Machine Learning Engineer Nanodegree
Capstone Proposal

Robert M Salom
January 20, 2017

Project Overview:

The overall goal of this project is to explore reinforcement learning as applied to the "simple" game of ticktacktoe The Nanodegree lectures introduced Q-learning as a sub-domain of reinforcement learning which culminated in the SmartCab project. In the end, Smartcab tackled a problem with a relatively small state space. There were primarily two main cases the agent had to handle. Making the correct choice when turning left on a green light, and making the correct choice when making a right turn on a red light. Additionally it would be relatively easy to just program the agent to do the right thing without appealing to a sophisticated method like Q-Learning. With ticktacktoe, which we will consider here in a more general nxn version where n is the length of a row, there is the opportunity to apply reinforcement learning on larger more complicated state space; that is, the set of all possible game sequences. It is also a problem where directly programing ideal behavior proved to be challenging, at least for me, and so Q-learning is a viable alternative approach for a problem of this type. On this point however, ticktacktoe can be solved with explicitly programmed routines. As pointed out by the proposal reviewer, the general purpose *minimax with alpha beta pruning* algorithm can play ticktacktoe perfectly at the expense of a lot of computational power. Two methods for explicitly playing ticktacktoe are are also explored and used as performance references for the reinforcement learning guided play; the previously mentioned alpha-beta algorithm and a naive solution that turned out to be really fast, with some of the same minimax spirit.
NOTE: LibreOffice word processor corrects tictactoe to ticktacktoe, Google says its tictactoe. Since I don't want to struggle with the word processor I'll use it's auto-complete form of ticktacktoe :|

Problem Statement:

The initial hope was to have two learning agents play ticktacktoe and accumulate experience iteratively in a global Q until their performance approximates that of a human player. That is;

To implement reinforcement learning so that a player agents can refer to a Q function to decide where their next mark on an NxN ticktacktoe board will go.

That being the main problem, a secondary problem that is important to me is to avoid as much as possible, even completely, the use of the explicitly programed perfect players as reinforcement devices. My curiosity is to see how well the agents can learn purely on their own with just the game's end state as feedback. I feel this will also give me insight on how to deal with a situation where I have no clear explicit programmed solution. The player's move, dictated by Q, should ideally approximate an actual optimal move. To facilitate my descriptions here I will define things in terms of their actual python code implementation.

- The "size" of the game is the length of a row of the game board so classic ticktacktoe is of size 3.
- A "mark" is the symbol an agent uses to mark the board for example the capital letters 'X' and 'O'

- A “move” is a tuple consisting of and index followed by a mark i.e, (4, ‘X’) or (18, ‘O’)
- An “index” is the position taken by a move on the ticktacktoe board. If the game is of size 3 then indices range from 0 to 8. In general, the indices range between 0 and size^2 – 1
- A move “sequence” is a list or tuple consisting of moves alternating in mark in the order they were made by the players. For example the first three moves of a game expressed as a sequence [(8 ,’X’) , (6, ‘O’) , (2, ‘X’)] or alternatively ((8 ,’X’) , (6, ‘O’) , (2, ‘X’)) . The first move is at the python list index of 0. Sequences in tuple form are the actual keys to the Q function.
- The “Q function” is a python dictionary with keys in the set of all move sequences for a particular size of board and values in the real numbers. The idea is that a size of game is chosen and then a Q-function is populated with keys and values.

The ticktacktoe game is usually thought of as existing on a grid with 3 rows and 3 columns, although here we will think it as n rows and n columns. There are also two diagonals which will be referred to as the positive diagonal “D-pos”, who’s start point is at game index 0 and end point is at game index n^2 -1, and a negative diagonal “D-neg” with start index n-1 and end index n^2 – n. If any of these rows, columns, or diagonals has marks belonging to a single player it will be referred to as a “line”.

- A “line” will be represented by a python list who’s first entry is the mark of the player it belongs to, followed by the indices belonging to the given horizontal row, vertical column or diagonal.

For example in the game below, column 0 and row 0 form lines denoted as;
[‘X’, 0 , 8] and [‘X’ , 0 , 2, 3] respectively.

X		X	X
	O		
X		O	
		X	

Row 1 is also a line denoted as [‘O’, 5]. The “length” of the line is the number of marks belonging to it. In the example above the line in row 0 has length 3, the line in column 0 has length 2 and the line in row 1 has length 1.

Metrics:

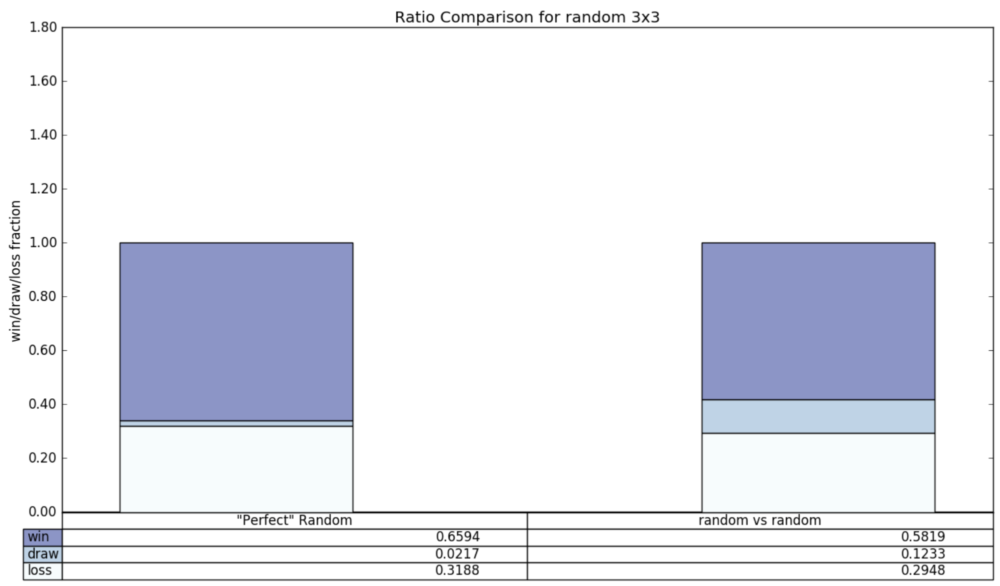
The two ticktacktoe playing agents can play according to four main playing policies. *Random*, *minimax* , *ideal,* and *reinforcement*. Performance is measured within the context of dueling policies. For the 3x3 case an explicit tally of all strategically relevant final game positions (https://en.wikipedia.org/wiki/Tic-tac-toe) will also be used. In other words a count of game positions that excludes geometrically equivalent final games. There are 91 ways player 1 can win, 44 ways player 2 can win, and 3 draws. Here a perfect win:draw:loss is readily available for comparison. For games of higher dimension, in the absence of explicit game position counts, the idea is to appeal to metrics of a more statistical nature. First of all, by playing Random vs Random for a large number of games, a rough (the reason why I say rough will be discussed later) baseline of the proportions of win/draws/loss will begin to appear. Armed with that and with the fact that a necessary condition for a perfect player is that it does not lose, one can at least have a gauge of whether a policy is essentially random, near perfect, or perfect. For example in a 3x3 game for a perfect player 1, it’s win/draw/loss count should be approximating 91 : 3 : 0, and 44 : 3 : 0 for player 2. So for games larger than 3 after using

Random vs Random to get an idea of the games global win : draw : loss, any reduction in the proportion of loss and subsequent allocation to win : draw is a measure of improvement in performance. As it turned out a lot of this reasoning was somewhat upended, but I still have to describe the metrics in this section.

Data Exploration/Visualization:

So to get an idea of what the space of end game sequences looks like under the above mentioned metric I started by comparing the ratios of actual game counts for a 3x3 game with the counts of 10000 games with agents choosing their next move randomly (but legally) using python's random.randint() function. That is, the proportions 91/138 : 3/138 : 44/138 were compared to (total # player 1 wins)/10000 : (total # draws)10000 : (total # player 2 wins)/10000. With the results seen on the figure below.

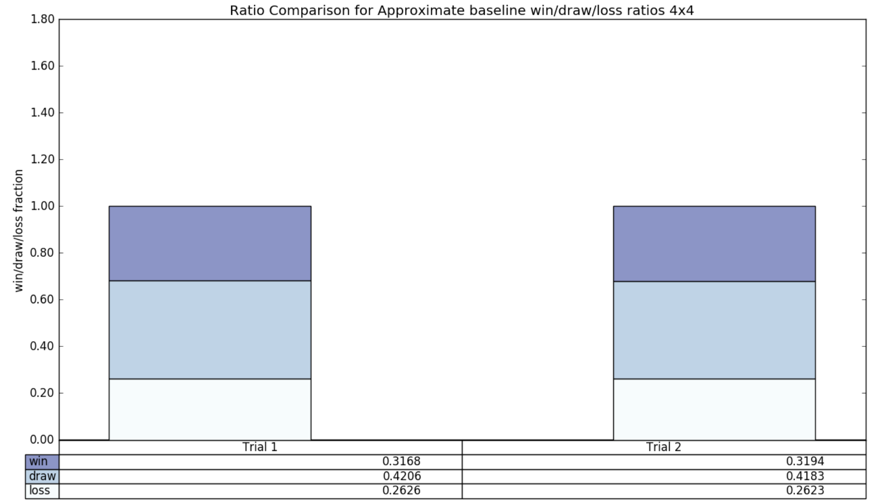
Figure 1



	"Perfect" Random	random vs random
win	0.6594	0.5819
draw	0.0217	0.1233
loss	0.3188	0.2948

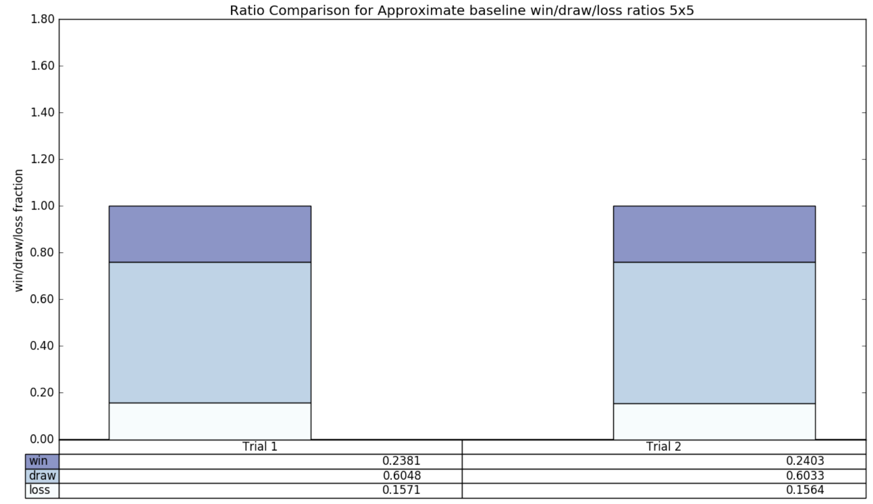
So the main thing to notice here is that the ratios acquired through simulation have an enlarged draw proportion. The win proportion is ~12% smaller than that of its counted counter part and the loss proportion is ~8% smaller. The draw proportion is almost 6 times bigger. So it jumps from being 2% of the total to around 14%. So a random policy does have a tendency towards draw. The above described metric is somewhat rough, but very practical, especially when used to measure increases in wins and reduction in losses, since two regions are not that far from the actually counted ratios. Draw's error is larger because it has been bled into by both win and loss. Confidence in this reasoning is somewhat shaken when the following trend is observed when the same process is applied to size 4, 5 and 6 games with figures below.

Ratio Comparison for Approximate baseline win/draw/loss ratios 4x4



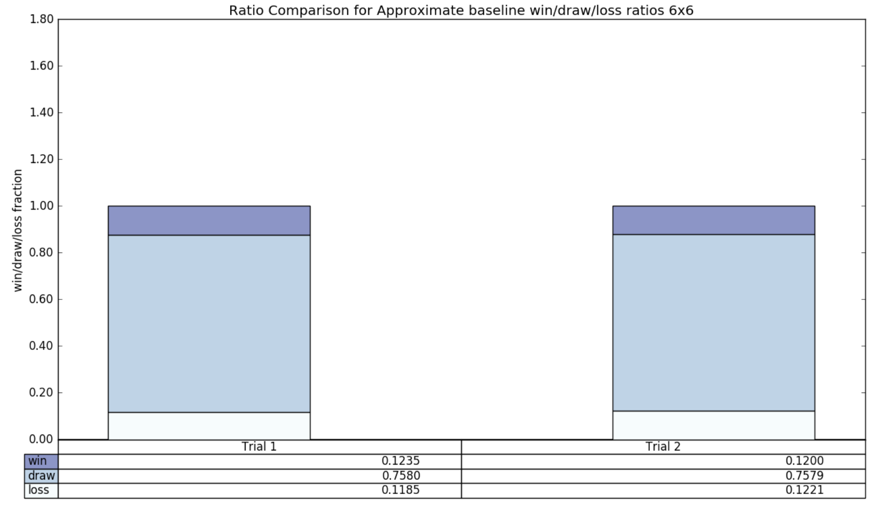
	Trial 1	Trial 2
win	0.3168	0.3194
draw	0.4206	0.4183
loss	0.2626	0.2623

Ratio Comparison for Approximate baseline win/draw/loss ratios 5x5



	Trial 1	Trial 2
win	0.2381	0.2403
draw	0.6048	0.6033
loss	0.1571	0.1564

Ratio Comparison for Approximate baseline win/draw/loss ratios 6x6



	Trial 1	Trial 2
win	0.1235	0.1200
draw	0.7580	0.7579
loss	0.1185	0.1221

In the above 3 figures two trials of 10000 random vs random games were performed for games of sizes 4,5,6. It is clear that an increase in the size of the game is leading to growth of the proportion of draws. So is this because of the simulated random gameplay's tendency towards draws or is it a real thing? That is, are these "sampled" ratios any good as rough benchmarks for nxn games? I think it is an actual trend, which shows that for larger n, a game is more likely to end in draw. Alternatively as n increases there are more ways to draw a game than to win it. The reason I think this is true, is because in order to win, one must fill a row, a column, or a diagonal. For arbitrary n, there will always be n rows, n columns, plus two diagonals. So 2n+2 "shots" at winning the game. To eliminate any one of these, you only need two opposing marks to be placed in line. For a given line, there are n places to place the first mark, and n-1 places to place the second, and then roughly times 2 because either player could have made the first mark (except for the very start move which belongs to player 1) so ~2n*(n-1) ways to "kill" a line. Though I am ignoring the count of ways you can get to a win or draw position (which I don't think would favor the wins), roughly there are 2n+2 winning lines and at least 2n*(n-1) * (2n+2) ways to kill them. So it looks like winning grows somewhat linearly and drawing grows cubicly. For all that follows however I will be restricting analysis to mostly the case of size 3 and 4, basically because of computational constraints.

Algorithms and Techniques:

Apart from the above sampled win/draw/loss ratios as benchmarks for comparison, two explicit algorithms have been implemented that are at least close to being perfect players. The first which I call "ideal" (which was a somewhat tortuous reinvention of the wheel on my part) basically seeks to generalize roughly the common sense game play we use as humans on a 3x3 board to the nxn case. The idea used here was to measure the state of the game in terms of the quantity of "lines" (as defined previously) each player owns, and what their "length" is (or how filled they are). More precisely if player has made lines, *l_1, l_2, l_3 … l_k* . Each of length *m_1, m_2, m_3 … m_k.* Then the measure of the game for said player is given by;

M = *l_1 * m_1 + l_2 * m_2 + l_3 * m_3 … + l_k * m_k*

The next move made by a player is in order of priority. First;
Win/Block:

- If the player can win by making a line of length size take it
- If the opponent will win on the next move, block it.

Secondly, after a certain point, more specifically when the game has reached n + (n-1) – 3 + n+(n-1) – 3 – 1 = 4n -9 steps, it will be the first time a player can generate a fork. The reasoning for the above count goes as follows; two intersecting full lines have n+(n-1) marks on them (because they share a mark). To make them into a fork, take two marks (not the intersection though) away from them. Thats a fork. And in order to make them into a "pre-fork", or the state before a fork can be made, take one more away; hence n + (n-1) -3 = 2n-4.
Ex.

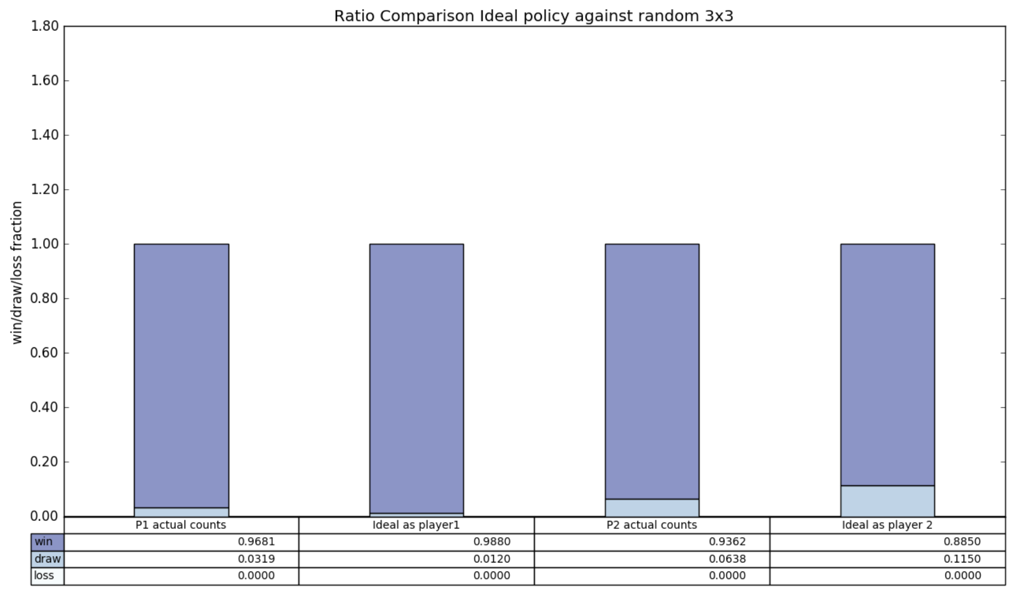
O O O O O O O	O O O O O	O O O O

Since if player 1 has made m moves, it means player 2 has made m-1 moves, then the number of steps taken at the first "pre-fork" is 2n-4 + 2n-4 -1 = 4n-9. At this point the ideal policy will;
Fork/Block:

- Look if there is a fork possible by *either* player in the next move
- If its the current player's fork, then make it
- If its the opponent's then block it. Ideal policy will now also look for lines of size n-1 which are one move away from a win.

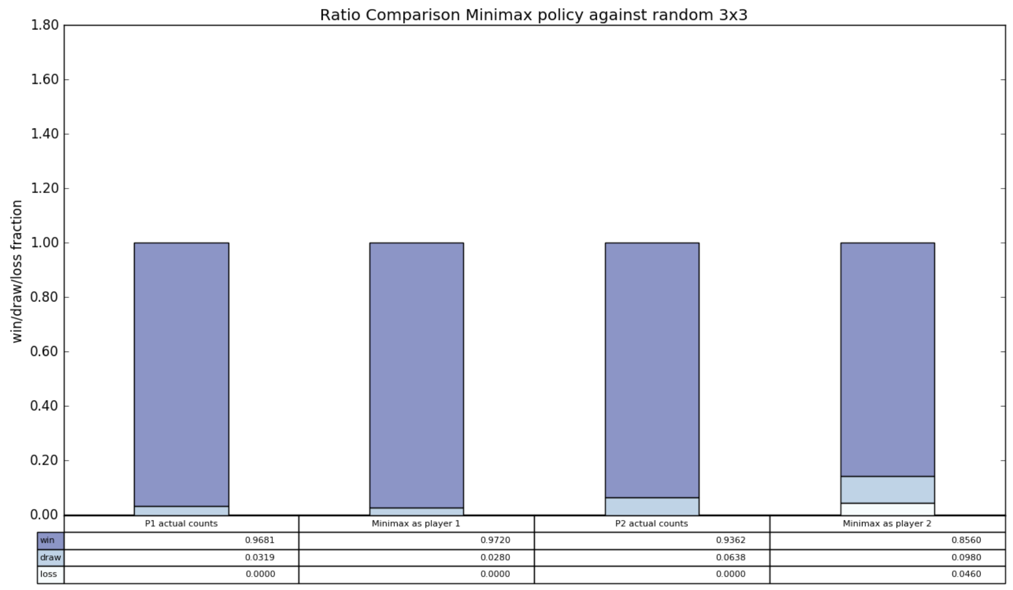
If neither of the above scenarios are in play, the next move for a given player is either the one that will result in the greatest measure, M, for himself, or, in the case that there is a move that gives his opponent a greater measure than what he would gain for himself with his best move, to block it by taking his opponent's move of highest measure.

The second explicitly programmed strategy is the minimax alpha-beta pruning algorithm. The source material for the the implementation used in this project comes entirely from https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm, which seem to be lecture notes for a Cornell CS class. As I only understand in a novice way, that source provides a much better step by step explanation than what I could. What is left unsaid in the previously mentioned notes, are key aspects of the minimax algorithm. Since it is a general purpose algorithm there are details that can only be addressed in the specific scenario it is being tailored too, some are small, like the determination of a leaf node or a child node, but a glaring one is the implementation of the "evaluate" function, which in our case I call minimaxMeasure. Minimax is a recursive algorithm that is described in the context of a tree structure, which in our specific case, the root of which, is player 1's first move. From there every possible next move by player 2 starts a branch rooted there. Player 1's next move causes each of the branches created by P2 to branch again, and the process is repeated until the tree structure ends at points called leaves; basically in our context, corresponding to moves that lead to a win, draw, or a loss. Paths from the root to any node in the tree basically correspond in our case to what was described in the problem definition section as "sequences". Minimax explores future moves up to a specified depth and assumes that each node can be evaluated or "measured" in some way. It also assumes that there is the desire to alternate between searching for "high" values and "low" values in consecutive steps down the tree. Luckily it was a matter of refurbishing the Ideal's measure function to provide minimax with its "evaluate" function. Basically, at each "node" (i.e. each game sequence) M1 (as defined for Ideal) is taken for player 1, M2 is taken for player2 and the minimaxMeasure is defined as M1 – M2. If the sequence corresponds to a player 1 win a much bigger reward of (10*size)^4 is assigned, or the negative of that for a player 2 win. 0 is the measure for a tie. Interestingly that was my first attempt at the measure function for Ideal but I couldn't get it to work. It seems to work reasonably well with minimax. Now, the Ideal policy, because of its aggressive blocking *should* never lose and, after much debugging, the data bare that out. It possess at least that property of a perfect player. What follows is a comparison of Ideal and minimax with "perfect" player 1 and player 2 ratios of 91 : 3 : 0 and 44: 3 :0 on a 3x3 board. First;



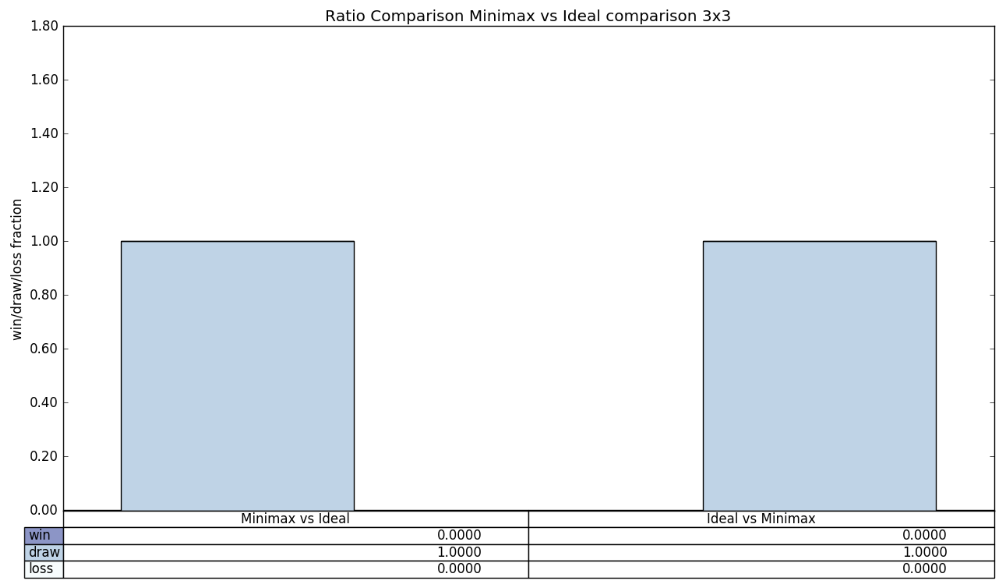
	P1 actual counts	Ideal as player1	P2 actual counts	Ideal as player 2
win	0.9681	0.9880	0.9362	0.8850
draw	0.0319	0.0120	0.0638	0.1150
loss	0.0000	0.0000	0.0000	0.0000

For the figure above we see that Ideal deviates somewhat from what we expect a perfect player to do. It tends to win more as player 1, ~2% increase and draw more as player 2 ~6% increase. The fact that random tends to overpower a player 2 is a recurrent theme. Importantly, however out of 1000 trials there are never loses, up to size 13 or so (reducing the trial number) that is always the observation. Similar analysis of minimax is not as clean because due probably to inexpert implementation (i.e. not the best evaluate function) it is slow and hence most of the time it is used with a search depth of only 2. Which should be enough to spot most forks, however it leaves it open to a small number of loses when playing the disadvantaged player 2 position. As can be seen here;



	P1 actual counts	Minimax as player 1	P2 actual counts	Minimax as player 2
win	0.9681	0.9720	0.9362	0.8560
draw	0.0319	0.0280	0.0638	0.0980
loss	0.0000	0.0000	0.0000	0.0460

Remarkably minimax's ratio as player 1 is virtually indistinguishable from the "perfect" player. Additionally, after tracing back several of the specific games it lost as player 2 and setting higher search depths it would reverse the loss to a win many times, and at least draw. I feel a better implementation of minimax or more computational power would reveal ratios exactly matching the ones derived from an explicit game count. Minimax's graph above is a result of 500 trials. To further confirm Minimax's and Ideal's near perfect behavior here are three trials of Minimax vs Ideal. Minimax is operating with a depth setting of 5 to dispense any doubts of under performance. In reality only one trial is necessary as, when traced, they play the same game every time because they are pursuing optimal strategies.



	Minimax vs Ideal	Ideal vs Minimax
win	0.0000	0.0000
draw	1.0000	1.0000
loss	0.0000	0.0000

As expected, they will always draw, regardless of being player 1 or 2. Finally a detailed explanation of the approach to reinforcement learning used here is overdue. I hesitate whether to call it Q-learning or not sometimes, because since the context is not that similar to what we saw in the Smartcab, the approach here differs enough that I am not sure if it still falls under that label or instead under some other type of reinforcement learning.

Methodology:

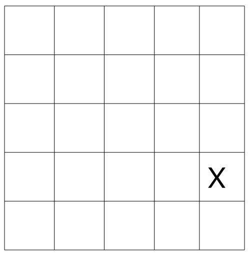
Code structure:

Before describing the details of the reinforcement learning implementation, I will briefly discuss the over all structure of the code base. It is divided into 4 python modules. The first, *game_state*, houses a GameState class, a QMap class, and bare-bones Player class. QMap houses the Q dictionary and methods that operate on it. Then there is *play* module containing the main method, along with multiple support functions to play and analyze games between player agents. It also contains the LearningPlayer class that extends Player, a similar layout to the smartcab. Third there is *strateegery* module, housing the Strateegery class who's methods decide player moves. Three have already been discussed, *Ideal*, *minimax,* and *random*. Finally the GameBoard class is supposed to add a graphical pygame face and user interactivity but it remains unfinished. To run a game between two player agents usually a

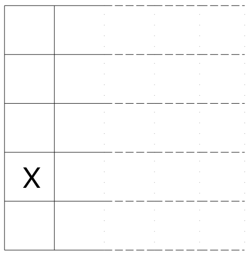
GameState instance is created of a certain size, a QMap object is created and set to the GameState, followed by two LearningPlayer agent objects who themselves take the GameState object as a parameter in their initialization. So that in the end a player has access to the GameState, and the GameState has access to the players.

Data Preprocessing:

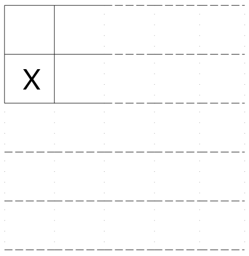
A first difficulty encountered when exploring possibilities for the application of reinforcement learning is described in the wikipedia article, https://en.wikipedia.org/wiki/Tic-tac-toe. Here, a naive count of the number of games in 3x3 ticktacktoe is 9! or 362,880. This follows in a straight forward way from the sequence representation described in the problem statement section. So for a nxn game a naive count would yeild n! possible games. However, again as mentioned in the article, the symmetries of the square make many of those games equivalent. For example starting on any of the corners is strategically equivalent. In order to reduce the number of keys that will exist in Q, I implemented a coordinate transform that uses the first move made to transform the game into a “standard” position of starting moves in an upper triangular region. This is accomplished by the use of at most 3 reflections about; a vertical symmetry line, a horizontal symmetry line and a diagonal symmetry line. Take for example this starting move;



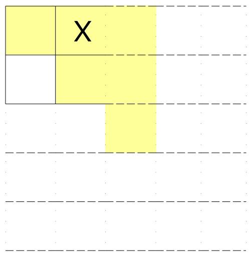
A reflection about the vertical symmetry line results in;



And about the horizontal symmetry line;



Finally about positive diagonal "D-pos";



So that start_index = 19 is equivalent to standard_index = 1. There are sequences that are equivalent if the moves coil around the triangular region especially as N gets larger and the region grows, but still this is a big reduction in the state space and for N = 3 it gives a minimal state space.

Reinforcement learning:

In this context, Q (a variable inside the QMap instance) is updated at the conclusion of a game. This is in contrast to after every step, as in the smart cab project. Now, since the intention is for Q to be used by a player agent to guide its next move, a reward, say r, is added to every subsequence of the end-game sequence that concluded the game. Since those subsequences (including the entire game sequence) are keys in Q, each explored sequence will have a real number value that will be used to guide subsequent player moves. Now, I experimented with different ways of performing the update, as well as different magnitudes for the reward, r. Regardless, if player 1 is the winner at the end of the game the sign of *r* is positive, if player 2 is the winner the sign of *r* is negative. If the game ended in a tie, essentially no update is performed because *r* is 0. For now I have settled for r=1 for a player 1 win and r = -1 for a player 2 win. Q-learning as seen in class was, seeking to implement a version of the Bellman equation, by approximating it with a recursive function of the form;

Q[previous_action_state] = reward + λ * Q[current_action_state]

where Q[current_action_state] was actually a stand in for the actual function we were trying to approximate, that is, the *discounted* sum of Q[action, state] for *all* possible future action states after previous_action_state. We would then look for the action that would maximize that Q. Now, the Smartcab in that context, at least for me, is significantly different to the ticktacktoe scenario; however, I feel the update of Q in this project can be made somewhat akin to Q learning as seen in Smartcab. Since the update occurs at the end of a game, from a step by step perspective of *that particular* end game sequence, its as if we know "all future action states". So I feel like we can also approximate the theoretical "bellman" Q, end-game sequence by end-game sequence, as they are encountered. So the first iteration of that idea, looked something like this; Since a subsequence in our context is a list of moves, for example;

[(0, 'X'), (3, 'O'), (6, 'X') ….]

It kind of already a captures both the action and the state of the environment. The "action" is simply the last move on the list, and the state is the entire list. So suppose we have an end game sequence, call it EG, take a subsequence of it call it, well.. subsequence, since r is fixed by how a particular game ended, r = -1, 0, 1, then Q-learning update can take this form

$Q[subsequence] = r + \lambda * (r + \lambda * (r + \lambda * (r + ….$

where that sum is repeated however many "future states" or *moves* remain from subsequence to EG. In other words we have the simple geometric series;

$Q[subsequence] = r + \lambda r + \lambda^2 *r + \lambda^3 * r ….+ \lambda^n * r$

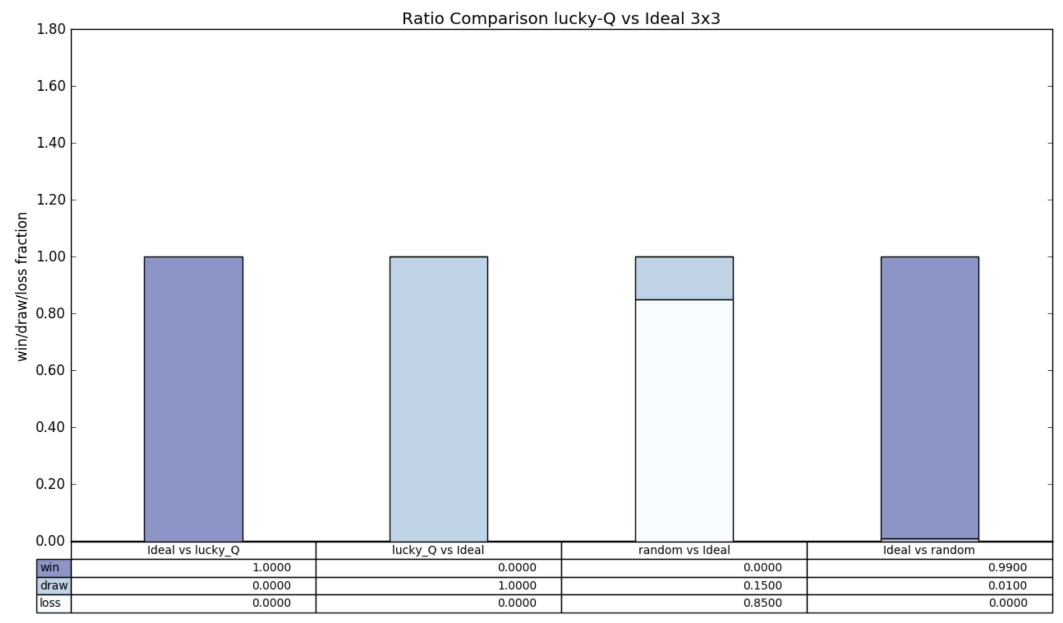
Here, *n* is basically the difference between len(EG) and len(subsequence). So that something like the *discounted sum* of future states can be put into place. After implementing it exactly in that way (you can still uncomment the code to try it) I ended up settling for even less Bellmanish update logic of the form,

Q[subsequence] += r * λ^ len(subsequence)

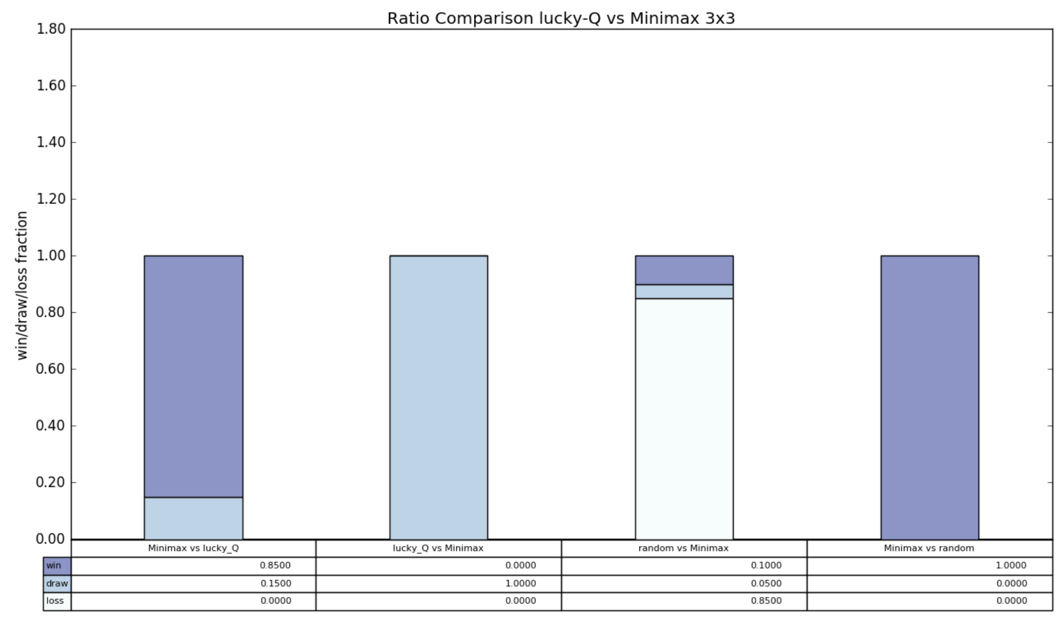
So some of the idea of addition of discounted future states is still present, but I was getting "better" results in this form. At some point r was also calculated as r^4 / len(EG). For now the code uses the equation immediately above with an r of +-1 or 0. Now, when agents refer to Q for there next action, player 1 would choose the next action (i.e, action_state == subsequence == key) that has maximum Q value, and player 2 would choose the one with smallest Q value. Also keys are added dynamically to Q as they are encountered, so that at any given state there could be basically "unexplored" future action_states. In that case the agent randomly chooses one according to an arbitrary threshold. To train Q the approach taken was to basically have a combination of;

- Purely random agents play against each other. In this way populating Q with keys and values, somewhat "uniformly", since that is the distribution that random.randInt() draws from.
- Purely random agent vs Q-learning agent, alternating between player 1 and player 2. Here the intention is to have Q shape itself to mimic actual game play, hopefully not adding value to better moves and not adding value to bad moves (by either player)

Additionally, I found that even if all subsequences at a given point had been already visited, during training having the agent still choose randomly helped dislodge Q from something like a "sink", where, randomly, some game sequences had been over favored leading to narrow behavior.
Let see some of the issues encountered after application of the above framework. Early on after the training process described previously; more specifically, after 70 random vs random 3x3 games, followed by 1000 random vs Qlearning games, and then followed by 1000 Qlearning vs random games, a Q was produced nicknamed lucky_Q (partly because of some difficulties I have yet to discuss). To test its behavior I pit it against the perfect or near perfect players Ideal and minimax. With the following results;

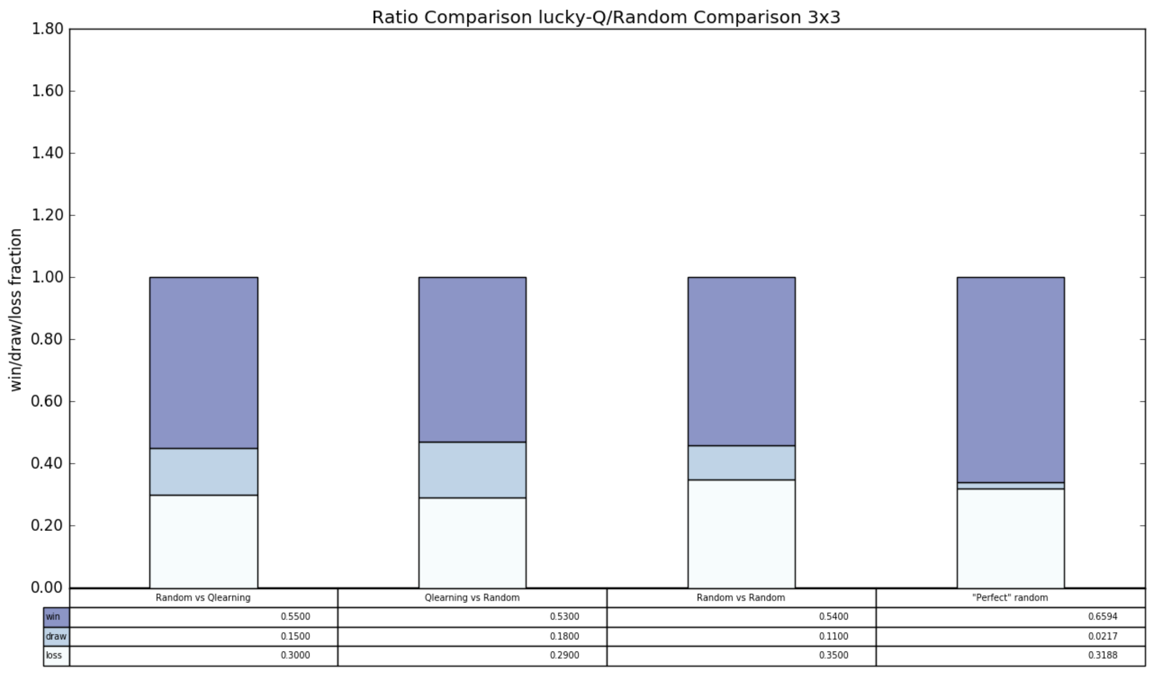


	Ideal vs lucky_Q	lucky_Q vs Ideal	random vs Ideal	Ideal vs random
win	1.0000	0.0000	0.0000	0.9900
draw	0.0000	1.0000	0.1500	0.0100
loss	0.0000	0.0000	0.8500	0.0000



	Minimax vs lucky_Q	lucky_Q vs Minimax	random vs Minimax	Minimax vs random
win	0.8500	0.0000	0.1000	1.0000
draw	0.1500	1.0000	0.0500	0.0000
loss	0.0000	0.0000	0.8500	0.0000

Although it is obliterated by both policies when playing as player 2, similar to how random would (although it shows some resistance to depth 2 minimax i.e, draw = 0.15); amazingly, it absolutely ties 100% of the time against the perfect players when it is in player 1 position! That's behavior, in a partial sense, only exhibited by the perfect players. At that point, this whole process seemed really promising. How does it perform against a random agent? Here are the results;



	Random vs Qlearning	Qlearning vs Random	Random vs Random	"Perfect" random
win	0.5500	0.5300	0.5400	0.6594
draw	0.1500	0.1800	0.1100	0.0217
loss	0.3000	0.2900	0.3500	0.3188

Bleakly, we see it is virtually indistinguishable from a player simply making random moves. There is a slight reduction in losses, maybe, and they mostly go towards the draw portion. So what is going on here? After tracing back the games being played it basically looks like, through its random initialization process, lucky_Q just happed to learn to favor an optimal sequence that the perfect agents respond to with optimal responses and so are drawn into this "sink" that ends in a tie. This shifted the focus from trying to mimic perfect play, to looking for behavior that moves away from random, that is, behavior that will reduce the draw and loss regions and increase the win region, even if most likely the resulting agent won't be able to compete with Ideal or Minimax. Lucky is also lucky because I have found it really difficult to reproduce consistently. Even though I have been using a fixed seed in the module responsible for random moves, strateegery, I have trouble understanding how exactly to trace behavior across classes. I would have thought that the seed is reset every time I create an instance of Strateegery, so that when I perform tests inside of run() inside of the *play* module I could treat them as an isolated runs. But its, not the case, if I have printed out say, some random vs random stats, and then I do some other graph Ideal vs random, previous calls affect the state of random inside *strateegery*. Thankfully I had pickled it, but also the easiest way to recreate it is by checking out the git hash;

f10d0d868cb752b3bb6e26daafe2b844e3ba408b

Here the code was at a more primitive but straight forward state, and the calls to tally inside run() at that point create lucky_Q. To complicate matters more, the seed then was different, and the reward function was also slightly different. It would seem that maybe there was something special about that reward function, but there really wasn't. It just happens to reproduce the state of affairs that produces a lucky_Q, through trial and error you could do the same with the present code base, I don't think its worth it.

New Strategy:

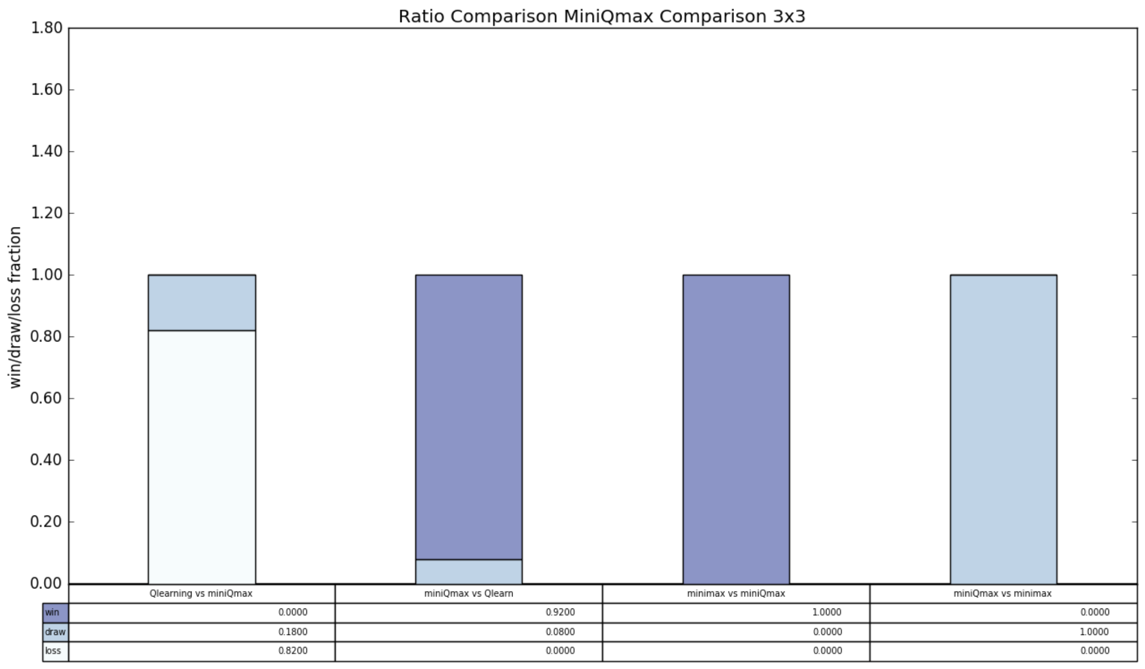
I then fumbled with different tweaks to the above strategy, which at some point included this ridiculous attempt to "breed" different randomly generated Q's (the code is still there for the curious) The most successful of these was lowering λ for Q training (in the code it is called gamma). This actually produced Q's that would widen the draw fraction with Ideal and Minimax more often, at least observationally (didn't gather stats to justify that anecdotal statement). However, it was not enough, and Q's would still play basically random games. So the compromise that dawned on me, which I think the reviewer of the proposal for this project was trying to point me too, and I failed to understand, was to seek to blend minimax with Qlearning. My implementation of minimax could technically play perfectly at sufficient depth but, it simply was too slow. Qlearning was fast but behaving essentially randomly, so the idea was to replace minimax's bottleneck, minimaxMeasure, with calls to Q. In the miniQmax strategy then, Q replaces minimaxMeasure as the evaluate function.

Results:

What follows is the result of Q trained randomly with the 70, 1000, and 1000 trials as described previously, producing a Q of moderate quality. By that I mean, I usually do a small tally of Qlearning against Ideal, minimax, and itself, simply to see if it manages to tie the perfects and if it is balanced with itself in this case those tallies were,

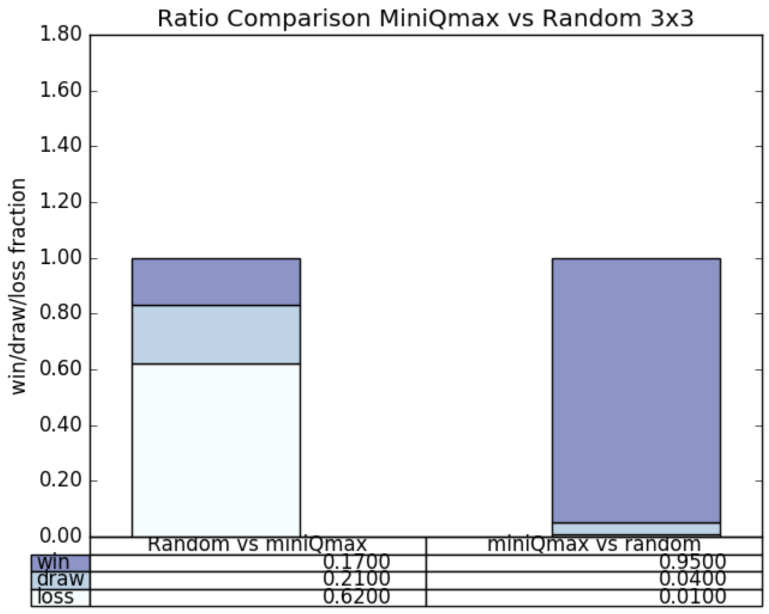
```
Q v Ideal:
{(False, True): 23, (True, False): 0, (False, False): 7}
Q v Minimax:
{(False, True): 24, (True, False): 0, (False, False): 6}
Q V Q:
{(False, True): 8, (True, False): 1, (False, False): 21}
```

So out of 30 trials it managed to tie Ideal 7 times, Minimax 6, and it most of the time ties with itself with 21 time. In contrast these numbers with lucky_Q are ties, 30 ,30, 30. Frustratingly I must say that the exact same code that produces this in my computer terminal , does not produce these numbers consistently when run in the following ipython notebook I provide with the material for this project. To get these numbers you must run that particular section of the code in run *by itself* , but not in the notebook. Again this is due to my partial understanding of the random module used across classes. Now, using this Q as the evaluate function to miniQmax we arrive at the following ratio comparison;



	Qlearning vs miniQmax	miniQmax vs Qlearn	minimax vs miniQmax	miniQmax vs minimax
win	0.0000	0.9200	1.0000	0.0000
draw	0.1800	0.0800	0.0000	1.0000
loss	0.8200	0.0000	0.0000	0.0000

The duels are between Qlearning and miniQmax (both using the same Q) and as a reference miniQmax vs minimax. Also I am able to run miniQmax at a depth of 3 more comfortably. What can be seen is first that miniQmax is *not* just basically Qlearning. In Qlearning vs miniQmax, Qlearning lost 82 % of the time, it was also dominated as player two with miniQmax winning 92% of the time. Minimax still obliterates miniQmax 100% of the time (still at depth 2) as player 1. However, as player 2 miniQmax manages to tie the monster 100% of the time! I call it Lucky's revenge. Here is miniQmax against a random player;



	Random vs miniQmax	miniQmax vs random
win	0.1700	0.9500
draw	0.2100	0.0400
loss	0.6200	0.0100

We see that as player 2 it basically reverses the ratio on player 1, that is if it were randomly playing it is player 1 who should take around 62% of victories. As player 1 miniQmax healthily dominates random with 95% of wins. In the end we get what something like what was hoped for, an agent playing significantly better than random, but exhibiting some tendencies of the perfect players, hurray! A major note on the implementation of miniQmax is that initially it did not work at all. Simply using Q as evaluate was not enough, at the results were if at all, marginally better than random. While debugging it became apparent that as the minimax algorithm would delve into future sequences it seemed to encounter many unvisited states, or "keys". I needed to use the Q.get method because it would try to explore keys that were not there. Here everything becomes more like black magic, because I am not fully confident with the reasons why things are working, but basically I decided to instead of using Q.get(sequence, 0), that is returning 0 when a key was not in the dictionary, I returned a small reward of *current depth's sign.* What that means is if it was on a max pass (player 1) it returns 0.01 when a key is not there, and -0.01 for a min pass. Somehow that small reward for, I guess, "discovering" a new state lead to the win:draw:loss ratios seen above.

Conclusion:

Even though there was some success with the methods used in this project by now I realize that applying a reinforcement learning approach to this problem is maybe not the best of ideas. The main benefit extracted from these explorations is my gaining familiarization with a general algorithm like minimax, and understanding the benefit of combining it with something like a Q dictionary. I feel know it would be maybe more effective to use the minimaxMeasure, that provides such strong measures of state, and simply store its values in Q. That is just use Q like a database to relieve minimax from having to actually perform calculations in the evaluate step, if it is trying to do so somewhere it has already been before. That may provide the speed benefit but with Q values much better tailored to the problem. I also really need to include a user interface so that you can actually play against some of these agents. Also not included in the report are explorations of larger sizes with the methods developed. I realize it is probably not proper form, but I desperately need to submit something, even if not finished, to get feedback from my reviewers, if only to correct course on any glaring problems. Thanks!