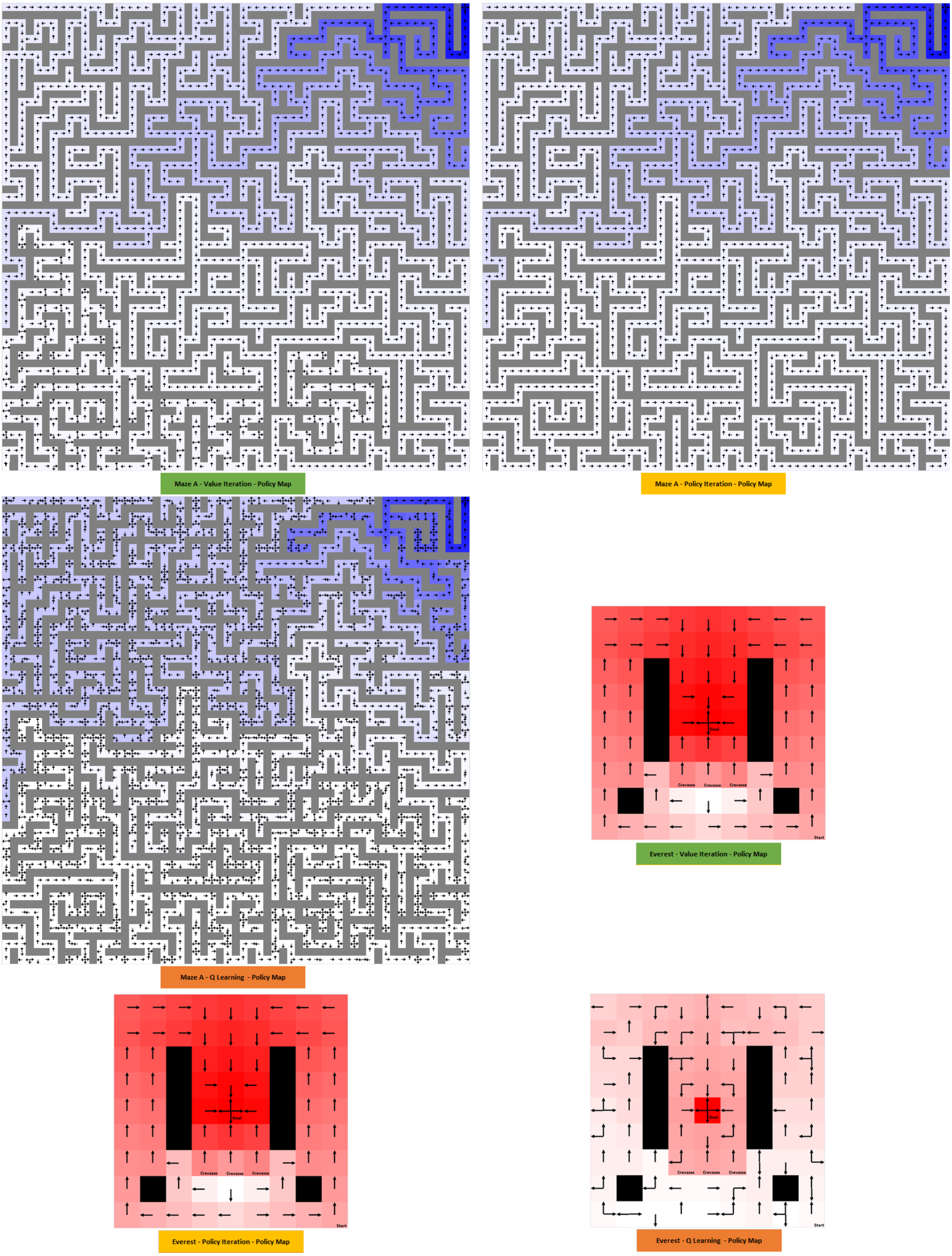
Figure 13: [Discount Factor Analysis] LEFT : Iterations, RIGHT : Computation Time

Figure 13 above shows the variation in no. of iterations to converge and computation time for the planning algorithms w.r.t. to changing discount factor γ. All this time during the analysis, the discount factor was kept constant and other parameters were tweaked. Here I have considered the primary maze 'Maze A' and primary grid 'Everest' and tried to identify changes, if any, caused by variation in discount factor.

Figure 13 [LEFT Section] shows the variation in no. or iterations required to converge for both MDPs. I have plotted logarithmic iterations for the maze problem as the no. or iterations for $\gamma = 0.99$ was very high (440). Although, both problems are of different nature (one has restricted paths and larger no. of states and the other has sort of open grid and smaller states) but still trend for the planning algorithms is pretty much the same. It can be seen that for varying discount factor, policy iteration algorithm requires orders of magnitude less no. or iterations to converge compared to value iteration. This can also be seen in the iteration plot for 'Everest' where log is no applied to scale the values.

Figure 13 [RIGHT Section] highlights the variation in computation time required for convergence. Again, for ease of representation I have scaled the computation time for 'Maze A' to log base 10. Here we see, that although no. of policy iterations required for convergence are way less but they require more time to execute.

6.2 Policy Comparison


Maze A - Value Iteration - Policy Map

Maze A - Policy Iteration - Policy Map

Maze A - Q Learning - Policy Map

Everest - Value Iteration - Policy Map

Everest - Policy Iteration - Policy Map

Everest - Q Learning - Policy Map

Figure 14: [Policy Analysis - Sherpa Problem] TOP : Maze Problem, BOTTOM : Sherpa Problem

Figure 14 highlights the policy maps for the two MDP problems considering the primary maze 'Maze A' and primary grid 'Everest'. For each algorithm, these maps were obtained with keeping γ constant at 0.99 and max delta threshold at 0.001.

6.2.1 Maze Problem

The top part of Figure 14 shows the optimal policy maps for value iteration and policy iteration when applied to the primary maze 'Maze A'. It can clearly be seen that value iteration algorithm does not converge to the same solution as the policy iteration algorithm. Although, the policy maps for both these algorithms are very similar in the upper part of the maze (closer to the end of the maze) but are very dissimilar in the bottom right hand side. Value iteration algorithm seems to favor either direction in most intersections while policy iteration is confined to a fixed path.

Second row of Figure 14 shows policy map if Q Learning was also considered for planning after thousand iterations. Here we can see that Q Learning fairs poorly to identify a specific solution to the problem with a lot of intersections that favor either direction. There might be a better combination of parameters that can be tuned to obtain a better solution using Q Learning but keeping discount factor at 0.99 and max delta constant at 0.001, this algorithms does not seem to provide a good solution.

Overall, it can be said that for 'Maze A', policy iteration algorithm seems to provide a better solution considering the nature of the problem.

6.2.2 Sherpa Problem

Second and third rows in Figure 14 highlight the policy maps for the Sherpa problem. First focusing on the policy maps for value iteration and policy iteration it can be seen that both these algorithms converge to almost identical solutions with just one state difference (arrow pointing in opposite direction) in the bottom center of the grid.

Looking at the policy map for applying Q Learning we see that again for this problem also, Q Learning does not provide as good a solution that the planning algorithms provided. This can be seen here that some of the states on the policy map favor multiple directions for next states which is not the case for the planning algorithms.

By observing the results of the policy maps, both policy and value iteration algorithm work equally well for this grid problem.

6.3 Convergence Comparison

Although, episodes required to learn in case of a particular problem cannot be compared to the no. of iterations required to converge but I have tried to highlight the differences in the best way possible. Bar graphs in Figure 15 and 16 compare the iterations to converge and episodes to learn on one side and compare computation time on the other side.

6.3.1 Maze Problem

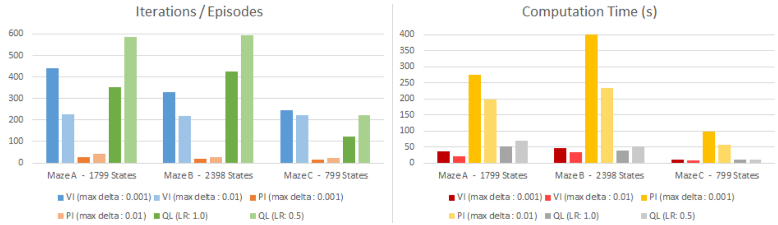


Figure 15: [Convergence Analysis - Maze Problem] LEFT : Iterations / Episodes, RIGHT : Computation Time

Figure 15 [LEFT] shows the iterations/episodes for the three algorithms when applied to the three variants of the maze problem. I can clearly be seen that policy iteration performs the best with least no. of iterations required to converge.

Figure 15 [RIGHT] shows computation time comparison, highlighting clearly that fastest algorithm is value iteration with Q Learning finishing in close second, while policy iteration takes almost 6 to 7 times more time to compute.

Having looked at policy maps above, iterations to converge and computation time the best algorithm for this problem is policy iteration as it has a better policy map and required lesser no. of iterations to converge.

6.3.2 Sherpa Problem

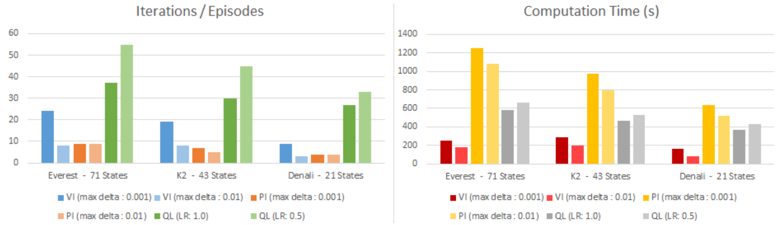


Figure 16: [Convergence Analysis - Sherpa Problem] LEFT : Iterations / Episodes, RIGHT : Computation Time

Figure 16 [LEFT] shows the iterations and episodes for the grid problem. It can clearly be seen that both planning algorithms perform better than the learning algorithm.

Figure 16 [RIGHT] we see that again, value iteration is the fastest algorithm in terms of computation time followed by Q Learning and policy iteration.

For this problem, we saw that both policy and value iteration provided better policy map solutions and with value iteration converging faster compared to policy iteration, I would pick it over value iteration over policy iteration.

7 Conclusion

Overall this was an interesting project in the sense to actually solve the Maze and the Sherpa problem. It was a little hard to deal with BURLAP in the beginning but as the time progressed so did my confidence to manipulate classes in BURLAP.

Overall things turned out as expected after studying in depth and applying these algorithms in greater detail. The interesting thing for me was how slow policy iteration was in the Maze problem. That could be due to the large no. of states of that problem.