Comparison of AI Algorithms for Playing Square Stacker from Coolmath Games

Dan Oates
Worcester Polytechnic Institute
Worcester, MA, USA
doates@wpi.edu

Jesse d'Almeida
Worcester Polytechnic Institute
Worcester, MA, USA
jfdalmeida@wpi.edu

Owen Smith
Worcester Polytechnic Institute
Worcester, MA, USA
ocsmith@wpi.edu

Tabitha Gibbs
Worcester Polytechnic Institute
Worcester, MA, USA
tgibbs@wpi.edu

***Abstract*—In recent decades there has been an explosion of AI playing human games with superhuman capability, from board games such as Chess and Go to video games such as Atari and Tetris. This project investigates and compares algorithms for playing the video game Square Stacker from Coolmath games, including Deep-Q Networks, search algorithms, and memory-based agents. Square Stacker is unique due to its highly stochastic state space, where one quarter of the state is randomized every three moves. The Deep-Q Network failed to perform better than the baseline random agent, but a depth-limited random search agent was able to significantly outperform human players on average. This project reveals the utility of repeated simulation-based search in environments where the future is highly unpredictable.**

1. Introduction

With the massive popularity of games and the rise of video games in recent history, it is unsurprising that many researchers have spent countless hours exploring the application of artificial intelligence to these games. From the more classic Chess [3], Go [1], and even Hide and Seek [2] to the relatively newer Atari [4] and Tetris [5] games, researchers have used games of all kinds to explore frontiers of artificial intelligence. Games provide a simplified repeatable environment with far fewer unwieldy variables than the real world. So, just as many equations and models simplify and approximate the physics of the world but still provide useful results, computer scientists can use games as a testing ground for artificial intelligence.

This project provides a comparative analysis of algorithms for playing Square Stacker from Coolmath games. With these rules in mind, the comparative analysis of the implemented algorithms looks to weigh score maximization against computational complexity.

Square Stacker provides unique challenges for artificial intelligence in its extremely large state space that has an upper limit of approximately 2.65×10^{30} game states. Additionally, the game has a highly stochastic environment as one quarter of the game state (the playable pieces) is randomized every three moves. On the implementation side, the team struggled to become familiar enough with the nuances of Deep-Q Network design techniques to construct an effective model to train. Another challenge of Square Stacker is its lesser popularity. Because of this, there are no open source simulation implementations of the game, so the team had to first build an emulator from scratch.

This comparative analysis of game-solving agents provides a useful benchmark for further exploration of ways to optimally traverse the Square Stacker state space. The results gathered in this inquiry can guide those that may continue the work or those that may be attempting to investigate similarly stochastic and deep state spaces. While humans can develop rules-of-thumb for playing the game (such as spreading out colors and leaving many spaces empty), AI has the potential to automatically discover even more optimal strategies and solutions. From there, one can learn how to apply the same AI principles to a broader spectrum of problems, both real and virtual.

2. Background

2.1. Game Rules

The Square Stacker board consists of a 3x3 grid, each with small, medium, and large tile spaces, and three playable pieces on the right side, as shown in Figure 1.

Each tile space can be either empty or contain a tile of one of six colors: red, green, blue, yellow, orange, or purple. The objective of the game is to maximize score by dragging game pieces from the right onto empty spaces in the grid and clearing colors in sets of three. Pieces can be cleared in a line (i.e. via rows, columns, and diagonals) and in single spaces. Each line match scores three points while a single space of one color scores 5 points. There is an additional bonus multiplier included for scoring during multiple moves in a row. The game begins with an empty grid and three playable pieces. Every time the three pieces are played, three more are generated at random. Initially, only a subset of the six colors are generated, and each playable piece consists of only a small, medium, or large tile. However, pieces containing more colors and tiles of two sizes begin to appear more often as the player's score increases. The game ends when there are no possible moves for the player.

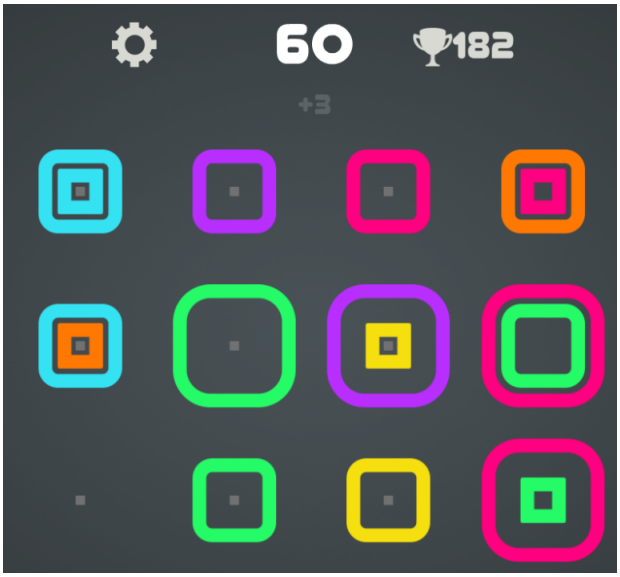

Figure 1. **Square Stacker Game Board Layout.** *The three pieces on the right can be placed on the non-overlapping tiles on the left.*

2.2. Deep-Q Networks

Deep-Q Networks (DQNs) are a subset of Q-learning, which is a subset of reinforcement learning. Reinforcement learning problems can be formalized as Markov Decision Processes, which require the task space to have a finite number of states s_t and actions a_t, a state-transition model $s_{t+1} = f(s_t, a_t)$, and a reward function $r(s_t, a_t)$. During Q-learning, an agent incrementally learns a utility function $Q(s_t, a_t)$ which is a weighted sum of the immediate reward $r(s_t, a_t)$ and the maximum future reward from $r(s_{t+1}, a_{t+1})$. The utility function Q is learned by performing a mix of optimal and random actions within the task space, where the optimal action is that which maximizes the current estimate of Q. Random actions are included to encourage exploration and avoid convergence to sub-optimal local maxima of Q, and are executed with probability ϵ. Estimates of Q are updated for each experience via the recurrent equation:

$$Q'(s_t, a_t) = (1 - \alpha)Q(s_t, a_t) + \alpha(r(s_t, a_t) + \gamma max(Q(s_{t+1}, a_{t+1}))) \quad (1)$$

Where α is the learning rate and γ is the discount factor, representing the relative importance of predicted future rewards of a given action [6].

Q-learning has two common implementations: Q-tables and DQNs. Q-tables store the Q-value of every possible action for every possible state in a lookup table. This is an efficient solution for small state spaces and easy to implement. However, Square Stacker and Q-tables both suffer from the curse of dimensionality. The board is a 36-dimensional discrete state (3 x 4 grid of 3 tiles each) each of which has 7 possible colors (including empty), resulting in an upper bound of approximately 2.65×10^{30} unique states, which is impossible to store on any modern computer. Additionally, many of these states are highly similar, and thus should elicit a similar action choice from the agent without explicit instruction to do so. DQNs (such as those used in [4]) attempt to solve the curse of dimensionality by replacing the explicit Q-table with a deep neural network, which takes the state vector as an input and outputs the predicted Q-values for each action. This allows for the Q-function to generalize and make decisions without explicitly experiencing every possible game state. In practice, the Q-network is trained in batches of experiences in order to promote generalization and improve network weight convergence.

While Q-learning is applicable and provably convergent for stochastic systems [6], it is limited in its practical training effectiveness in stochastic systems such as Square Stacker. The network begins as a random Q-function with no domain-specific knowledge, and the researchers are highly limited in computational resources for training the network. Thus, if the system is too random, then the (relatively) small experience training batches will not be informative enough for the network to converge to an optimal strategy.

2.3. Short-Term Memory Agent

A simple memory agent is a variation on a simple reflex agent as described by Russell and Norvig in "Artificial Intelligence: A Modern Approach" [8]. A simple reflex agent is any agent that uses solely conditional rules applied to its current state to make decisions. The simple memory agent differs as it uses conditional rules based on its current and previous states. This combination allows the agent to make more informed decisions which is particularly useful in creating sequential rules to describe game strategies.

2.4. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an improvement on the popular minimax algorithm, which is a decision rule used in games. One of the first big debuts of MCTS was in Google's AlphaGo agent, which utilized a neural net on top of MCTS to beat Lee Sedol and become a Go master. MCTS is most beneficial in games with large branching factors and where turns have a time limit. This is because MCTS is anytime, meaning that no matter when the algorithm is stopped, it will give an answer. It is also a-heuristic, unlike minimax, which means that no domain-specific knowledge is needed to determine the strength of a game state [7].

As shown in Figure 2, Monte Carlo Tree Search works by cycling through 4 main steps: selection, expansion, simulation, and back-propagation.

MCTS is given a game state as the root node with the goal to determine the best next move. During selection, the algorithm starts at the root node and works its way down the children until a leaf (node without children) is reached. It then determines which child to go to via a function called the Upper Confidence Bound Tree (UCBT):

$$\frac{\omega_i}{s_i} + c\sqrt{\frac{\log s_p}{s_i}} \quad (2)$$

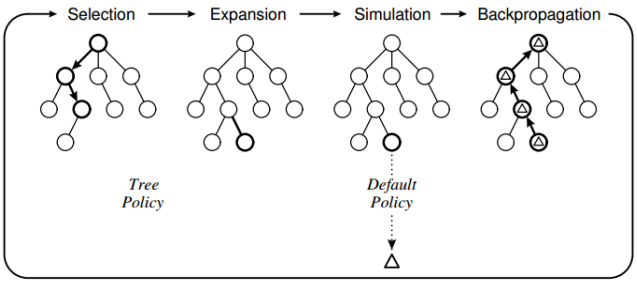

Figure 2. **Monte Carlo Tree Search Steps.** *MCTS can be broken down into four steps for evaluating the next best move: Selection, Exploration, Simulation, Back-propagation*

The function consists of the sum of two terms: exploitation, and exploration. Here, w_i is the score of the node, and s_i is how many times it has been traversed. The first term is exploitation and increases with large game scores, favoring nodes that have good results. s_p is how many times the parent node has been visited and c is a constant called the exploration parameter. This second term is the exploration term, favoring nodes that have not been visited as often.

Once a leaf node is reached, the successive child states are determined for future traversal in the expansion step. In the simulation step, the game is randomly played from the leaf node out to a terminal state. Once reached, the visited nodes (from leaf through its parents up to the root) are all updated with the new score. This cycle continues until some time or computational limit is reached, in which case the UCBT function is applied one last time to the root's children to make a decision on which move to make.

3. Methodology

Each of the agents developed in this investigation were created to explore the application of AI to a discrete, stochastic game state space. This was accomplished by developing a game emulator, testing two baseline agents, and evaluating each agent based on min, mean, and max scores from repeated testing in comparison to two baseline agents: a random agent and human players.

3.1. Game Emulator

The researchers unsuccessfully attempted contact the original developer of Square Stacker to get the game source code. As a result, an inexact emulator of the game was written in Python. Python was selected due to its ease of rapid-prototyping and compatibility with TensorFlow for DQN development. Because exact game rules were unavailable, some simplifying assumptions were made. First, it was assumed that tiles were generated uniformly at random from all possible colors and sizes. Second, the generation of multi-color pieces was left out, as their appearances in the real game were highly uncommon and with unclear origins. Finally, during play, the researchers found that the game never generated three impossible-to-play tiles for the player. This inductively-observed rule was also not implemented in the emulator due to uncertainty of its truth and the design complexity involved in implementing it.

3.2. Baseline Agents

Two agents were used as baselines for evaluating the performance of the agents tested. First, a random agent was designed to select each move at random uniformly from the set of all legal moves. Second, a human agent (two of the researchers) played the game and recorded their scores. The random agent's scores were used as a threshold to determine if other agents had any statistically-significant rationality, while the latter was used to determine if the agents played more intelligently than a human.

3.3. Deep-Q Network

The DQN model was implemented with two densely-connected hidden layers of size 128 with hard RELU activations. The input layer consisted of the colors of all 36 playable and non-playable tile spaces, the current score, and the current combo (number of scores in-a-row). Each color was represented with 7 disjoint binary inputs, one indicating emptiness, and the other 6 indicating the presence of each of the six colors. The output layer was a 27-dimensional vector indicating the predicted Q-value of all 27 theoretically possible moves (moving 1 of 3 pieces to 1 of 9 locations).

A variety of discount factors, epsilon values, training batch sizes, and other network structures were tested and all yielded similar results. The final parameters used were a learning rate of 1 (i.e. no reference to past Q-values) and a discount factor of zero (i.e. no reference to future Q-values). These decisions were made in an attempt to minimize feedback noise due to the highly stochastic state space. The probability of making random moves ϵ was selected to be a modest 0.05. The network was trained in experience batches of 5000 games 20 times in a row, resulting in a total of 100000 games of experience. More training than this was unrealistic given the limited computational resources available.

The DQN generates the predicted Q values for all 27 theoretical moves for a given game state. However, in most game states, at least one of these moves is impossible due to tile overlaps. Thus, the DQN agent selects that move which maximizes Q only within the subset of legal moves. If no moves are legal, then the game terminates.

3.4. Short-Term Memory Agent

The Short-Term Memory Agent was utilized in this investigation to explore a simple agent implementation. It attempted to bring some order to the stochastic environment by providing the agent with an action memory of variable length. With this information, the agent could select previous moves contained in its memory that match the color of one of the current playable pieces and place them in valid spaces

adjacent to the remembered location. The agent's percepts consisted of the size of the board, the playable pieces, and valid move locations. Its actions were placing a piece in any valid locations on the game board. A remembered move consisted of a play location in the board's grid and a color. Algorithm 1 details the agent's process for move selection. This agent was tested over 50000 games with memory lengths of 3, 10, and 25. At the end of each test, the agent's scores were recorded.

Algorithm 1 Short-Term Memory Move

```
Input: Game State (game)
Output: Move
  pieces = game.getPieces()
  if aPieceMatchMemory(game) then
    vaildMoves = game.getValidMoves()
    memMatch = getMatch(pieces)
    playableAdjacent = validAdjacents(validMoves, mem-
    Match)
    if playableAdjacent is NULL then
      return makeRandomMove(game)
    else
      return playableAdjacent
    end if
  else
    return makeRandomMove(game)
  end if
```

3.5. Graph Search Agents

Graph search agents by nature are uninformed, and therefore rely only on score feedback from tests when making a decision. As a result, the graph search approaches tested used systems of repeated random actions and comparisons to estimate the next move which maximizes long-term score. Each search algorithm was parameterized by search depth and/or number of games simulated per possible next move. Computational complexity of these algorithms was bench-marked by the number of moves simulated per actual move decision.

3.5.1. Random Search. For each possible next move, the random graph search (RGS) agent makes the move then simulates N games of entirely random moves until loss, recording the mean score of these games. The next move with the highest mean score from random play is selected.

3.5.2. Exhaustive Search. The exhaustive graph search (EGS) agent recursively simulates every possible move once out to a search depth D for each possible next move. The next move whose child yielded the highest score of all children is selected.

3.5.3. Depth-Limited Random Search. The depth-limited random graph search (DLRGS) agent combines the parameters N and D from RGS and EGS, respectively. The DLRGS agent simulates N games of D sequential moves for each possible next move, and selects the next move with the highest mean score of its children. This algorithm aims to minimize reliance on unpredictable states in the far future while also not exhaustively exploring all local states, ideally increasing efficiency in comparison to its predecessors.

3.6. Monte Carlo Tree Search

Since MCTS is a-heuristic, there is no training involved. Instead the only parameters that can be changed are the exploration parameter c and the limit on how long it has to determine the next move. In this case, since there is no second player and no real time constraint, the limit was chosen to be the amount of simulation cycles it went through for each move. Initially, the simulation limit was set to 200 simulations per move (SPM), and ran for 10 games to test the algorithm. Once it was verified, tests of 50, 100, and 150 SPM were run for 100 games each. The pseudocode of the algorithm is provided in Algorithm 2.

Algorithm 2 Monte Carlo Tree Search

```
Input: Root Game State
Output: Best Move
  for number of simulations do
    leaf = selection-and-expansion(root)
    result = simulation(leaf)
    backpropagate(leaf, result)
  end for
  return best-child(root)
```

3.7. Assessment Protocols

All agents were tested by playing a fixed number of games and recording the final scores of each game. Summary statistics were generated for brevity, namely the min, max, mean, and standard deviation of the scores. The graph search agents, each of which has at least one algorithm parameter, were evaluated similarly over multiple parameter combinations, and compared to one another by both score and the distribution of move search counts before making a final decision for each move. The move search count parameter acts as an indicator of time complexity, and thus these algorithms were also compared in terms of efficiency (mean score per mean move search count).

4. Results

4.1. Baseline Agents

The score statistics for the baseline agents (random and human) are shown in Table 1:

TABLE 1: **Baseline Agent Score Statistics.** *The random agent selected moves at random uniformly from the set of possible moves. The human agent consisted of two researchers playing the game and recording their scores. "Stdev" is short for "standard deviation".*

Agent	Game Count	Score Statistics			
		Min	Max	Mean	Stdev
Random	10000	0	188	19	20
Human	20	86	1694	454	428

The distribution of scores for the random and human agents are shown in Figures 3 and 4, respectively.

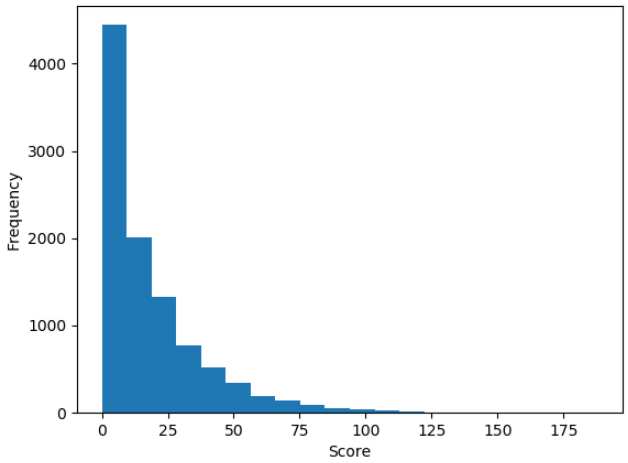

Figure 3. **Random Agent Scores.** *Distribution of scores of the random agent over 10000 games.*

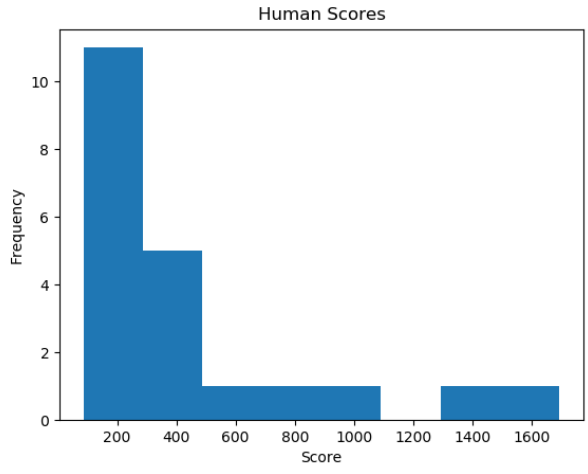

Figure 4. **Human Agent Scores.** *Distribution of scores of the human agent over 20 games.*

4.2. Deep-Q Network

The DQN computed the min, max, and mean score of every 100 games over its 100000-game training session. The results are shown in Figure 5.

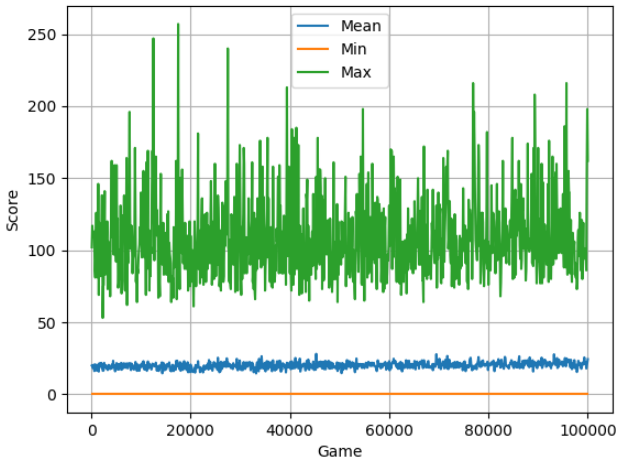

Figure 5. **DQN Training Score Progression.** *The mean, min, and max scores of every set of 100 games during DQN training.*

The network failed to show any statistically significant improvement over its training session. The mean score was barely above that of the random agent and remained stagnant for the duration of training. Additionally, the agent always scored zero at least once in every 100 games.

After training, the agent was tested for 1000 games without any additional training. This resulted in a score range of 0 to 223, with a mean score of 21 and standard deviation of 21. The distribution of DQN agent scores for the 1000 games is shown in Figure 6.

The distribution of scores for the DQN agent is highly similar to that of the random agent. The DQN scored only 10% higher than the random agent on average, and far below the human agent.

4.3. Short-Term Memory Agent

The memory-based agent was assessed by average score across a number of games, the associated standard deviation, and a graph of the game scores. Tests were run on agents with memory lengths of 3, 10, and 25 previous moves over 50000 games.

The agent performed similarly across all different variations of the test. The agent's performance had a consistent mean of 19 points, as shown in Figure 7 and Table 2.

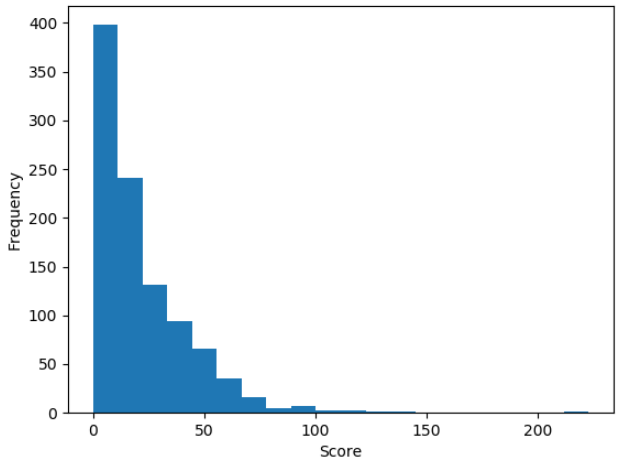

Figure 6. **DQN Agent Scores.** *Distribution of scores of the DQN agent over 1000 games after being trained for 100000 games.*

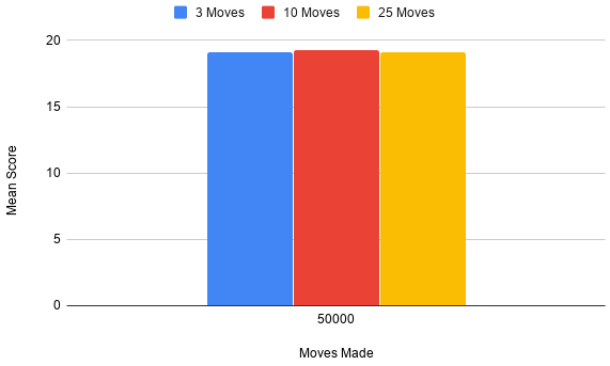

Figure 7. **The Mean Scores of the Memory Agent.** *The agent's performance did not change when the length of its move memory was varied.*

TABLE 2: **Score Statistics of Memory Agent.** *The memory agent was tested with three memory lengths and summarizing statistics were taken at different completed game intervals.*

Memory Length	Game Count	Score Statistics			
		Min	Max	Mean	Stdev
3	50,000	0	248	19	20
10	50,000	0	219	19	20
25	50,000	0	269	19	20

This agent did not improve on the performance of the baseline random agent in any variation of its tests.

4.4. Graph Search Agents

4.4.1. Random Search. The RGS agent was tested using 1, 2, and 3 games per move (GPM) for 1000 games each. As seen in Figure 8, the mean scores increased as the GPM value increased.

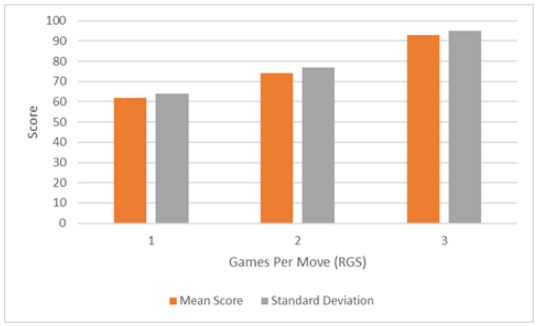

Figure 8. **Random Graph Search Score Results.** *The mean and standard deviation of scores over 1000 games for search depth values of 1, 2, and 3 using the random graph search agent.*

The distribution of scores and moves searched per decision for the RGS agent simulating 3 games per move are shown in Figures 9 and 10, respectively.

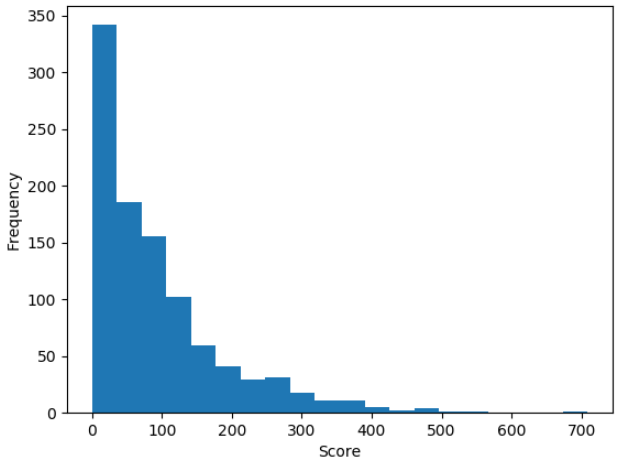

Figure 9. **RGS Agent Scores.** *Distribution of scores of the RGS agent simulating 3 games per move over 1000 games.*

4.4.2. Exhaustive Search. The exhaustive search agent showed much better overall scores than the random search agent. Search depth values of 1 and 2 were tested for 1000 games each, while a search depth of 3 was only tested for 500 games due to the long duration of the games. The mean score did increase as search depth increased, but the improvement nearly flattened after a search depth of 2, as shown in Figure 11.

The distribution of scores and moves searched per decision for the EGS agent with a search depth of 3 moves are shown in Figures 12 and 13, respectively.

4.4.3. Depth-Limited Random Search. Depth limited random graph search showed the best results out of the three graph search methods. This method combined the parameters and strategies from the the exhaustive and random

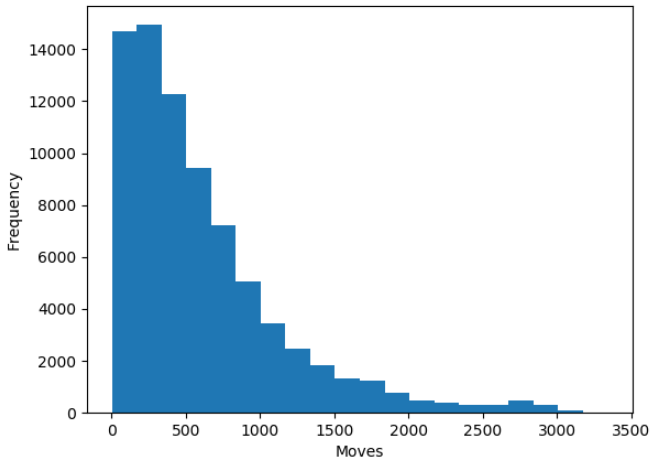

Figure 10. **RGS Agent Moves Searched.** *Distribution of moves searched per decision by the RGS agent simulating 3 games per move over 1000 games.*

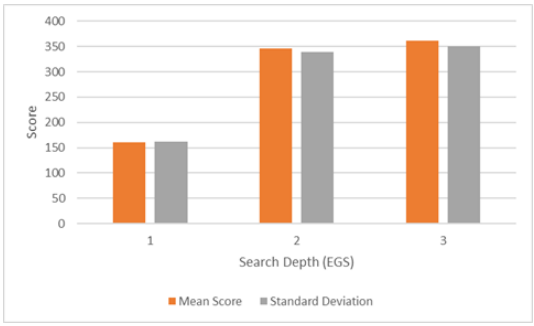

Figure 11. **Exhaustive Graph Search Score Results.** *The mean and standard deviation of scores for search depth values of 1, 2, and 3 using the exhaustive graph search agent.*

search agents. Search depth values of 1, 2, and 3 were tested, and for each search depth value, games per move values of 3, 5, and 10 were tested for 1000 games each. The results have been broken down by search depth (Figure 14) and by games per move (Figure 15). The trends in the search depth graph are nearly identical to that of exhaustive search, and the trends in the games per move graph are nearly identical to those in random search graph. This further shows that increasing the search depth past 2 has minimal effect on the scores, but increasing the games per move has a larger positive impact on the scores.

The distribution of scores and moves searched per decision for the DLRGS agent with a search depth of 3 moves and simulating 10 games per move are shown in Figures 16 and 17, respectively.

4.4.4. Comparison of Graph Search Agents. To compare the efficiencies of each version of the graph search algorithm, the mean scores and mean number of moves selected by the agents were compared in Figure 18. Each data point

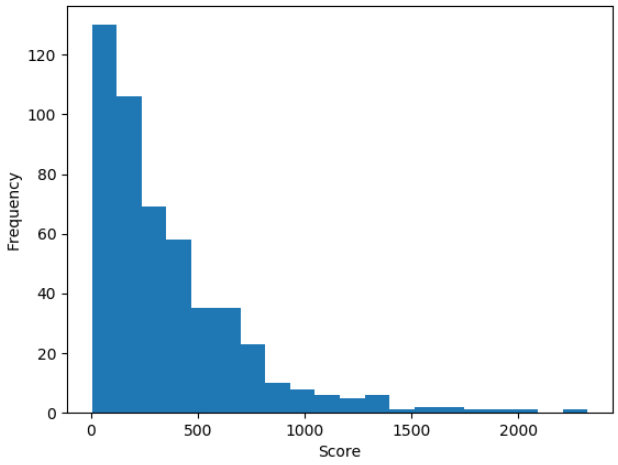

Figure 12. **EGS Agent Scores.** *Distribution of scores of the EGS agent with a search depth of 3 moves over 500 games.*

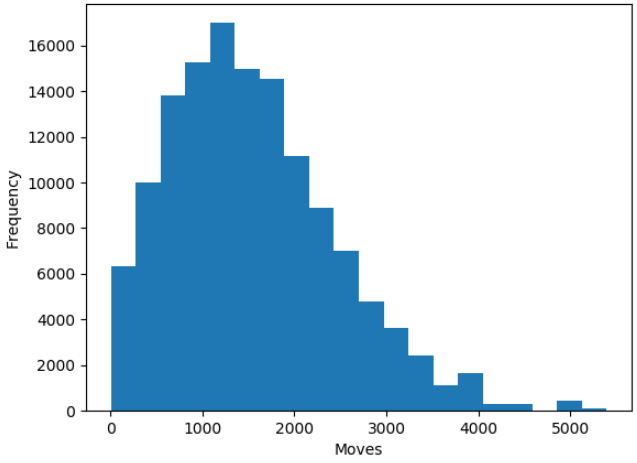

Figure 13. **EGS Agent Moves Searched.** *Distribution of moves searched per decision by the EGS agent with a search depth of 3 moves over 500 games.*

represents a different parameter combination. For RGS, 1, 2, and 3 GPM are plotted. For EGS, search depths of 1, 2, and 3 are plotted. For DLRGS, search depths 1, 2, 3, and 4 are each plotted separately with varying GPM. While RGS and EGS plateau, DLRGS with a search depth of 3 has the most positive slope on the graph. This supports the conclusion that a search depth of 3 with multiple games tested per move is the optimal graph search approach for Square Stacker. Higher GPM testing for higher search depths is needed to determine if the trend continues forward.

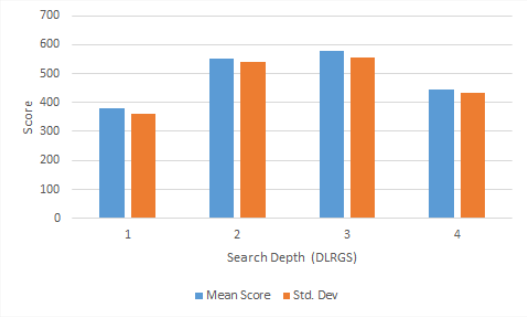

Figure 14. **DLRGS Agent Scores by Search Depth.** *The mean and standard deviation of scores over 3000 games for search depth values of 1, 2, and 3 with varying games per moves using the depth-limited random graph search agent.*

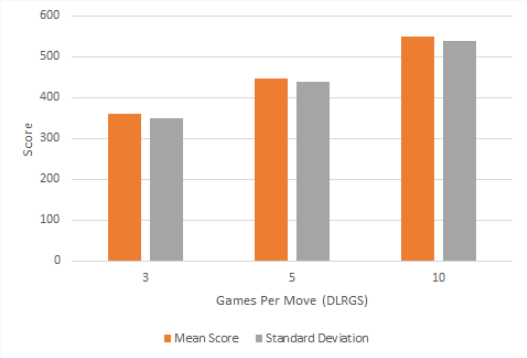

Figure 15. **DLRGS Agent Scores by Games per Move.** *The mean and standard deviation of scores over 3000 games for games per move values of 3, 5, and 10 with varying search depths using the depth-limited random graph search agent.*

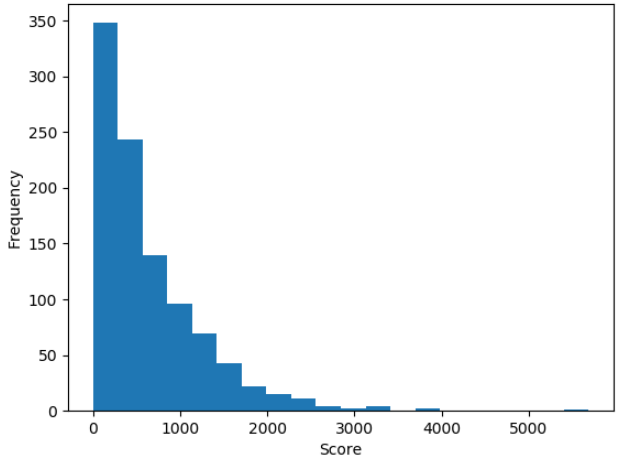

Figure 16. **DLRGS Agent Scores.** *Distribution of scores of the DLRGS agent with a search depth of 3 moves and simulating 10 games per move over 1000 games.*

4.5. Monte Carlo Tree Search

The MCTS agent beat the random agent, but did not have a higher average score than the human agent. However,

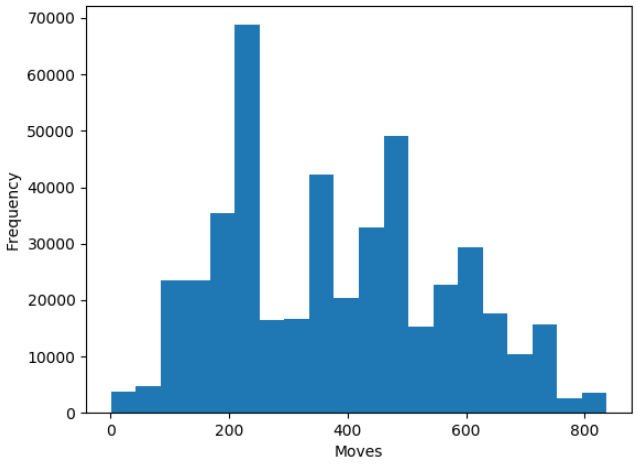

Figure 17. **DLRGS Agent Moves Searched.** *Distribution of moves searched per decision by the DLRGS agent with a search depth of 3 moves and simulating 10 games per move over 1000 games.*

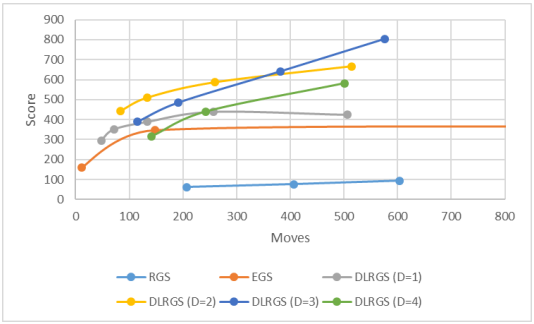

Figure 18. **Graph Search Agent Efficiencies.** *Efficiency scores of each version of the graph search agent. DLRGS is shown to have the highest score and best trend line. DLRGS with search depth D=1, D=2, D=3, and D=4 are each plotted separately with varying GPM.*

its max score was higher than any human score. The only parameter that was varied was the amount of simulations per move (SPM). In general, each move took about a minute, and each game lasted for a couple hundred moves, which means that games took a large amount of time to simulate. Testing 150 SPM with 100 games took about 25 hours in total.

All of the averages were fairly close together, indicating that a higher amount of simulations did not result in significantly better moves. Despite scoring significantly higher than the random agent, there were still games that resulted in very low scores.

4.6. Comparison of Methods

The best results collected from each agent are shown in Table 3, ranked by mean score.

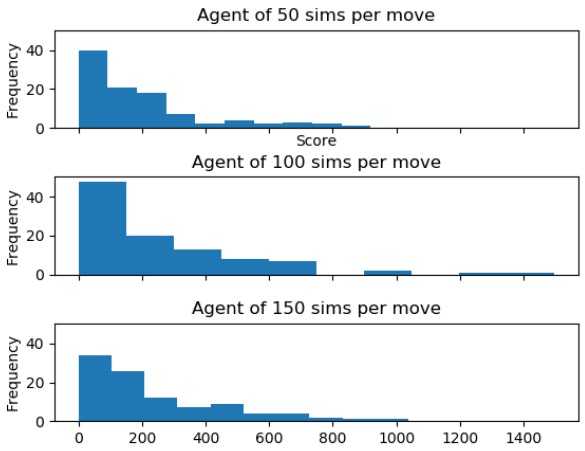

Figure 19. **Frequency of Scores of MCTS Agent.** *Results of scores of different amounts of simulations per move with 100 games each.*

TABLE 3: **All Agent Score Statistics.** *Best results of each agent ranked by mean score.*

Agent	Game Count	Score Statistics			
		Min	Max	Mean	Stdev
Random	10000	0	188	19	20
STM	1000	0	171	20	22
DQN	1000	0	223	21	21
RGS	1000	0	709	93	95
MCTS	300	0	1498	233	238
EGS	500	3	2327	362	350
Human	20	86	1694	454	428
DLRGS	1000	6	3678	804	750

5. Conclusion

Overall, the search-via-simulation agents (RGS, EGS, DLRGS, MCTS) far outperformed the rule-based agents (STM, DQN) in Square Stacker. It seems that the practice of simulating short-term outcomes multiple times (as in DLRGS) is the most efficient and effective method for this task. DLRGS was the only agent to outperform the average human player with the parameter combinations tested.

One of the reasons that this game is difficult to play is due to high amounts of randomness. If the game state is represented as 4 columns of 3 rows each (3 columns for the board itself and 1 where the tiles are generated) then every 3 turns, 25% of the game state is randomly generated. This random generation is something that is extremely hard to predict as it causes small mistakes early in the game to be disastrous later. It is due to this randomness and the large amount of possible game states that rule-based agents seemed to fail. The agent simply cannot abstract rules from playing the game that are comprehensive enough to apply to other situations. This is a difficult task for human players as well. However, with enough practice, some intuition was developed by the human players which allowed for higher scores. However, most of the algorithms tested still focused on maximizing short-term scoring. As shown in the human agent results, the randomness still affected human agents a lot, as there were often games with very low scores.

It is hypothesized that the DQN failed to learn due to the large size and stochastic nature of the state space, but it is possible that a DQN for this game could be trained with a large enough number of experiences, or perhaps in a hybrid model learning from one of the search agents explored in this paper, as done in [9]. Additionally, more or wider hidden layers in the deep neural network may be needed to emulate the complex logic used in estimating the rewards. Alternatively, the DQN concept may be entirely redundant, as the expected reward of game moves can be estimated fairly well via the random simulation used by the graph search agents. Reinforcement learning in highly-stochastic environments is still a wide-open field in need of additional research.

DLRGS was the most successful agent, because out of the agents that were tried, it is the best suited to deal with random future states. The best strategy for the game is to make moves that clear rows or set up the board to clear rows later on, hopefully scoring combos along the way. Therefore, the ideal agent would need to see possible futures and select moves that set up the board for high-scoring row clears. DLRGS does this best because it tries each move multiple times to see different opportunities it could be setting up, and also goes a few moves deep to see if each move will be good for setting up a row clear in the future. The best depth to search ahead seems to be three as the future state of the game gets too randomized past three moves. On the other hand, the positive trend between mean score and GPM continues through 10 GPM, which does not show any sign of plateauing. Due to time constraints and the length of time each test takes past games 10 GPM, higher GPMs were not tested but will likely lead to higher scores. Further testing could verify higher GPMs help the agent see more possible futures and therefore lead to higher scores.

References

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without Human Knowledge," *Deepmind*, 18-Oct-2017. [Online]. Available: https://deepmind.com/research/publications/mastering-game-go-without-human-knowledge. [Accessed: 03-Dec-2019].

[2] B. Baker, "Emergent Tool Use from Multi-Agent Interaction," *OpenAI*, 29-Oct-2019. [Online]. Available: https://openai.com/blog/emergent-tool-use/. [Accessed: 04-Dec-2019].

[3] F.-hsiung Hsu, *Behind deep blue: building the computer that defeated the world chess champion*. Princeton, NJ: Princeton University Press, 2004.

[4] Y. Naddaf, "Game-independent AI agents for playing Atari 2600 console games," University of Alberta, 2010.

[5] I. Szita and A. Lörincz, "Learning Tetris Using the Noisy Cross-Entropy Method," *Neural Computation*, vol. 18, no. 12, pp. 2936–2941, Dec. 2006.

[6] C. J. C. H. Watkins and P. Dayan, "Q-learning," Machine Learning, vol. 8, no. 3-4, pp. 279–292, 1992.

[7] M. Liu, "General Game-Playing With Monte Carlo Tree Search," Medium, 13-Jul-2018. [Online]. Available: https://medium.com/@quasimik/monte-carlo-tree-search-applied-to-letterpress-34f41c86e238. [Accessed: 07-Dec-2019].

[8] S. J. Russell and P. J. Norvig, *Artificial intelligence: a modern approach*, 3rd ed. Upper Saddle River, New Jersey: Pearson Education, Inc., 2010.

[9] McAleer, S., Agostinelli, F., Shmakov, A. and Baldi, P. (2019). Solving the Rubik's Cube Without Human Knowledge. [online] arXiv.org. Available at: https://arxiv.org/abs/1805.07470 [Accessed 1 Dec. 2019].