PLAYING RAMSEY GAMES USING A Q-LEARNING AGENT

TIANBO YANG ADVISOR: YUXIN ZHOU

Abstract. In this paper, a Q-learning agent is trained to play graph Ramsey games with the goal of verifying results for small Ramsey numbers already found in literature. An overview of the theory behind Ramsey games and the Q-learning algorithm will be provided before moving on to describe results. We then analyze what reinforcement learning can tell us about Ramsey theory and whether this method is applicable to larger Ramsey numbers.

1. Introduction

Finding new Ramsey numbers is one of the long-standing unsolved problems in combinatorics. Over the years, many approaches have been tried to tackle the problem using graphs. A prominent methodology has been to play Ramsey avoidance games, but given the limitations of human analysis and computational power, this method has been traditionally restricted to small Ramsey numbers. This paper tests the efficacy of reinforcement learning as a new tool of analysis. A Q-learning agent is trained to play graph Ramsey games. In sections 2 and 3, we provide relevant background on graph Ramsey theory and Q-learning. We then move to discuss the methodology by which we trained the algorithm in the fourth section. In the fifth section, we analyze in detail the performance of the model and what it can reveal to us about Ramsey numbers. Finally, we conclude by discussing the limitations of this methodology and future plans to upscale the capabilities of the model.

2. Graph Ramsey Theory

A popular analogy to derive some intuition for Ramsey numbers is to imagine guests at a party. Some guests know each other, some don't. The Ramsey number $R(m, \ell)$ is the minimum number of guests such that one of two things must be true: either m of those guests know each other, or ℓ of those guests do not know each other. Formally:

Date: 2024-04-20.

Definition 2.1. The *clique Ramsey number on two colors* $R(m, \ell)$ is the least positive integer n such that for any arbitrary partitioning of the edges of the complete graph K_n into two color classes $\mathcal{P}_1$ and $\mathcal{P}_2$, either the edge-induced subgraph $\langle \mathcal{P}_1 \rangle$ contains a graph isomorphic to K_m, or the edge-induced subgraph $\langle \mathcal{P}_2 \rangle$ contains a graph isomorphic to K_ℓ.

For example, the statement that $R(3, 3) = 6$ implies that no matter how the edges of the complete graph K_6 is partitioned into two color classes, there must exist a color such that edges of that color form a graph that contains a K_3 (known colloquially as a triangle). This statement will be proven later in this section. Ramsey numbers are not just limited to cliques and two color classes. We can extend this definition to any type of graph and any number of color classes to arrive at the generalized definition of Ramsey numbers.

Definition 2.2. [5] For a collection of graphs $M_1, M_2, \ldots, M_k$, the *Ramsey number* $R(M_1, M_2, \ldots, M_k)$ is the least positive integer n such that if $(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k)$ is an arbitrary partition of the edges of the complete graph K_n, then for some i, the edge-induced subgraph $\langle \mathcal{P}_i \rangle$ contains a graph isomorphic to M_i.

We say that $R(M_1, M_2, \ldots, M_k)$ is *symmetric* if $M_1 \cong M_2 \cong ... \cong M_k$. Otherwise, we call it an *asymmetric* Ramsey number.

In a graph setting, $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k$ are usually represented as k color classes, and so the partition $(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_k)$ is a k-coloring of the graph K_n. Thus, the problem of finding Ramsey numbers can be understood as the optimization problem of finding the smallest complete graph (in terms of the number of vertices) such that one of a set of subgraphs, each with their individual type and/or color, must exist.

However, it can be difficult to find an algorithm for determining the solution to this problem. One approach is to gamify it.

Definition 2.3. The *Ramsey avoidance game* for determining whether $n = R(M_1, M_2, \ldots, M_k)$ is a turned based game played on the complete graph K_n with initially uncolored edges between k players $P_1, P_2, \ldots, P_k$. Players are each associated with their own unique color and take turns choosing an edge to color in K_n. Player P_i loses when a subgraph of P_i's coloring is formed that is isomorphic to M_i. If K_n is completely filled and no player has lost yet, then the game is considered a draw.

With these game rules, the following theorem holds:

Theorem 2.4. *If $n \geq R(M_1, M_2, \ldots, M_k)$, then there does not exist a sequence of moves in the corresponding Ramsey avoidance game on n vertices that results in a draw.*

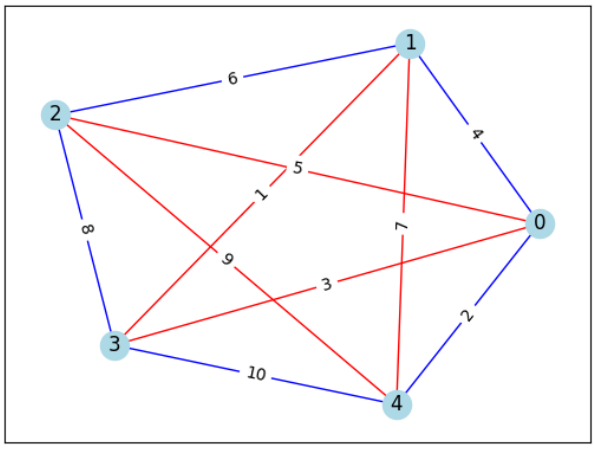

FIGURE 1. An example of a game on K_5 that resulted in a draw. This example was actually produced by a Q-learning agent playing with itself.

With this theorem, we can finally prove that $R(3,3) = 6$. Consider the following lemma:

Lemma 2.5. [11] $R(3,3) = 6$.

Proof. By definition, the Ramsey avoidance game for $R(3,3)$ is played on an edge-2-colored K_6, with the colors red and blue representing the 2 players respectively. Both players have the objective of not forming a monochromatic 3-edge subclique (also called a triangle) of the game graph in their own color. Take any vertex v of the game graph K_6. Since there are 5 edges connected to v, at least three of them will have the same color by the pidgeonhole principle. Without loss of generality, assume that this color is red. Consider the three vertices connected to v through these three edges; either one of the edges that connect two of these vertices is red (and then there is a red triangle with the edges connected to v), or all three edges that connect these three vertices are blue (and then there is a blue triangle). Either way, a draw is not possible. On the other hand, if we play on K_5, we do not have the situation that at least three edges connecting to the same vertex will have to have the same color. Then we can avoid creating a triangle of either color, as shown in Figure 1. Thus, six is the smallest number of vertices in the game graph such that either a red triangle must be formed or a blue triangle must be formed. □

From analytical methods similar to the one shown above, we can determine the values of other small Ramsey numbers. Below is a chart documenting the values of clique Ramsey numbers already found in literature.

r \ s	1	2	3	4	5	6	7	8	9	10
1	**1**	1	1	1	1	1	1	1	1	1
2		**2**	3	4	5	6	7	8	9	10
3			**6**	9	14	18	23	28	36	40–42
4				**18**	25[9]	36–40	49–58	59[13]–79	73–106	92–136
5					**43–48**	58–85	80–133	101–194	133–282	149[13]–381
6						**102–161**	115[13]–273	134[13]–427	183–656	204–949
7							**205–497**	219–840	252–1379	292–2134
8								**282–1532**	329–2683	343–4432
9									**565–6588**	581–12677
10										**798–23556**

FIGURE 2. Most known Ramsey numbers [4]

As mentioned above, Ramsey numbers can be defined for any kind of graph, not just limited to cliques. A well studied alternative to cliques is the cycle Ramsey number.

Definition 2.6. The Ramsey number $R(M_1, M_2, \ldots, M_k)$ is a called a *cycle Ramsey number* if the graphs $M_1, M_2, \ldots, M_k$ are all cycles.

Denote by C_i the cycle of length i. Since $K_3 = C_3$, it is fairly evident from Lemma 2.5 that $R(C_3, C_3) = R(3, 3) = 6$. Vclav Chvtal and Frank Harary further proved that $R(C_4, C_4) = 6$ [10]. Ramsey theorists have also discovered the following pattern regarding asymmetric cycle Ramsey numbers on two colors:

Theorem 2.7. [7]

$$R(C_n, C_m) = \begin{cases} 2n-1 & \text{for } 3 \le m \le n, m \text{ is odd}, (n,m) \ne (3,3) \\ n-1+\frac{m}{2} & \text{for } 4 \le m \le n, m \text{ and } n \text{ are even}, (n,m) \ne (4,4) \\ \max\{n-1+\frac{m}{2}, 2m-1\} & \text{for } 4 \le m < n, m \text{ is even and } n \text{ is odd} \end{cases}$$

Ramsey games provide a solid theoretical method for determining Ramsey numbers. In practice, however, these games quickly become computationally untractable for large n. To combat these limitations, we introduce reinforcement learning as a new analytical tool.

3. Q-LEARNING

Reinforcement learning is a machine learning paradigm in which a learner, called an agent, learns to interact with an environment by mapping situations (called states) to actions in order to maximize a goal defined by a reward function.

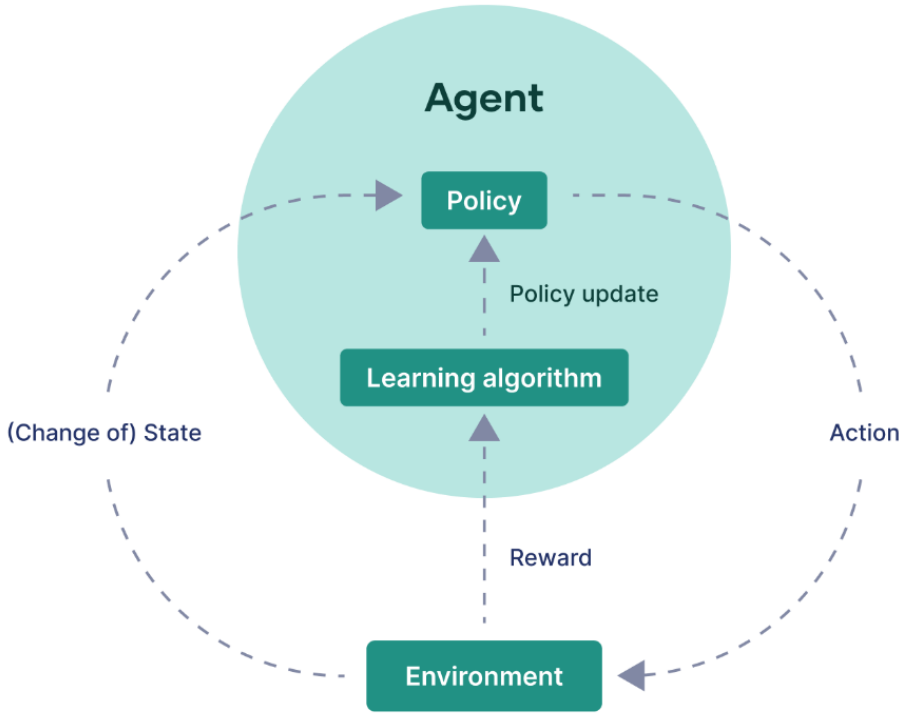

FIGURE 3. A typical reinforcement learning system [2]

An example of a very successful reinforcement algorithm is Q-learning developed by Christopher Watkins [6]. It is an incremental method for dynamic programming that works by successively improving its evaluations of particular actions at particular states [1]. Over the past decade, Q-learning algorithms have become adept at playing turn-based, finite-state games with a clear objective, such as chess, blackgammon, and go [6]. This aptitude motivated our selection of this particular algorithm, since Ramsey avoidance games are a clear example of that kind of game. Below, we provide a cursory description of the algorithmic foundations of Q-learning.

Consider a computational agent moving in some discrete, finite world (or set of states) X, choosing one from a finite collection of actions A at every step. The world constitutes a controlled Markov process with the agent as a controller. At

the i-th step, the agent is equipped to register a state $x_i \in X$ of the world. From x_i, there exists an action space $A_i \subseteq A$ consisting of the set of valid actions the agent can take. It can then choose an action $a_{i_j} \in A_i$ accordingly. The agent receives a probabilistic reward r_{i_j}, whose mean value $R_{x_i}(a_{i_j})$ depends only on the state and action. Choosing a_{i_j} results in the state of the world changing probabilistically to another state $x^+ \in X$. We denote this probability by $P_{x_i x^+}(a_{i_j})$.

The agent is tasked with determining an optimal policy (or course of action) that maximizes total discounted expected reward. By policy we mean some function ϕ that maps a state x_i to some action a_{i_j} taking it probabilistically to some new state x^+, essentially a function that defines the decision making of the agent. By discounted reward, we mean that rewards received t steps in the future are worth less than rewards received now, by a factor of λ^t, where λ is some fixed hyperparameter with a value between 0 and 1. Then immediately after taking an action that ϕ recommends, the agent expects to receive the reward $R_{x_i}(\phi(x_i))$ and move with probability $P_{x_i x^+}(\phi(x_i))$ to x^+ with some measure of expected future reward, called value.

Definition 3.1. [1] For a policy ϕ, the *action value* (or Q *value*) of an action a at state x_i is defined recursively as

$$Q^\phi(x_i, a) = R_{x_i}(a) + \lambda \sum_{x^+} P_{x_i x^+}(a) Q^\phi(x^+, \phi(x^+)).$$

Definition 3.2. [1] The *value* of state x_i under a policy ϕ is defined by

$$V^\phi(x_i) = Q^\phi(x_i, \phi(x_i)).$$

By the theory of dynamic programming, there exists at least one optimal policy ϕ^* such that

$$\phi^*(x_i) = a : \max(Q^{\phi^*}(x_i, a)).$$

Then

$$V^{\phi^*}(x_i) = \max_{a \in A_i}\{Q^{\phi^*}(x_i, a)\}.$$

Denote $Q^* := Q^{\phi^*}$ and $V^* := V^{\phi^*}$. We can calculate V^* and ϕ^* using dynamic programming provided that $R_{x_i}(a)$ and $P_{x_i x^+}(a)$ are known [1]. The Q-learner, then, is an algorithm to solve for ϕ^* in an efficient manner without initially knowing these values by estimating the action values.

Observe the difference between state value and action value. In state value, your immediate reward $R_{x_i}(\phi(x_i))$ is determined by the decision your policy chooses at state x_i. In action value, on the other hand, your immediate reward is determined by the executing the action in question a_{i_j} at state x_i, regardless of your policy. Policy is followed afterwards when considering expected future rewards.

In practice, computing action values directly using the recursive definition is too computationally expensive to be practical. As mentioned above, one popular way to reduce the computational complexity is to store the action values in a lookup table and use dynamic programming. Below, we describe the update rule for this process.

Algorithm 3.3. [1] **Q-learning** *is an incremental method, meaning that it happens in a sequence of distinct episodes, each corresponding with an action the agent takes. The initial action values $Q_0(x, a)$ for all $x \in X$ and $a \in A$ are usually randomly or arbitrarily set, and the algorithm updates them with each episode. Also defined before the training process is a learning factor α_i, which controls how much the action values are updated each episode. In the ith episode, the agent*

(1) *observes its current state x_i*
(2) *selects and performs an action a_i*
(3) *observes the subsequent state x_i^+*
(4) *receives an immediate payoff $r_i = R_{x_i}(a_i)$ for some predefined reward function R*
(5) *updates its Q_{i-1} values for all states $x \in X$ and actions $a \in A$ using a predetermined learning factor α_i by*

$$Q_i(x,a) = \begin{cases} (1-\alpha_i)Q_{i-1}(x,a) + \alpha_i(r_i + \lambda max_{b\in A}\left\{Q_{i-1}(x_i^+, b)\right\}) & \text{if } (x = x_i) \wedge (a = a_i) \\ Q_{i-1}(x,a) & \text{otherwise} \end{cases}$$

where $max_{b\in A}\left\{Q_{i-1}(x_i^+, b)\right\}$ represents the agent's estimation of the greatest possible future reward from going to state x_i^+.

Assuming that the $Q_i(x, a)$ is calculated using a look-up table representation, and that there is an infinite number of (not necessarily continuous) episodes for each starting state and action, the algorithm described above has the following convergence theorem:

Theorem 3.4. [1] *Let $i^j(x, a)$ be the episode index of the jth time that action a is tried in state x. Given bounded rewards $|r| \leq R$, learning rates $0 \leq \alpha_i < 1$, and*

$$\sum_{j=1}^{\infty} \alpha_{i^j(x,a)} = \infty, \quad \sum_{j=1}^{\infty} \left(\alpha_{i^j(x,a)}\right)^2 < \infty, \quad \forall\, x, a,$$

then $Q_i(x, a) \to Q^*(x, a)$ as $i \to \infty$ for all x, a, with probability 1.

4. Methodology

Say that we are interested in finding a lower bound for $R(M_1, M_2)$. Recall from Theorem 2.4 that the existence of a draw in a Ramsey avoidance game on a graph with n vertices implies that $R(M_1, M_2) > n$. Then, it follows that an algorithmic method for establishing the lower bounds of Ramsey numbers can be attained by finding whether a draw exists when playing a Ramsey avoidance game on a graph with n vertices. If a draw is found, we repeat the process on $n + 1$ vertices, and so on, until we can no longer find a draw. Thus, all we are missing is a suitably powerful algorithm for finding draws given an arbitrarily large number of vertices. In this paper, we will use Q-learning for this purpose.

Now we describe the implementation of the Q-learning algorithm for playing Ramsey avoidance games with two color classes (or "players"). As stated in section 3, Q-learning is an iterative process in which we define an environment, in our case, the Ramsey avoidance game, and have the agent play the game repeatedly, improving the action values associated with each action based on the outcomes from playing said action. To address issues with cooperation between players, single agent will make the decisions for both players. We employ a traditional lookup table implementation based on the principles of dynamic programming. Compared to other model classes such as neural networks, the advantage of using a lookup table lies in their interpretability and relatively transparent decision making. We also tried a neural network implementation using the library PyTorch, but preliminary testing indicated that the increased training time did not justify the single digit percentage improvements in accuracy, and so the idea was dropped.

The lookup table implementation involves using environmental parameters such as the size of the action space and the number of states to construct an array for keeping track of action values during the training process. This technique, known as dynamic programming, massively reduces the computational complexity the training algorithm since the action values of the previous episode can now be simply searched for in an array instead of computed recursively. Let n denote the number of vertices in the game graph. Then the number of possible actions at the start of the game is the same as the number of edges in the graph $\binom{n}{2}$. Thus, we can represent our action space as a $\binom{n}{2} \times 1$ array. On the other hand, the maximum possible number of states in our game is $2^{\binom{n}{2}}$. Thus, our lookup table can be represented as a $2^{\binom{n}{2}} \times \binom{n}{2}$ array. Denote the lookup table by Q and let $Q_i(x, a)$ be its value in row x and column a after the ith training game, representing the action value ascribed to taking action a at state x.

Here, we simplify our notation by equating an episode with a training game rather than with a single action. Recall from Algorithm 3.3 that i denotes the current episode count, and that episodes correspond to individual actions taken by the agent. In the Ramsey avoidance game, each action is only taken at most once per game since edges cannot be removed or colored more than once. Thus, action values are updated only once per game and there is no distinction between associating $Q_i(x, a)$ with the ith game or the ith action. From this point onwards, we equate the ith episode with the ith training game and specify when we are talking about individual actions.

During training, we want the agent to explore different strategies to minimize the chance it will become stuck with an suboptimal policy. This is usually accomplished by letting the agent sometimes play a random move instead of following its current policy, giving it a chance to discover a better policy. The exploration algorithm we employ is called ϵ-decay. In the ith training game, the agent has a probability ϵ_i chance to choose a random available action; otherwise, it will follow the current policy by choosing the action corresponding to the largest action value in the current lookup table. At the start of the training process, $\epsilon_0 = 1$ and decreases incrementally each episode. The effect of ϵ-decay is to let the agent explore a lot at the beginning of the training process (when exploration is most useful) and gradually adhere more to the policy in later games.

To remove illegal actions, we use a technique called action masking, where after each step, the column of action values representing the action the agent just took is set to a large negative value for the remainder of the game, thereby guarenteeing that the agent will not choose that action in future moves.

To check if the win condition of the game is fulfilled, we simply designate edges placed during odd steps as "red" and edges placed during even steps as "blue" and use the depth-first search algorithm to find if the graphs M_1 and M_2 are contained within the subgraphs formed by the respective color classes. If either is found, the agent receives a negative reward (-50) and the game ends with a loss. If neither is found, the game continues until step $\binom{n}{2}$, when if M_1 and M_2 are still not found, the game ends with a draw and the agent receives a positive reward (+50).

Other optimizations to the reward structure that were found to improve training quality include rewarding the agent for playing edges not adjacent to an already played edge of the same color and "blocking" the opposing player from forming M_1 or M_2 in one color by preemptively completing the graph using the other color. We also penalize the agent for playing edges that are adjacent to an already played edge of the same color to discourage M_1 and M_2 formation. These intermediate rewards encourage the agent to play longer games, discourage it from making moves that lead to a loss, and to "help" the other player. The table below

summarizes the rewards given to the agent when various conditions are observed at each state, subject to further optimization:

Condition	**Reward** r
Draw	+50
red player forms M_1	-50
blue player forms M_2	-50
"blocking" opponent from losing	+40
form larger cycle than M_1 and M_2	+30
play adjacent edge	-40
play non-adjacent edge	10
illegal move	-100
otherwise	0

TABLE 1. Reward Structure

The action values in the lookup table are updated according to Algorithm 3.3 after every game. In our implementation, we use the learning rate $\alpha = 0.1$ and the discount factor $\lambda = 0.95$. For some action a played from state x, we update action value $Q_i(x, a)$ by the update rule

$$Q_i(x, a) = 0.9Q_{i-1}(x, a) + 0.1(r + 0.95\text{max}_{b\in A}\left\{Q_{i-1}(x_i^+, b)\right\})$$

where r is the reward given to the agent for making action a according to Table 1.

Training involves the agent playing Ramsey avoidance games repeatedly to gradually improve the policy. In the next section, we will discuss and analyze the resulting models.

5. Analysis

We now analyze the performance of our Q-learning model when solving for various Ramsey numbers using the Ramsey avoidance game. We focused on cycle Ramsey numbers since Theorem 2.7 allows us to check our observations against the theoretical true value of the Ramsey number. First, we clarify some notation. When speaking of a game $R(x, y) > n$, we mean a Ramsey avoidance game on the graph K_n in which player 1 must avoid forming a x-cycle and player 2 must avoid forming a y-cycle. If a draw in this game can be found, then we have successfully established that $R(C_x, C_y) > n$. The Training draw rate refers to the proportion of training games that resulted in a draw. While not a particularly significant performance metric, the training draw rate can still be interesting to look at since a low value can indicate that the model is struggling during the learning process.

The testing draw rate refers to the proportion of games that result in a draw from conducting 1000 testing games of the model under the specified conditions. This statistic will be the primary performance metric we will be using to evaluate our model.

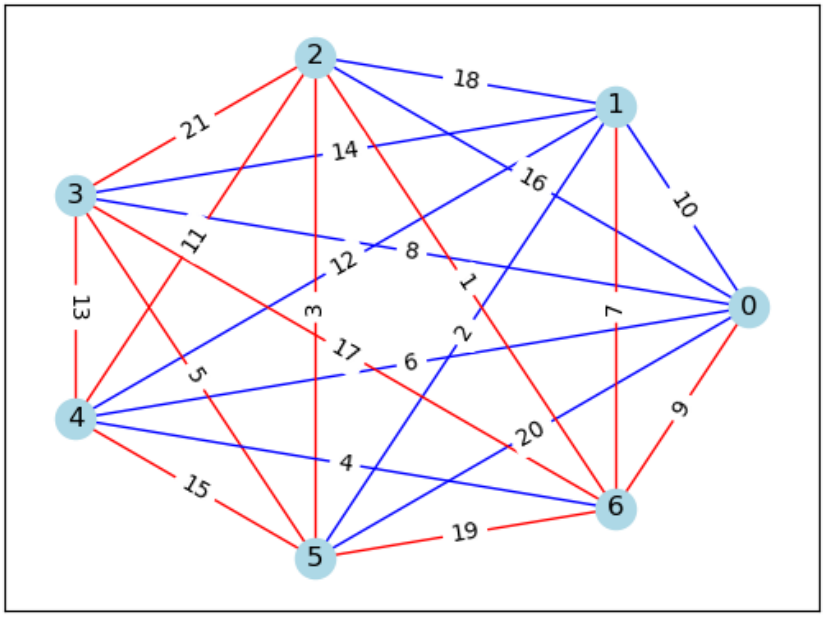

FIGURE 4. A Ramsey avoidance game for finding if $R(6,6) > 7$ that resulted in a draw.

To establish a baseline for comparison, we first play the Ramsey avoidance game using a random algorithm in which the actions are always decided by random number generator. We run this random algorithm 4 times for good measure on each Ramsey avoidance game of interest to obtain the values shown in Table 2.

We then train our Q-learning agent with 1000, 2000, 5000, and 10000 training games. Table 3 depicts the resulting draw rates from 1000 testing games in which the agent's actions strictly follow the policy laid out by the actions values obtained from training.

A cursory comparison between Tables 2 and 3 shows that our Q-learning model performs substantially better than the random algorithm. However, in order to generalize our model to be able to solve for larger Ramsey numbers in the future, we care about its ability to find draws in general, not its ability to find any specific draw. In other words, we want the model to be able to adapt to changing

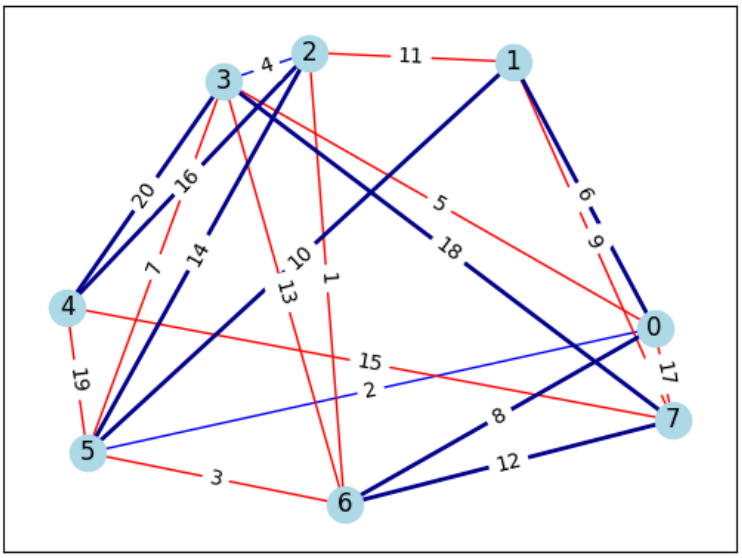

FIGURE 5. A Ramsey avoidance game for finding if $R(4,8) > 8$ that resulted in player 2 (blue) losing.

TABLE 2. Random Algorithm

Game	Testing Draw Rate
$R(3,3) > 5$	0.0490, 0.0440, 0.0480, 0.0460
$R(4,4) > 5$	0.2710, 0.2940, 0.2650, 0.3050
$R(3,4) > 6$	0.0130, 0.0160, 0.0130, 0.0210
$R(6,4) > 6$	0.0870, 0.0870, 0.0820, 0.0740
$R(6,6) > 7$	0.0480, 0.0400, 0.0400, 0.0400

circumstances instead of only recreating the same exact draw over and over again. A good way to test this ability is to introduce an element of randomness into the testing process. In one test, we let each action have a 10% chance of being decided randomly instead of following the policy. In the other test, we let the first move of each player be decided randomly and follow the policy from there on.

To better evaluate the performance of the model under the various testing conditions, we can plot test accuracy vs the number of training games for individual

TABLE 3. Q-learning Model (Follow Policy During Testing)

Game	Training Games	Training Draw Rate	Testing Draw Rate
$R(3,3) > 5$	1000	0.8010	0.9430
$R(3,3) > 5$	2000	0.0170	0.0050
$R(3,3) > 5$	5000	0.9170	0.9510
$R(3,3) > 5$	10000	0.5159	0.0060
$R(4,4) > 5$	1000	0.7540	0.9680
$R(4,4) > 5$	2000	0.8595	0.9710
$R(4,4) > 5$	5000	0.9562	0.9700
$R(4,4) > 5$	10000	0.5664	0.0150
$R(3,4) > 6$	1000	0.0320	0.0020
$R(3,4) > 6$	2000	0.8835	0.9310
$R(3,4) > 6$	5000	0.0022	0.0060
$R(3,4) > 6$	10000	0.0616	0.0110
$R(6,4) > 6$	1000	0.0110	0.0070
$R(6,4) > 6$	2000	0.0120	0.0070
$R(6,4) > 6$	5000	0.0204	0.0070
$R(6,4) > 6$	10000	0.1246	0.9410
$R(6,6) > 7$	1000	0.0000	0.0000
$R(6,6) > 7$	2000	0.0005	0.0000
$R(6,6) > 7$	5000	0.0472	0.0150
$R(6,6) > 7$	10000	0.1201	0.0150

games. As shown in Figure 6, the model's testing draw rate suffers heavily when randomness is introduced to the testing process. This suggests that the agent struggles to adapt to unfamiliar situations. Another interesting observation is that for any given number of vertices, there seems to be an optimal number of training games for the best testing performance. Training substantially more than this optimal number causes the model to overfit and the test accuracy to plummet. From the plots, we can see that for games on 5 vertices, the optimal number of training games seems to be less than 5000, for 6 vertices, the optimal number seems to be around 10,000, and for 7 vertices, we need more than 10000 training games, perhaps 15000 or 20000. We are currently in the process of training for larger numbers of games to verify this hypothesis.

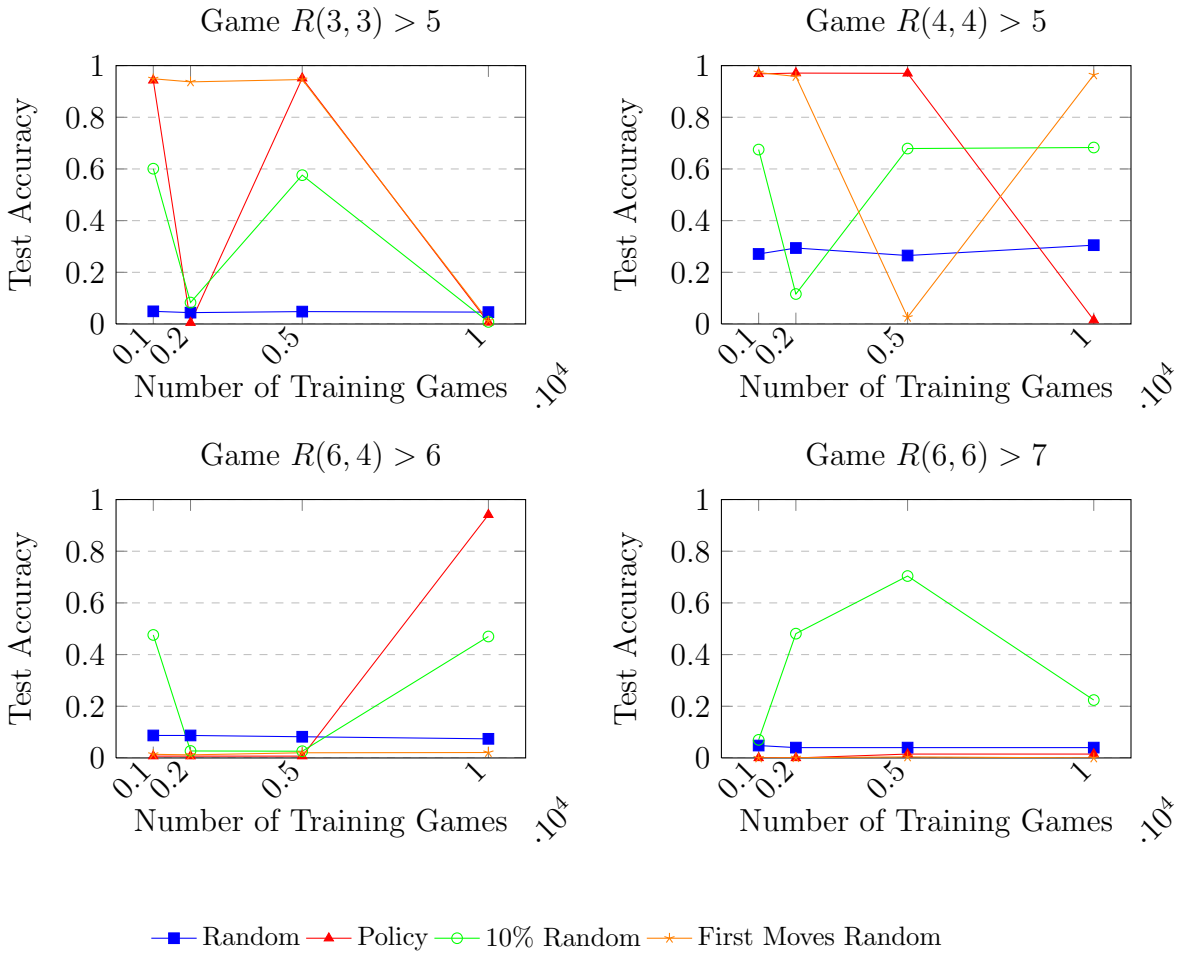

FIGURE 6. Testing Draw Rate vs Number of Training Games for select Ramsey avoidance games of interest

6. Further Research

Our Q-learning model and its subsequent testing successfully establishes reinforcement learning as a viable method for computationally establishing the lower bounds of Ramsey numbers. Under normal testing conditions, our model demonstrated its ability learn to find draws in Ramsey avoidance games on small numbers of vertices, while significantly outperforming the random algorithm. However, at the same time, testing exposes some of the limitations of reinforcement learning. We already mention the problem of failing to adapt to unfamiliar situations due to the agent's tendency to memorize and reconstruct draws it has

already seen before. To address this issue, we have tried to introduce more randomness into to training process to make memorization an unviable strategy, but the results were highly inconsistent. The agent failed to learn any coherent strategy from such a volatile and unpredictable environment. Experimenting with ways to improve the model's resistance to random noise will be one of our key directions for future research.

Another area that has already undergone substantial improvement since the completion of the iteration of the model showcased in this paper is its ability to train on larger numbers of vertices. Due to memory constraints on our computational resources, we were only able to train games on 7 vertices or less. Our environment contained many redundant states that can be eliminated using methods such as vertex permutation. Through such optimizations to our environment as well as the implementation of dynamic memory allocation, we were able to substantially increase the number of vertices the agent can play on to at least 9. We have not yet determined what our new upper limit is.

Training on larger numbers of vertices reveals the final limitation of reinforcement learning; since the proportion of games that are draws decreases for larger numbers of vertices, it becomes increasingly difficult to find draws. The model will need more and more training games in order to produce results. This problem will be the most difficult to overcome in the long term since there is only a finite amount of computational resources that can possibly be mustered for this project. Theoretically, there may be an upper limit to the size of the Ramsey avoidance game the Q-learning agent can solve.

References

[1] Christopher Watkins and Peter Dayan. "Q-learning". *Machine Learning*, vol. 8, 1992, pg. 279-292. DOI: https://doi.org/10.1007/BF00992698.

[2] Kassiani Nikolopoulou. "Easy Introduction to Reinforcement Learning". Scribbr, Aug. 2023. URL: https://www.scribbr.com/ai-tools/reinforcement-learning/.

[3] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning: State-of-the-art.* Springer, 2012.

[4] Neil Sloane et al. "The On-line Encyclopedia of Integer Sequences". 2003. arXiv: https://arxiv.org/abs/math/0312448.

[5] Paul Erdos et al. "Generalized Ramsey Theory for Multiple Colors". *Journal of Combinatorial Theory*, Series B, vol. 20 (3), 1976, pg. 250-264.

[6] Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction.* 2nd edition, The MIT Press, 2018.

[7] Stanislaw Radziszowski. "Ramsey Numbers Involving Cycles". *Progress in Mathematics*, vol. 285, Springer, 2011.

[8] Stanislaw Radziszowski. "Small Ramsey Numbers". *The Electronic Journal of Combinatorics*, 2021. DOI: https://doi.org/10.37236/21.

[9] Tom Needham. "Introduction to Applied Algebraic Topology". *Course Notes*, 2019, pg. 30-107. URL: https://research.math.osu.edu/tgda/courses/math-4570/LectureNotes.pdf.

[10] Vclav Chvtal and Frank Harary. "Generalized Ramsey Theory for Graphs. II. Small Diagonal Numbers". *Proceedings of the American Mathematical Society*, vol. 32, no. 2, 1972, pg. 389-394. DOI: https://doi.org/10.2307/2037824.

[11] Wolfgang Slany. "Graph Ramsey Games". *DBAI Techinical Report*. Institut für Informationssysteme Abteilung Datenbanken und Artificial Intelligence, 1999. no. DBAI-TR-99-34. arXiv: https://arxiv.org/abs/cs/9911004.