Using Reinforcement Learning to Develop a Better Strategy for Ramsey Games

KANDASAMY CHOKKALINGAM

ABSTRACT

Reinforcement learning models allow for quicker learning of patterns that are hard to identify. By rephrasing Ramsey's theorem as a game, we apply Reinforcement learning to them to develop a better strategy. In this study, we trained and developed reinforcement learning models to play Ramsey games on graphs to find a better strategy that was the least costly - one that takes the least time and memory. A Graph Neural Network with Q-Learning model was hypothesized to have the best strategy and be the most efficient. Agents were trained over multiple games and data was collected to determine the strength, performance, and cost of the model. Graph Q-Learning models were found to be the best performing and least costly overall. Further experimentation needs to be done to evaluate the performance of these networks on more complex graphs with more computing power.

1 INTRODUCTION

1.1 Ramsey's Theorem

Let there be a graph G with n vertices. Edges will be added and colored with one of 2 colors: red or blue. A group of edges of size j is monochromatic if they fully connect all j vertices and share the same color. Ramsey's Theorem states that for all $m \geq 2$, there exists $n \geq m$ such that every 2-coloring of G of n has a monochromatic K_m [3]. This K_m consists of points that all have the same colored line between them. This means that for every graph size and group size, there is some graph where there is a fully connected group of edges that share the same color. For a 2-coloring in red and blue of a graph, let the least number of vertices a graph needs be equal to $R(a)$ where there must be a blue K_a or a red K_a. Ramsey numbers have applications in network reliability, information theory, and proving that programs terminate.

Ramsey numbers are known for small values of a. With certain theorems, bounds can be established for Ramsey numbers, but for larger values of a, the bounds become much larger than the actual Ramsey number [4]. Upper bounds can be found but they are often not tight, and it is difficult to prove there is no better way to color the graph.

This theorem inspires the following game. This game starts with an empty graph of n vertices and two players. Both alternate coloring lines between vertices and the first one to get a monochromatic K_m in their color wins. Every turn, each player will have certain legal actions it can perform. In this case, the actions will be adding an edge of the player's color between two vertices. The player will look at these actions and make a decision based on its policy, which includes the state of the graph. This decision will be reflected in the graph and on its next turn, the player will receive a reward based on its action. In this case, if the action wins the game for the player by making a monochromatic K_m, the player will receive a reward of 1. In Ramsey games, the player cannot make a move that will directly cause it to lose, but it will receive a reward of -1 when the adversary wins. The player will also receive a reward of -1 if the game ends because all vertices are colored and no one has won; this can only happen on graphs with a number of vertices that is not a game Ramsey number. Game Ramsey numbers are different from Ramsey numbers because the number of blue and red edges is roughly the same at any point in the game. With normal Ramsey numbers, a coloring can have any amount of edges in a certain color. However, game Ramsey numbers can still provide a good estimate of Ramsey numbers. If the player's action does not end the game, it will receive a reward of 0.

An ideal player is one that will always play the best it can, winning if at all possible, drawing if not, and then losing if it is impossible to draw. By Ramsey's theorem if the number of points is large enough, namely at least $R(a)$, then the game cannot be a draw. Our interest is in finding smaller numbers such that the game cannot be a draw.

An ideal policy is one that will always produce a move that leads to the best outcome. A player will always force a draw if possible if it has an ideal policy. This is significant because this approach can reduce the amount of time needed to prove if certain graphs' vertices are game Ramsey numbers. If an ideal policy existed, only one game would need to be played on a graph with two ideal players to tell if it is a game Ramsey number. It is

difficult to find a truly ideal policy, but an optimal policy may be able to be found instead through reinforcement learning. This would not eliminate all graphs, but would have the ability to eliminate some. This way, without brute-force or mathematical proofs, the cost of proving game Ramsey numbers may be able to be significantly reduced through the use of reinforcement learning.

1.2 Reinforcement Learning

Reinforcement Learning is a type of machine learning where models start with only the basic knowledge about a game: the core rules and the win conditions. The models learn a strategy, also called a policy, from playing the game repeatedly. They use the outcome of a game to get feedback on the moves they have played.

One reinforcement learning policy can be done with Monte Carlo Tree Search (MCTS). This method uses random sampling of playouts of a game with limitations to get an approximation of the benefit of different actions. MCTS simulates many random playouts of possible games from the current state. After the random simulation, MCTS looks at which moves leads to the most wins and chooses that one. This method is computationally less expensive and can be limited to fit computation needs.

Another method is Q-Learning. Q-Learning assigns a score to every combination of state and action based on the quality of the action in the long run. Q-Learning finds q-values based on the maximum q-values of its children and the reward for that state. In Tabular Q-Learning (TQL), every state-action pair and its corresponding q-value are stored in a table to be accessed and updated. Every combination has a q-value of q_0, which is updated according to the Bellman formula: $q_{new} = q_{old} + \alpha * (r + \gamma * m - q_0)$. Here, α is the learning rate, γ is the discount rate, and m is the maximum optimal q value from that state [2, 7]. While this approach does allow for specific storing of q-values, it quickly becomes impractical in storage. The larger the amount of states and actions, the more q-values need to be stored.

To fix this issue, Deep Q-Learning (DQN) can be used. Deep Q-Learning uses a neural network to approximate the q-values. The neural network generates the q-values for all actions from a certain state. This network is trained with the same Bellman equation. This method can approach the same amount of precision as tabular Q-Learning, takes much less storage, and has the added advantage of being able to estimate q-values for states it has not seen before.

Deep Q-Learning takes in the board state as input and has fully connected layers in the neural network. This network can be augmented with Graph Neural Networks, which gain more information about the graph state and take into account relations between vertices. We call these models GQN models. GNNs can allow us to improve feature detection in graphs by considering different groups of vertices. GNNs work by taking in a feature matrix, which in our case would be a 6x1 matrix of colors for each node, and an adjacency matrix, which in our case would be a 6x6 matrix of all 1's except on the diagonals. [1, 5, 6, 10]. By incorporating information from neighboring nodes, GNNs may be well suited for Ramsey games because making groups of nodes is the crux of Ramsey games. A GQN model is predicted to be the best overall because it effectively considers groups of vertices which gives it the most information and least cost.

2 METHODS

All code was written in Python 3.8.5. The code for each agent contains all of its decision making only. All utility functions are in the Utils class so they can be used consistently across agents. All agents that used neural networks used an Adam optimizer, MSE loss function, and Xavier uniform weight initialization for the layer's weights unless otherwise noted. All agents that used neural networks made use of replay experience and a target network. This graph has n nodes and colored edges are represented by weights with 1 representing a red edge and -1 representing a blue edge.

2.1 Agent Specifications

The Tabular Q-Learning (TQL) agent was implemented using a dictionary to store q-values for each state. The dictionary's keys consisted of the state's current adjacency matrix weighted by the edge weight and then the current action. The current action was in the form of a tuple of nodes.

The Deep-Q Learning Agent (DQN) was implemented using pytorch. The agent's policy was a neural network where the input was the upper half of the weighted adjacency matrix of the state, and the output was the q-values for each possible action. The neural network had an input layer, 2 hidden layers of the hidden layer size hyperparameter, and a single hidden layer of half the size of the hidden layer size hyperparameter and the output layer. Each layer other than the output layer went through a ReLU function before being fed into the next layer.

The Graph Q-Learning (GQN) agents utilized Graph Neural Networks (GNNs) in addition to deep q-networks. These GQN's neural networks are structured similarly to how Jiang et al. structured their network. The networks have an input which is a GNN layer, two hidden GNN layers with skip connections from the previous layer and then 2 fully connected linear layers. All layers have ReLU activation functions in between. Three GQN agents were made, one using Graph Convolutional Layers [6] called GQN-2, one using EdgeConv Layers [9] called GQN-3, and one using GATConv Layers [8] called GQN-1. Each GQN is the same except for the type of layer used. GQN-3 did not use weight initialization.

The Monte Carlo Tree Search (MCTS) Agent utilized monte carlo tree search for its policy. The tree is limited to a depth of 4 because of the large tree structure. The MCTS agent uses upper confidence bounds to select its next move after simulation. A heuristics function was used to aid the algorithms in selecting a state during rollouts. The heuristics algorithm was used with a probability of $1 - \epsilon$ or random rollouts with an ϵ probability. The heuristics function takes into account the number of cliques, average length of the cliques, number of cliques missing the same edge, and the number of mixed cliques that are mostly the agent's color. It measures the amount of each and returns the sum as the heuristic score. The number of cliques and average length of cliques promote both more and longer cliques.

Since each agent had a number of hyperparameters, Ax was used to tune the hyperparameters with bayesian optimization. The following hyperparameters were predetermined unless otherwise noted. The MCTS algorithm samples 200 times. The minimax with depth restrictions has a depth restriction of 4. The Q-Learning algorithms has an α of 0.8, ϵ of 0.5, which decays every move by a factor of 0.99997. The α hyperparameter is a constant that

preserves old q-values for a state, γ is the discount rate for rewards, and ϵ is the chance of a random action being selected. To encourage quick games, the algorithms have a γ of 0.3.

2.2 Training and Testing

Each agent type was trained against an opponent of the same type as this increases robustness of a model. Training was conducted by allowing the agents to play games from no knowledge and learn from the games they played as they played. The agents were trained for 30,000 games, or for a maximum of 6 hours. During training the number of moves, win rate, mean time taken to make a move, memory used, average loss were collected every game. The time collected was measured in nanoseconds using the python time module. The memory used was measured using the python tracemalloc module.

After being trained, each model was saved. Each model was played against a random agent 3 times. The random agent chose a random valid move each turn. This served as a common benchmark for each model. Each model played against the random agent for 500 games except for MCTS which played against it 10 times as it was limited to 6 hours of training time. The amount of wins the agent had being both the player and the opponent was collected for each trial. The percentage of wins was averaged over the three trials and used to statistically compare the models. An ANOVA test was performed to check that the groups were significantly different. Afterwards, the Bonferroni method for multiple tests was used when comparing each model against each other. For models that were not equal, one-sided t-tests were used to determine which model performed better. Both of the best performing models in $R(3)$ and $R(4)$ will play a game against a random agent, and their moves will be used to study their strategy.

3 RESULTS

3.1 Random Player Win Rates

Each of the trained models played against a random player for 500 games 3 times each. The mean win rate of each set of 500 games is shown in the following graphs grouped by agent type. The following agents were tested: MCTS, GQN-1, GQN-2, GQN-3, TQL, DQN. Agents were trained on graphs with the number of vertices for $R(3)$ and $R(4)$. The Players label refers to agents who go first in a game, and the Opponents label refers to agents who go second in a game. A 100% win rate means the agent wins every game against the random opponent, indicating the agent's strategy for playing the game is optimal.

Fig. 1: Win Rate of Models against a random player on R(3)

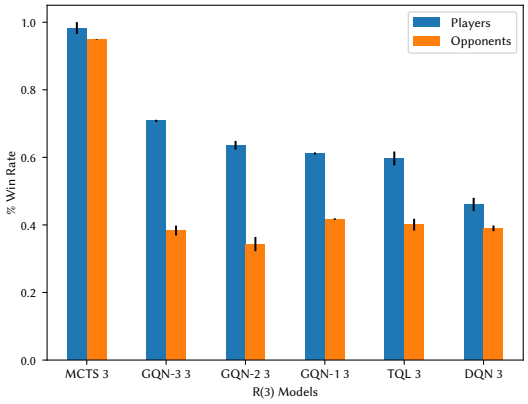
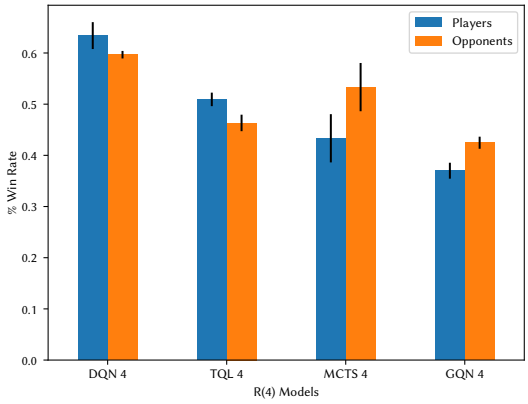


Fig. 2: Win Rate of Models against a random player on R(4)



Random Player Win Rates

In Fig.1, MCTS Player had the highest 98.3% win rate and MCTS Opponent had the highest 94.9% win rate, while the other agents had win rates ranging from 65% to 35%. The GQN players had a higher win rate than the other agents except for MCTS, but a lower opponent win rate. All opponent win rates were lower than their respective player win rates. In addition, player win rate did not correspond with opponent win rate, meaning a model type that had a high win rate for its player type was not also guaranteed to have a similarly high win rate for its opponent type.

On $R(4)$, DQN Player had the highest win rate of 63.4% win rate and DQN Opponent had a 59.7% win rate. The win rates ranged from 60% to 35%. GQN had the lowest win rate on $R(4)$ for both player and opponent. MCTS did not have as high a win rate on $R(4)$ as it did on $R(3)$, and its opponent type did better than its player type, which is not reflected in other models. After comparing the means, we find that for both Player and Opponent DQN is better than all the other models. There is no statically significant evidence to prove the other models are not equal, although the TQL Player is better than the GQN Player.

3.2 Model Training Characteristics

While each model was being trained, win rate, average move time, number of moves and memory usage was collected every epoch. One epoch corresponds with one game. On $R(3)$ TQL and DQN agents were trained for 30,000 epochs, GQN agents were trained for 3000 epochs, and MCTS agents for 1000 epochs. On $R(4)$, DQN agents were trained for 1500 epochs, TQL agents for 100 epochs, GQN agents for 500 epochs, and MCTS agents for 20 epochs.

Fig. 3: Win Rate of GQN and MCTS Models on R(3)

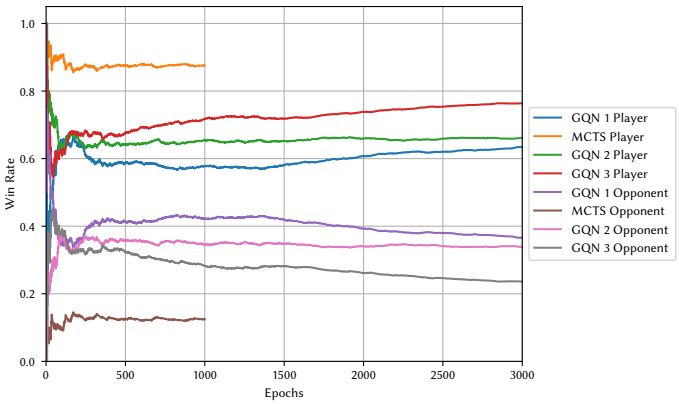


Fig. 4: Win Rate of GQN and MCTS Models on R(4)

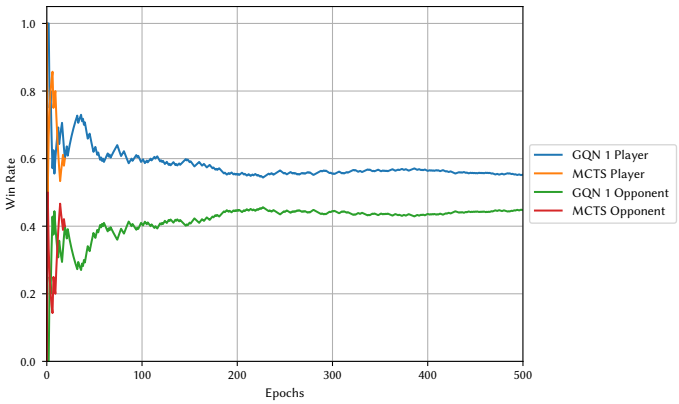


Fig. 5 : Win Rate of TQL and DQN Models on R(3)

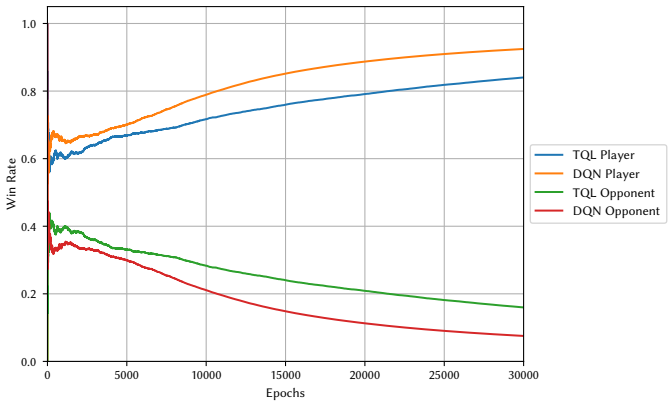
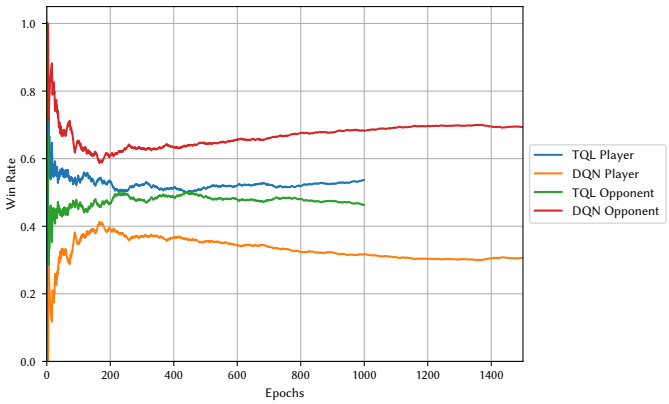


Fig. 6: Win Rate of TQL and DQN Models on R(4)



Win Rates

Win rate was calculated by taking the total amount of wins and dividing by the total amount of played games. The faster the growth of the win rate, which means the model is winning more, and a win rate farther from 50% represents the model having a better strategy to play games.

For $R(3)$, Figures 3 and 5, GQN-3's win rate grows the fastest. MCTS has a high win rate very quickly, and its win rate does not change significantly. GQN-1 and GQN-3 had similar growth in win rates, with GQN 3 having a greater win rate. The TQL player had the highest final win rate of 92% and the DQN player had a final win rate of about 84%.

For $R(4)$, Figures 4 and 6, DQN's win rate grows the fastest across all models. TQL did not significantly differ from a 50% win rate for both players. GQN 1's win rate converges to 55%. DQN had a final win rate of 69%. All win rates tended to be lower than those in $R(3)$ and win rate growth was also slower. DQN had the highest win rate of 88%.

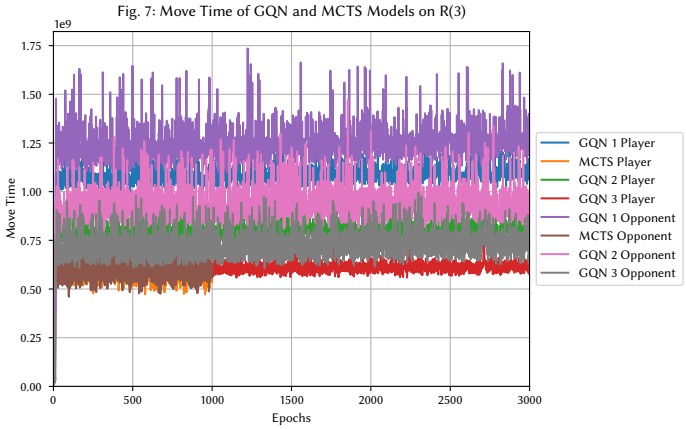


Fig. 7: Move Time of GQN and MCTS Models on R(3)

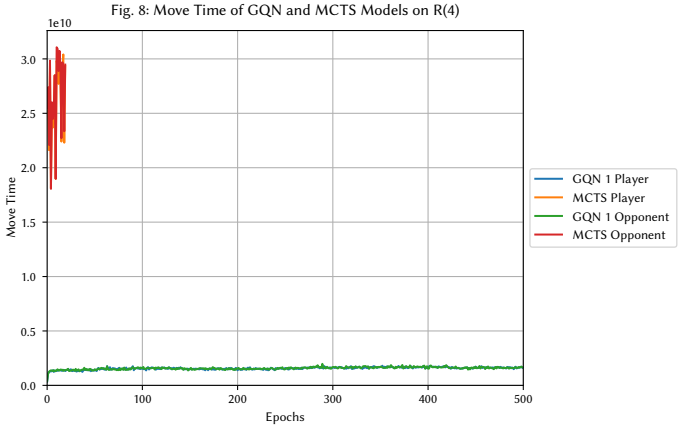


Fig. 8: Move Time of GQN and MCTS Models on R(4)

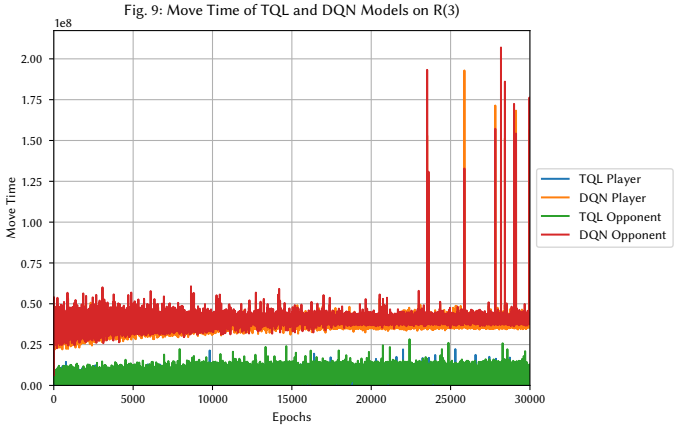


Fig. 9: Move Time of TQL and DQN Models on R(3)

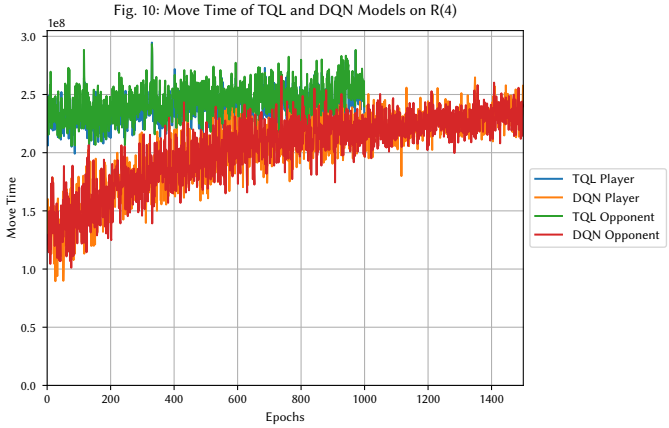


Fig. 10: Move Time of TQL and DQN Models on R(4)

Average Move Time

Average move time is in nanoseconds. It is calculated for an epoch by taking the average of the times taken for an agent to make its move for all moves in that epoch. An agent with a faster move time is more efficient and less costly. An agent with a move time that does not grow with epochs is also more efficient and less costly.

For $R(3)$, Figures 7 and 9, GQN-1 Opponent had the highest move time. TQL agents had the lowest move time. GQN and MCTS agents had a move time at least 1 order of magnitude higher than TQL and DQN agents. DQN agents move time slightly varied with epochs, with less time in earlier epochs.

For average move time on $R(4)$, Figures 8 and 10, DQN agents had the lowest move time. DQN agents' move times varied with epochs growing as epochs increased. MCTS agents had the highest move times, which were at least 2 orders of magnitude higher than TQL and DQN agents'. GQN agents had a consistent move time, and were higher than those of TQL and DQN agents.

Fig. 11: Average Number of Moves of GQN and MCTS Models on R(3)

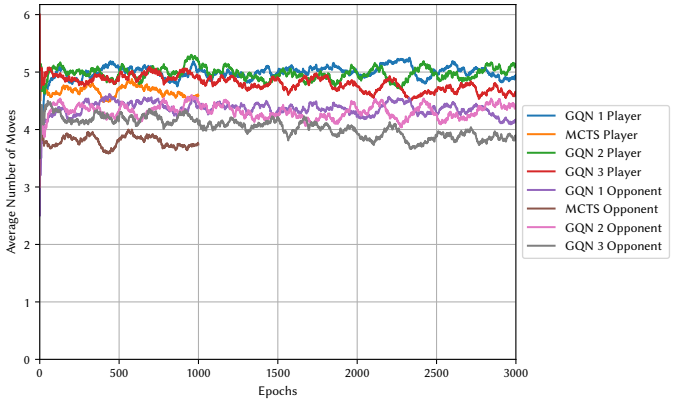


Fig. 12: Average Number of Moves of GQN and MCTS Models on R(4)

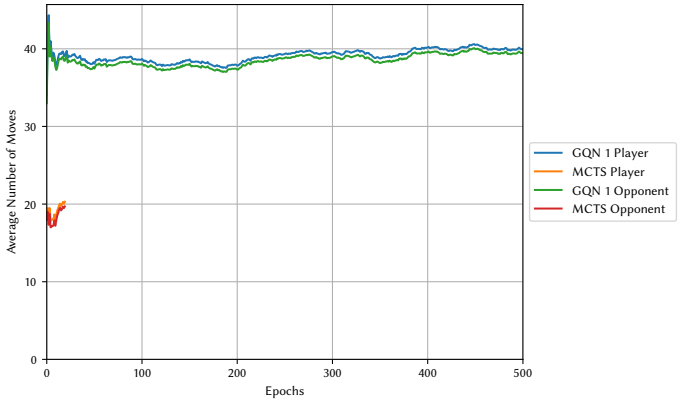


Fig. 13: Average Number of Moves of TQL and DQN Models on R(3)

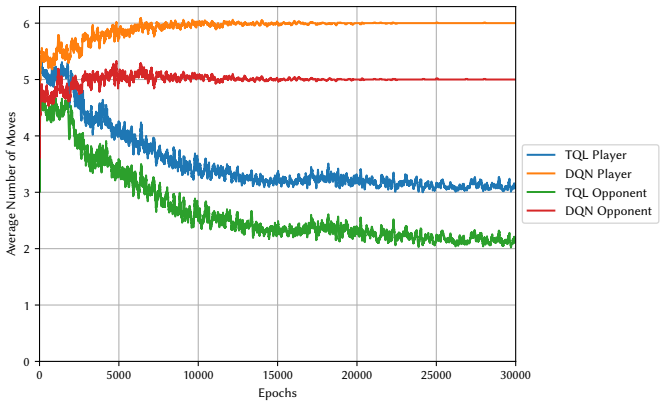
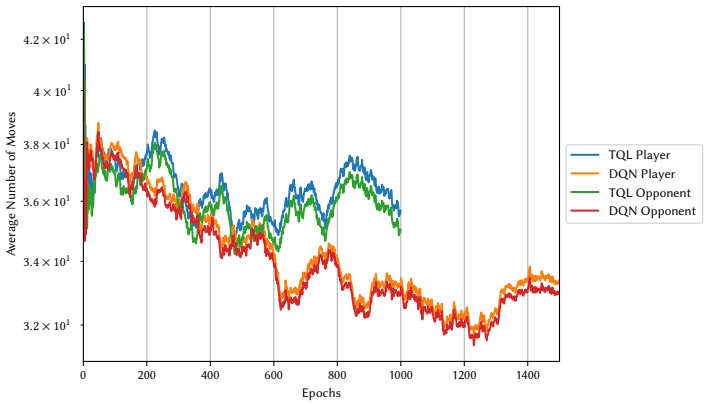


Fig. 14: Average Number of Moves of TQL and DQN Models on R(4)



Average Number of Moves

The average number of moves is the amount of moves an agent made during an epoch. Since the data varied significantly, the graphs show a smoothed version of the data where a rolling window average of size 100 was used. An agent with the most amount of moves is competing itself the most optimally and has the most optimal strategy.

On $R(3)$, Figures 11 and 13, DQN had the most amount of moves. GQN 3's number of moves decreased with epochs. All agents had between 8 and 3 moves every epoch. TQL had the lowest amount of moves and they decreased over the epochs suggesting that there is some disparity between the players. On $R(4)$, Figures 12 and 14, MCTS had the lowest amount of moves. GQN, TQL and DQN agents had between 30 and 50 moves in one game, while MCTS had about 20. TQL and DQN agents' number of moves became less varied with epochs, and the number of moves also decreased. GQN-1 had the highest number of moves.

Fig. 15: Memory Usage of GQN and MCTS Models on R(3)

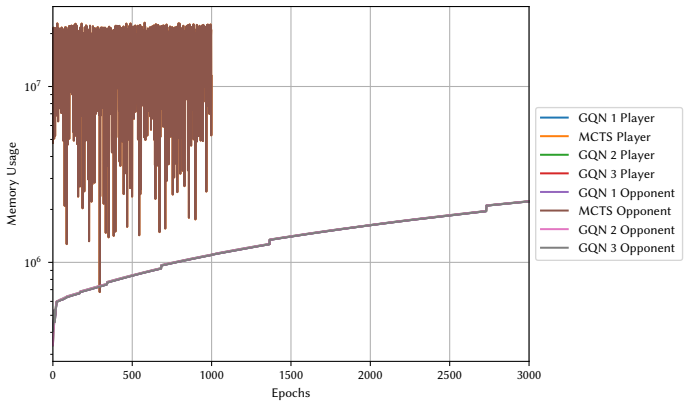


Fig. 16: Memory Usage of GQN and MCTS Models on R(4)

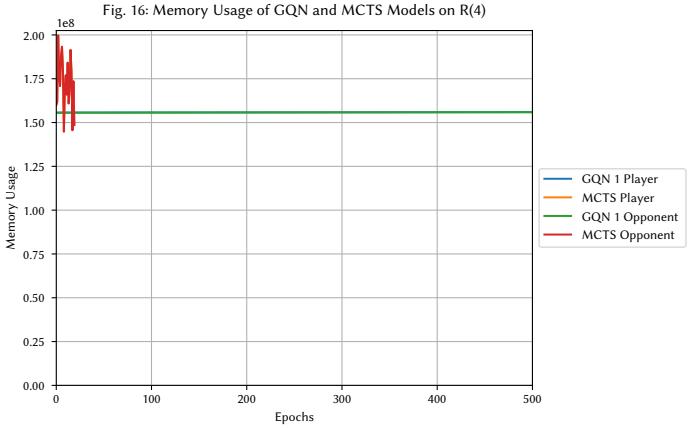


Fig. 17: Memory Usage of TQL and DQN Models on R(3)

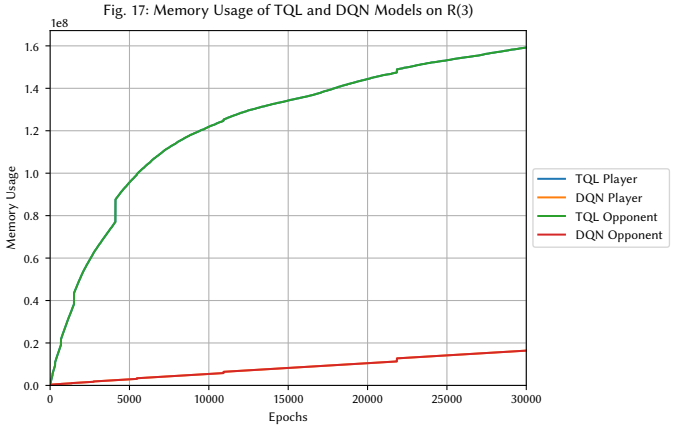
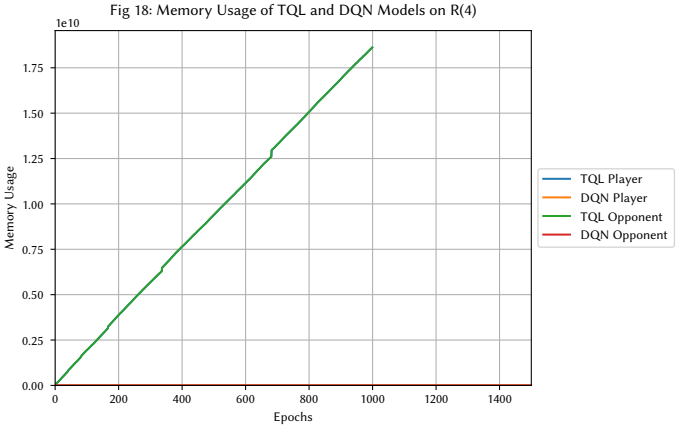


Fig 18: Memory Usage of TQL and DQN Models on R(4)



Memory Usage

The above graphs show memory usage of agents every epoch. Higher memory usage is less efficient and more costly. Memory usage that does not grow with epochs is more efficient and less costly. The bottom graphs' y axes are on a logarithmic scale.

On $R(3)$, Figures 15 and 17, the TQL agents had the highest memory usage. TQL agents' memory usage grew significantly with epochs. The GQNs all had similar memory usage and had the lowest usage overall. The MCTS agents varied significantly in their memory usage over epochs.

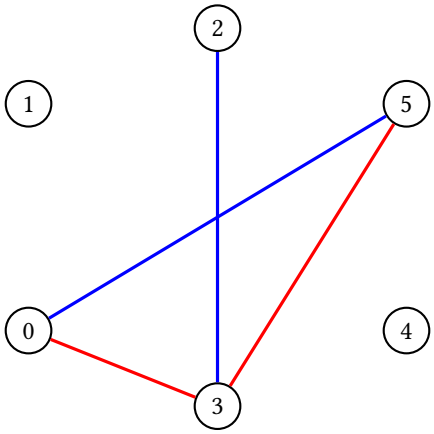
On $R(4)$, Figures 16 and 18, TQL agents had the highest memory usage by at least 2 orders of magnitude at the peak. The memory usage of the TQL and DQN agents grew steadily with epochs. MCTS agents' usage varied significantly as well. GQN had the least amount of memory used which was also consistent.

3.3 Strategy Evaluation

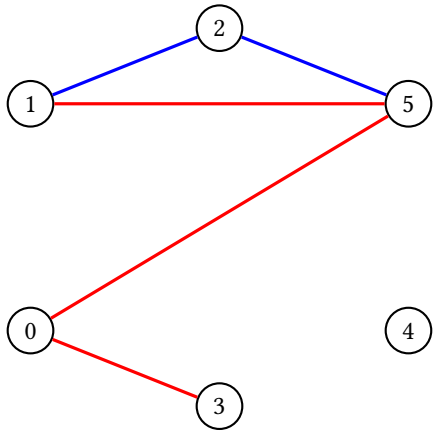
Each of the top models in win rate against a random player were tested for strategy. This was done by having each model play against itself. Therefore DQN was tested for $R(4)$ and MCTS for $R(3)$.

An model with an optimal policy, and therefore strategy, would have the most amount of moves possible if both agents have a similar strategy. The number of moves would be smaller if one agent has a better strategy than the other. The number of moves should be compared to the total possible for that board size, in this case 15 for $R(3)$ and 77 for $R(4)$.

When playing against itself, the MCTS Player on $R(3)$ took 8 moves to win. The MCTS Player prioritized forking from the same node, connecting multiple nodes to the same node for multiple moves. An example can be seen in Figure 20. It is now the Opponent's turn and since the Player will win by either the (0,1) or (3,5) edge. The Player has forced a win through forking. The MCTS Opponent blocked the other player from winning the game by blocking a clique, an example of which is shown in Figure 19. The edge from 1 to 2 blocks a red clique consisting of (3,5,0). The game was ended by MCTS Player making a fork that forced an end to the game, where two possible edges would have ended the game.



(a) Fig. 19 Blocking



(b) Fig. 20 Forking

Strategy

When playing against itself, the DQN Player on $R(4)$ took 65 moves to win. Both DQN agents did not prioritize forking as much as MCTS agents did. More clusters of connections were present, with multiple separate forks from different nodes. With more moves, agents started to fork from a few nodes more often, creating multiple possible cliques with fewer nodes.

4 DISCUSSION

In this study, multiple types of reinforcement learning agents were trained on Ramsey games to determine both an optimal strategy and determine which model would be the least costly in finding a strategy. Agents were trained on Ramsey games for both the $R(3)$ and $R(4)$ variants. The $R(3)$ variant was played on a graph with 6 vertices and required a monochromatic fully connected group of 3 vertices, and the $R(4)$ variant was played on a graph with 18 vertices and required a monochromatic fully connected group of 4 vertices.

Agents were evaluated on their win rate against a random player, win rate while training, move time while training, average number of moves while training and memory usage while training. On $R(3)$ the top model (MCTS) had a win rate of 97% going first and second, and on $R(4)$ the top model (DQN) had a win rate of 62% and 60% going first and second respectively. The GQN model was predicted to have the highest win rate, however, it had the second highest win rate for $R(3)$ with all 3 GQN models having higher win rates than TQL and DQN. On $R(4)$, however, GQN had the lowest win rate. For win rate while training, a faster growing win rate represents a faster learning model. The fastest win rate growth for $R(3)$ was a GQN which was expected. On $R(4)$, the DQN model had the fastest win rate growth. Average number of moves was also measured and an agent with a higher number of moves is the most optimal. A higher number of moves is most optimal because when both agents have an optimal policy, they will try and force a tie by using as many moves as possible. An agent with a lower amount of moves could represent one agent type having a better strategy than the other, and as such using the least amount of moves to win. On $R(3)$, DQN had the most moves and on $R(4)$ GQN had the most moves. When evaluating strategy, the best performing models in terms of random player win rate were used. MCTS on $R(3)$ was found to prioritize connecting edges from the same node more than DQN on $R(4)$ did. Forking may have been a high performance strategy because it leads to wins more often than a simple strategy does. DQN on $R(4)$ also focused on connecting edges from the same node, but did so later in the game, and focused on forking from a few base nodes. The spreading out of base nodes may be due to larger graph size and the more complex fully connected group of 4 needed.

GQN was predicted to be the least costly model. Measurements of cost, the resources used to train, in this study were time taken to move and memory used. On both $R(3)$ and $R(4)$ GQN had the least memory usage, with TQL having the highest memory usage by about 1 order of magnitude on $R(3)$ and 2 orders of magnitude on $R(4)$. This is an expected result as TQL must store an entry for each state-action pair, while other models do not maintain values for state action pairs. In terms of time taken to move, TQL had the least time taken to move on $R(3)$ and DQN on $R(4)$. GQN was expected to have the least time to move. GQN's time taken to move was about 4 times the time taken to move for TQL and DQN on both $R(3)$ and $R(4)$. This could be due to a larger q-value network, because GQNs had more complicated networks than DQN, and TQL only had to store values. Overall, GQNs were the least costly because they have the least memory usage and while their time taken to move is not the least, the factor by which it is bigger is compensated for by the extremely low memory usage.

Overall, GQN models performed the best on $R(3)$ and DQN did so no $R(4)$ with MCTS showing high performance on $R(3)$ but not on $R(4)$. The win rate, cost and strategy of these models proves promising for finding better

strategies for Ramsey games. With more computing power, these models may be able to be used on higher Ramsey numbers of graphs and used to develop a more comprehensive strategy that can be expanded to Ramsey numbers as well. Other types of GNNs, such as those that are better on dynamic graphs may also have applications here. Using these improvements, we may able to develop a comprehensive strategy that may be able to more easily narrow bounds for Ramsey numbers.

ACKNOWLEDGMENTS

I would like to thank Dr. William Gasarch and Mr. Joshua Twitty for their mentorship.

REFERENCES

[1] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2017. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. *arXiv:1606.09375 [cs, stat]* (Feb. 2017). http://arxiv.org/abs/1606.09375 arXiv: 1606.09375.

[2] Maxim Egorov. [n. d.]. Multi-Agent Deep Reinforcement Learning. ([n. d.]), 8.

[3] W. Gasarch. [n. d.]. *Ramsey's Theorem on Graphs.* https://www.cs.umd.edu/users/gasarch/COURSES/858/S13/ramsey.pdf

[4] W. Gasarch. [n. d.]. *Small Ramsey Numbers, Lower Bounds On Ramsey Numbers: An Exposition by William Gasarch [Lecture.*

[5] Jiechuan Jiang, Chen Dun, Tiejun Huang, and Zongqing Lu. 2020. Graph Convolutional Reinforcement Learning. *arXiv:1810.09202 [cs, stat]* (Feb. 2020). http://arxiv.org/abs/1810.09202 arXiv: 1810.09202.

[6] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv:1609.02907 [cs, stat]* (Feb. 2017). http://arxiv.org/abs/1609.02907 arXiv: 1609.02907.

[7] Michael L Littman and Csaba Szepesvari. [n. d.]. A Generalized Reinforcement-Learning Model: Convergence and Applications. ([n. d.]), 19.

[8] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. (Oct. 2017). https://arxiv.org/abs/1710.10903v3

[9] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. 2018. Dynamic Graph CNN for Learning on Point Clouds. (Jan. 2018). https://arxiv.org/abs/1801.07829v2

[10] Qikui Zhu, Bo Du, and Pingkun Yan. 2020. Self-supervised Training of Graph Convolutional Networks. *arXiv:2006.02380 [cs]* (June 2020). http://arxiv.org/abs/2006.02380 arXiv: 2006.02380.