Conner Lane

AI Final Project Report

Group members: Grant Cordle, Cody Adams, Will McCarty

****ALL SOURCE CODE AND GATHERED DATA CAN BE FOUND AT: https://github.com/connerlane/reinforcement-learning-checkers ****

I. Task Definition

A. High-level Description

For our project, we decided to create an AI that learns how to play the board game checkers from scratch using Reinforcement Learning. We employed both the use of Q-learning and SARSA to accomplish this.

B. Input/output

The intent of the program is to generate a policy that can take the state of any checkers board as input and return the optimal action from that state as an output

C. Evaluation Metric

The evaluation metric for our problem is defined as the percentage of games that the AI wins over a given time interval. The more games against any given opponent the AI is able to win by following its generated policy, the more successful it is considered.

D. Data source

Originally before we started working on this project, we had figured that it might be feasible to simply play against the AI ourselves to train it. However, once we started, we realized that there are an incredibly high number of states that a checkers board can be in, and it is simply not feasible to train it by hand. Therefore we created instances two other, simpler AI’s that could be made to compete against our Reinforcement Learning AI. I will describe these simpler AI’s in detail later in this report.

II. Infrastructure

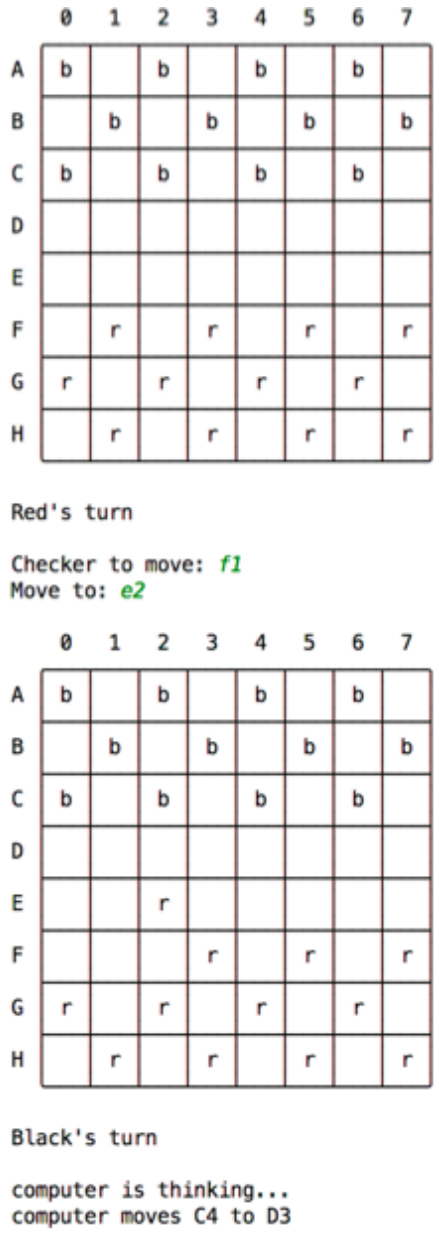
A. Description

For this project, we build the engine completely by hand from scratch in Python. Humans can interact with the board through a command-line interface that we wrote. On this page, you see a screenshot of the command line interface. The board exists as an instance of a class we wrote called Board. It contains the state of the board, represented by a 2-dimensional array, where each spot in the array contains an integer representing the type of game piece (or lack thereof) is located there. The board contains methods to get all legal moves given any configuration, hashing the board state into a base-36 number for q-table storage, and determining the ‘score’ of a board.

The ‘score’ of the board could also be called the ‘goodness’ of a state. We defined the score of a board to be the following:

$$\mathrm{Score} = N_a + 2K_a - N_b - 2K_b$$

Where N_a is the number of normal checkers the AI has, K_a is the number of kings the AI has, N_b is the number of checkers the opponent has, and K_b is the number of kings the opponent has on the board.



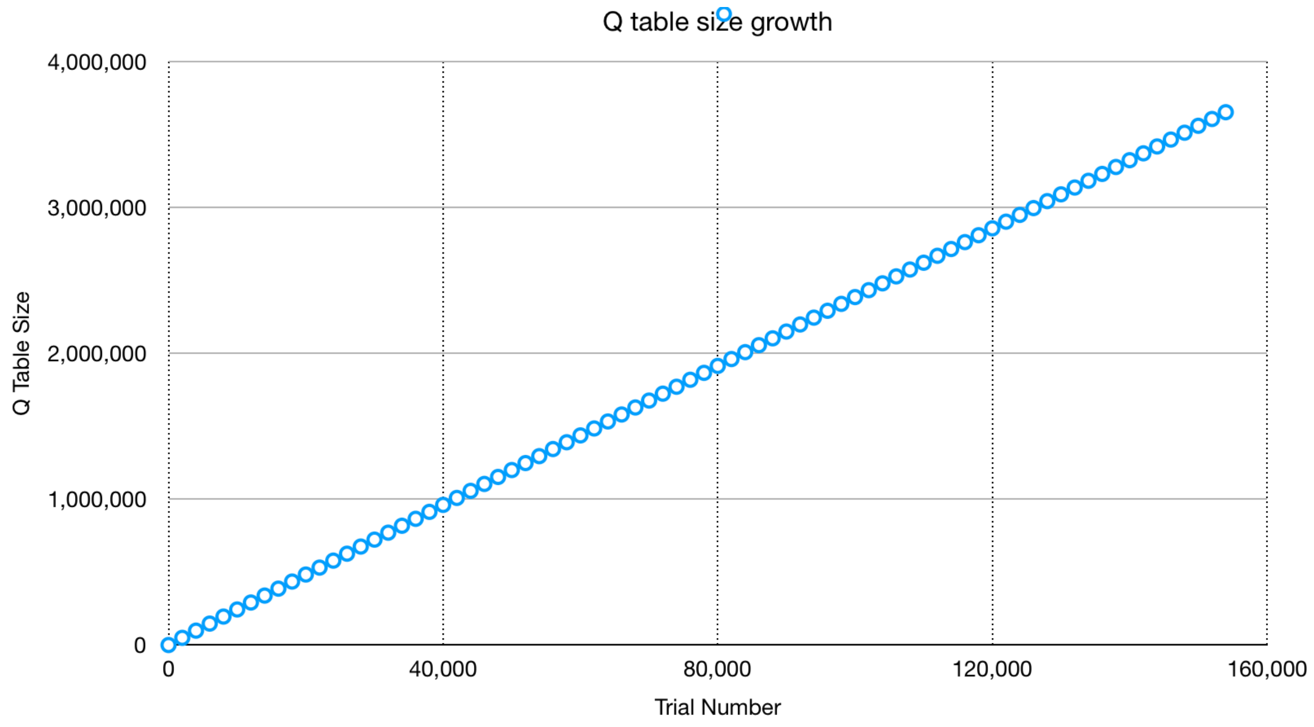
```
    0   1   2   3   4   5   6   7
  +---+---+---+---+---+---+---+---+
A | b |   | b |   | b |   | b |   |
  +---+---+---+---+---+---+---+---+
B |   | b |   | b |   | b |   | b |
  +---+---+---+---+---+---+---+---+
C | b |   | b |   | b |   | b |   |
  +---+---+---+---+---+---+---+---+
D |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
E |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
F |   | r |   | r |   | r |   | r |
  +---+---+---+---+---+---+---+---+
G | r |   | r |   | r |   | r |   |
  +---+---+---+---+---+---+---+---+
H |   | r |   | r |   | r |   | r |
  +---+---+---+---+---+---+---+---+

Red's turn

Checker to move: f1
Move to: e2
    0   1   2   3   4   5   6   7
  +---+---+---+---+---+---+---+---+
A | b |   | b |   | b |   | b |   |
  +---+---+---+---+---+---+---+---+
B |   | b |   | b |   | b |   | b |
  +---+---+---+---+---+---+---+---+
C | b |   | b |   | b |   | b |   |
  +---+---+---+---+---+---+---+---+
D |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
E |   |   | r |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
F |   |   |   | r |   | r |   | r |
  +---+---+---+---+---+---+---+---+
G | r |   | r |   | r |   | r |   |
  +---+---+---+---+---+---+---+---+
H |   | r |   | r |   | r |   | r |
  +---+---+---+---+---+---+---+---+

Black's turn

computer is thinking...
computer moves C4 to D3
```

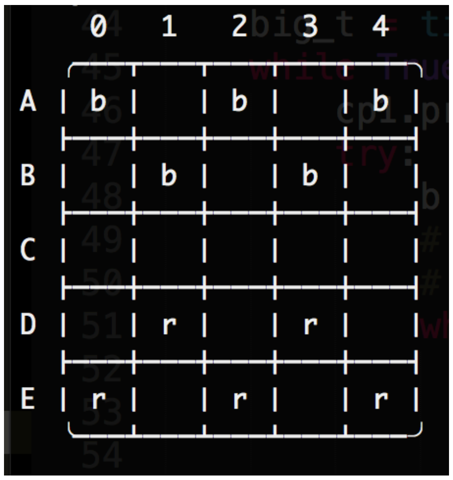
B. Challenges

We encountered many challenges while building the game engine. One challenge was that we had to come up with a clever way to hash the state of a board into a representation that was as small as possible so as to keep the reading and writing of the Q-table as efficient as possible.

Perhaps our *biggest* challenge, however, is that we ended up having to reduce our board size from a standard 8x8 board to a smaller 5x5 board.

After running one of our computers for about 15 hours non-stop with our Q-learning algorithm (which I will describe in greater detail later in the report), we saw that the number of states in the Q-table was growing almost completely linearly, and showing absolutely no signs of converging at all. The AI was seeing completely new paths to terminated games *almost every game*, and as such, had no way to learn the optimal policy given any reasonable amount of time. The chart below shows the (linear) growth rate of the Q-table for an 8x8 board.





```
   0   1   2   3   4
  ┌───┬───┬───┬───┬───┐
A │ b │   │ b │   │ b │
  ├───┼───┼───┼───┼───┤
B │   │ b │   │ b │   │
  ├───┼───┼───┼───┼───┤
C │   │   │   │   │   │
  ├───┼───┼───┼───┼───┤
D │   │ r │   │ r │   │
  ├───┼───┼───┼───┼───┤
E │ r │   │ r │   │ r │
  └───┴───┴───┴───┴───┘
```

Our solution to this problem of a linearly growing Q-table was to reduce the amount of possible states. We decided that the best way to do this, for the sake of the project was to reduce the board size from a standard 8x8 board to a 5x5 board. The starting state of the 5x5 board is seen to the right of this paragraph. There are 32 legal spaces in an 8x8 board, and 13 in a 5x5 board. There are 5 different states a space could have (empty, black checker, black king, red checker, red king). 5^{13} is 5^{19} (about 19 trillion) times smaller than 5^{32}. Therefore, this change *greatly* reduced the magnitude of the possible board configurations, and allowed us to obtain reasonably good performance after running the learning algorithm for a few hours.

C. Baseline/Oracle

The baseline algorithm for this project was simply a policy that said “if a jump is possible, take the jump. otherwise, choose a random move”. This is naive and aggressive, but serves as a good baseline

Unfortunately, there is no oracle for the game of checkers that we were able to make use of, as we ourselves do not always know the best move to take, and cannot algorithmically determine in a reasonable amount of time

III. Algorithms

A. Overview

We generated many different training results from scratch by tweaking parameters and opponent behavior across 2 different variations of Reinforcement Learning: Q-Learning and SARSA. Reinforcement learning involves the utilization of rewards. For our implementation, a reward for an action in state S that results in state S+1 is defined as:

```
Reward = score(S+1) - score(S)
```

(using the score function defined on a previous page of this report).

For an action *a* in state *s* that resulted in a **won** game, the Q value for (*s, a*) was updated to a value of 25.

For an action *a* in state *s* that resulted in a **lost** game, the Q value for (*s, a*) was updated to a value of -10.

The AI's were made to compete against 2 different "dumb" AIs.

One of the opponents was simply the baseline algorithm. I will refer to this algorithm as being the "naive AI" or alternatively the "aggressive opponent"

The other opponent simply selected moves completely at random. This one resulted in a much higher state discovery rate against the AI.

B. Q-learning

Q learning involves updating the Q-value (or quality) of a state-action pair based on the reward received from taking that action from that state, and the maximum Q-value for the resulting state. The update rule is defined as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

C. SARSA

SARSA is similar to Q-learning, but instead of considering the highest Q-value for the successor state, it considers the Q-value for the chosen action from the current state. This can save computation for the max Q

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

value, especially if the action-space is very large.

IV. Literature Review

The first comparison I will make is against an existing implementation of a reinforcement learning found at https://github.com/SamRagusa/Checkers-Reinforcement-Learning written by Sam Ragusa.

Ragusa wrote a similar reinforcement learning algorithm to ours, and he had very similar design in his game engine as we did. His “score” function is also identical to ours, although he uses different notation.

However it appears that the key difference between his implementation and ours is that he chose to not store states of the board, as there are too many. He instead chose to discretize board states based on several elements in the board.

There are pros and cons to Ragusa’s approach. One significant pro is that his state space will be much more explorable, as it will be significantly smaller. One con to this approach, is that it will never be able to truly solve the game, or ever achieve as good of results as ours would one day achieve given enough time and memory, as he cannot truly determine the best move for a given state, as his algorithm sees that state as being the same as other different states.

The second comparison I will make is against *Reinforcement learning project: AI Checkers Player* by Amco Dubel, Jaap Brandsema, and L. Lefakis. The authors of this article detail yet another way to avoid the massive amounts of states for a checkers board, and that is through the use of function approximation. They detail a way by which a state can be approximated by using a neural network with one hidden layer.

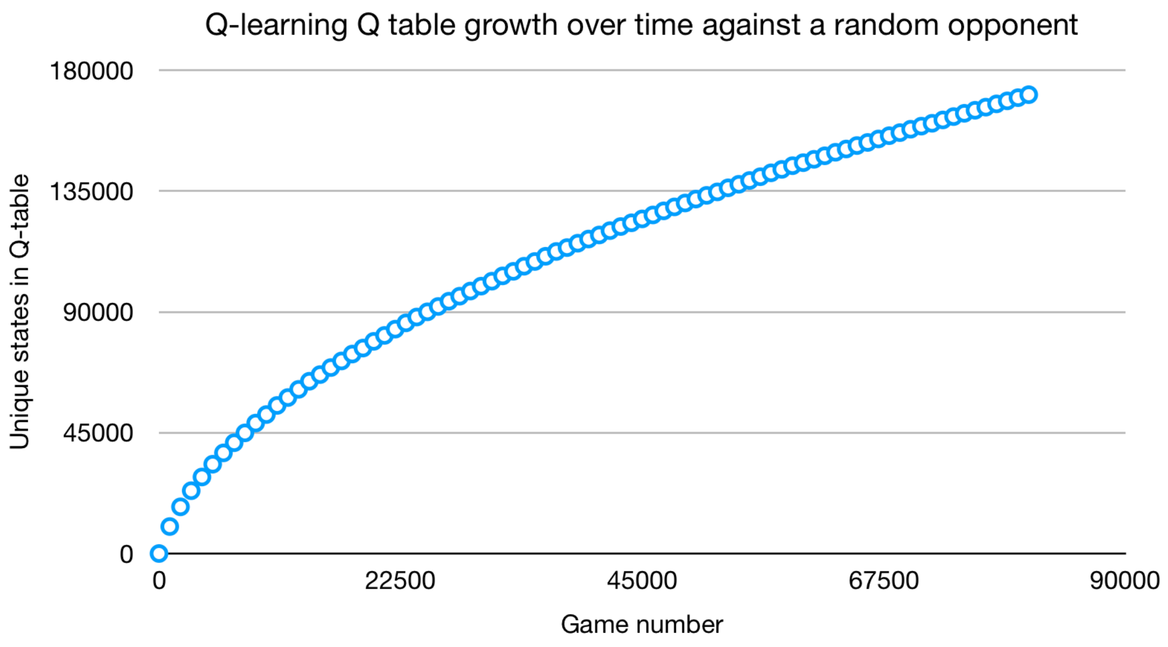
Another thing that these authors did differently, is that they found that by having the learning agent perform greedily all the time, the performance was surprisingly unaffected.

V. Analysis

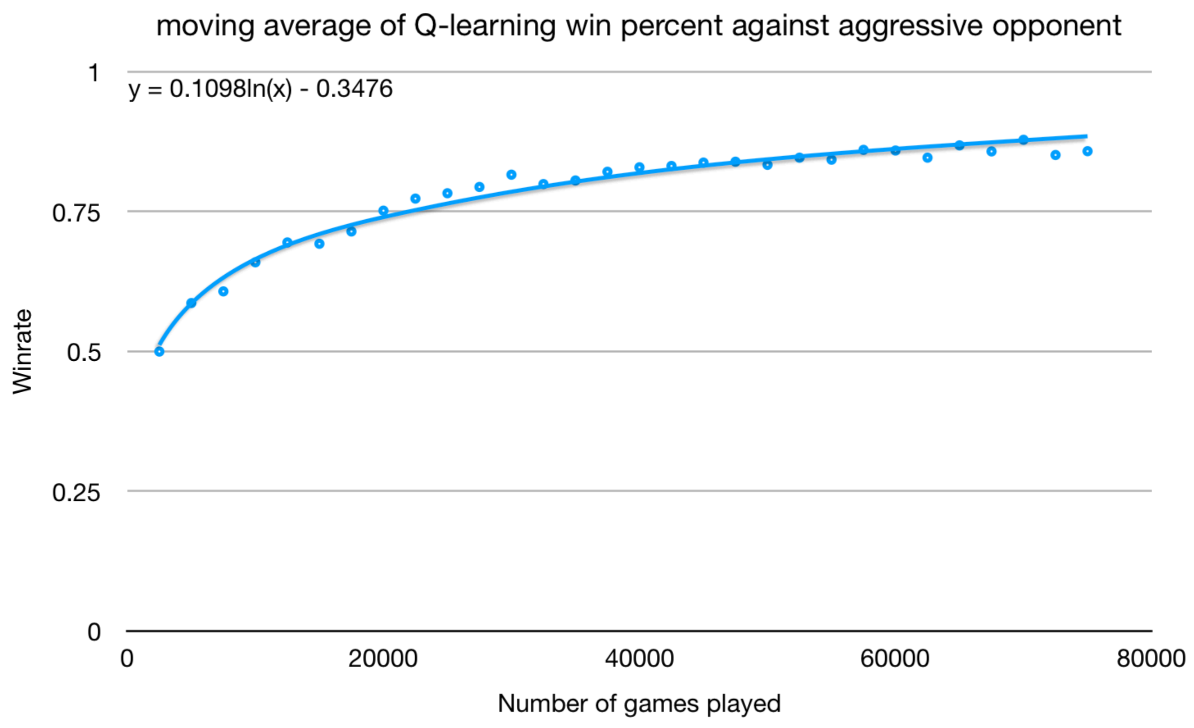
A. Description

We gathered a large amount of raw data that can be found through the GitHub link on the first page of the report. We have used that data to compile charts that provide insights to the differences between Q-learning and SARSA

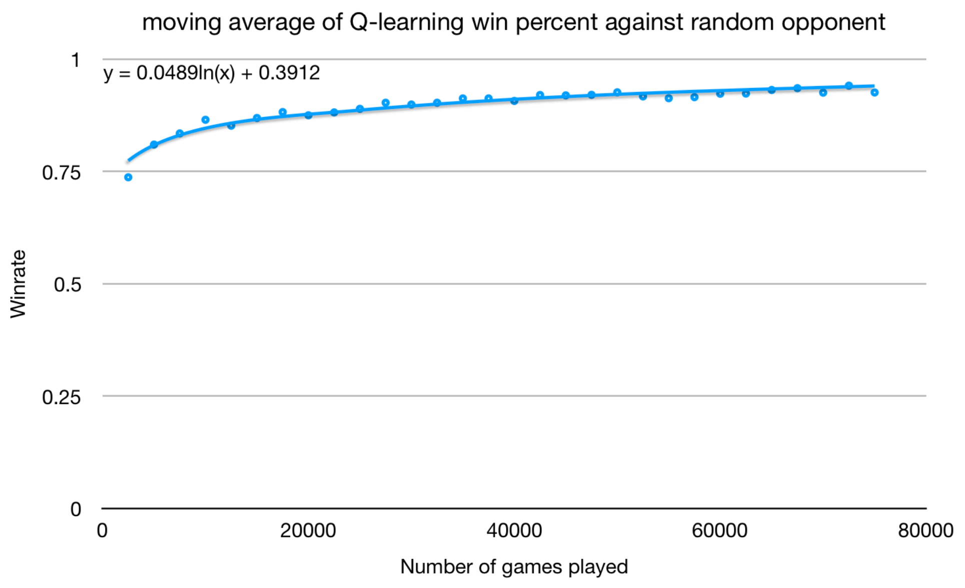
B. Q-Learning



The first chart to be seen is the growth of the Q-table. This is simply the number of explored states in the Q-table as a function of number of games played. (for this graph, the random AI was used as the opponent).

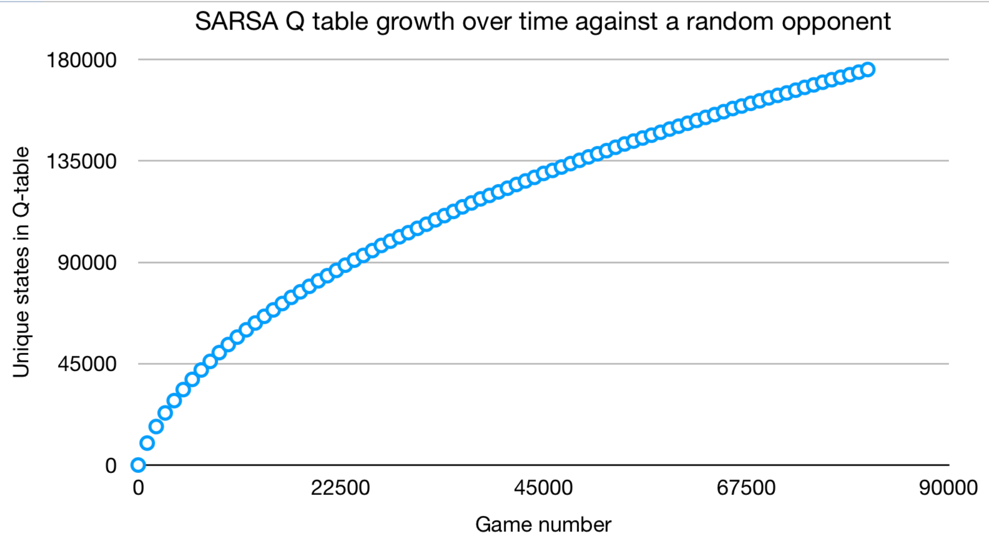


Above is the measured performance of Q-Learning vs the aggressive AI

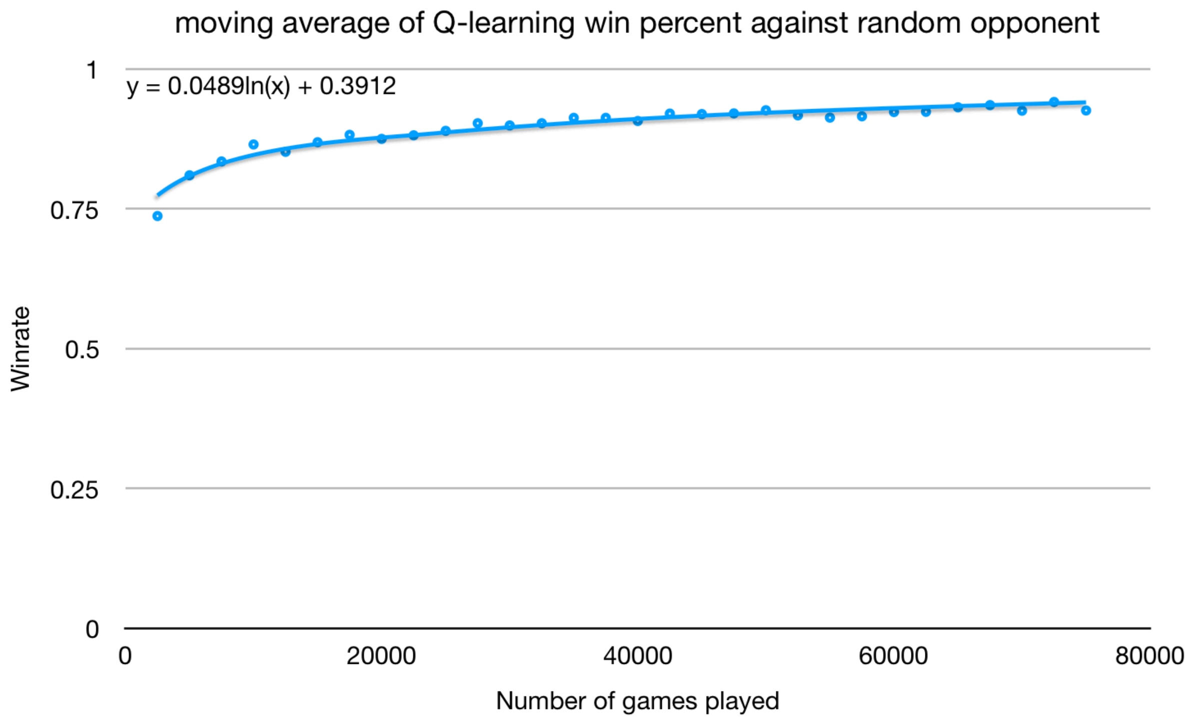


This figure is the performance of Q-Learning against the random AI. As you can see, although the random opponent helps the AI to find states faster, our Q-Learner has an easier time beating the opponent.

C. SARSA



This is the SARSA Q table growth against the random AI. It is almost exactly the same as the growth for Q-Learning. SARSA did do *slightly* better in this, however.



Here is the performance of SARSA against a random opponent. As you can see, SARSA performed slightly worse than Q-Learning did in terms of actual performance according to our evaluation metric.

Overall, therefore, Q-Learning appears to be superior to SARSA for the problem definition without using function approximation.

Works Cited

Dubel, Amco, et al. “Reinforcement Learning Project: AI Checkers Player.” *ResearchGate*, 1 Jan. 2006, www.researchgate.net/publication/242405861_Reinforcement_learning_project_AI_Checkers_Player?enrichId=rgreq-a5b54cf8e1d4c080ec390949cef3950a-XXX&enrichSource=Y292ZXJQYWdlOzI0MjQwNTg2MTtBUzoxMDY2MjE2MTUyMTQ1OTRAMTQwMjQzMTk4MjkxOQ%3D%3D&el=1_x_3&_esc=publicationCoverPdf.

"SamRagusa/Checkers-Reinforcement-Learning." GitHub - SamRagusa/Checkers-Reinforcement-Learning: A Checkers Reinforcement Learning AI, and All the Tools Needed to Train It.. N.p., n.d. Web. 11 Dec. 2017. <https://github.com/SamRagusa/Checkers-Reinforcement-Learning>.