CS4180 - Project Proposal

Tak Boardgame Environment & AI Player Algorithm

Pablo Kvitca - Fall 2021 - December 14, 2021

Introduction

The game *Tak* was created for the fictional world of *The Kingkiller Chronicles* (by Patrick Rothfuss in collaboration with James Ernest). It is a "chess-like" game from a fictional world, with 2 adversarial players taking turns to play black/white pieces. Unlike Chess, Go, and other 2 player board games, *Tak* has a 3-dimensional "stacking" move that makes it stand out. At the sametime it's more complex, both to human players and to implement the rules for players' actions in a computer program.

My project has two parts: 1) implementing the environment and rules of the *Tak* board game; 2) implementing and testing learning/planning algorithms on it. The code for the environment, algorithms, experiments, and analysis is available on github.

NOTE: all the figures are repeated in a larger format in the appendix

Problem Description

The *Tak* board game is a **two-player**, **zero-sum adversarial**, **full-knowledge** board game. The game being full-knowledge means the rules and transitions from one state to another are known and do not vary randomly. The zero-sum adversarial part means one player wins and gets the maximum reward when the other player loses and gets no reward.

Rules of the Tak Board Game

The full rules of the board game can be found online at ustak.org. This is a short description of the rules, but I will focus this report on the learning algorithm and the results on the game, rather than this specific environment.

- **Board**: the game can be played with square boards of sizes of 3, 4, 5, 6, 7, or 8
- **Game Pieces**: each player has a number of *stones* and (possibly) one *capstone.* The *stones* can be placed on the board as *flatstones* or *standing*
- **Goal**: create a ***path*** from one side of the board to the opposite side (bottom-top or left-right). A path is a line of *flatstone* or *capstone* pieces (but not *standing*) connected in orthogonal directions (diagonals do not count).
- **The first turn**: the players must place a *flatstone* of the *opponents* color in any empty space
- Each following **turn**, a player may:
    - Place a new *flatstone*, *standing stone*, or *capstone* (if available) on an empty space
    - Move a ***stack*** they control in a single direction
- A stack is controlled if the piece on top belongs to the player
- A move action consists of:
    - Pick up a stack (at most *B* pieces, where *B* is the size of the board)
    - Move the stack in one direction (up, right, down, left)
    - Drop at least one piece from the bottom of the stack in the positions in the direction moved
- When moving pieces:
    - Any piece might be placed on top of a *flatstone*
    - A *flatstone* cannot be placed on a *standing stone* or a *capstone*
    - A *capstone* might be placed on a *standing stone*, by "flattening" that stone
    - A *capstone* cannot be flattened.
- The game ends if: a path is formed, a player runs out of *stones*, there are no more empty positions on the board (even if move actions are possible).
    - In case the game ends without a path, the winning player is determined by the number of positions controlled by each player, which may result in a **tie**

Scoring the Game

There are several methods for scoring the results of the game. The environment I implemented has functionality for all of these. However, they don't change who wins the game, just how they are scored. The scoring rules vary in minimum and maximum points the winning player can get, so games scored with one method cannot be compared to the other. While the end conditions for the game don't vary, the different scoring methods would vary the optimal game for getting the maximum scores. In my experiments, I strictly only used the "default scoring", where the winning player gets one point per position on the board plus one point per each unplayed stone. For example: if the black player wins with *12* stones left on a *5x5* board, their score would be *5*5 + 12 = 27*. The other scoring

methods can be found on the full ruleset. Additionally, due to my implementation, if the game ends in a *tie* both players get *0* points.

Background

There is some previous work on implementing this game online and as an environment on OpenAI Gym, *DeepRL-TakEnv*. However, the online version is hard to interact with, especially due to the 3D component of the game. Additionally, the existing environment for OpenAI Gym is not being maintained and I am unsure of its correctness. I decided it would be easier to write and test the algorithms if I implement the environment from scratch

Game Environment

My implementation of the environment is done fully in Python and it extends the OpenAI Gym's environment interface. I used this interface to have a clear standardized way of interacting with the environment. I have not used any of OpenAI Gym's algorithms, but they should be applicable without too much work. To spend more time exploring the learning algorithms and experiment results, I am only giving a short description of the implementation here. The diagram below shows a high-level class diagram for the implementation with the key components used by the environment (*TakEnvironment* class). The environment has some initial settings and a current *state*, which is an instance of *TakState*. The state is defined by: the *current_player* (*TakPlayer* enum), the number of *stone* pieces available to each player (integers), whether each player has a *capstone* available (booleans), and the *board* data.

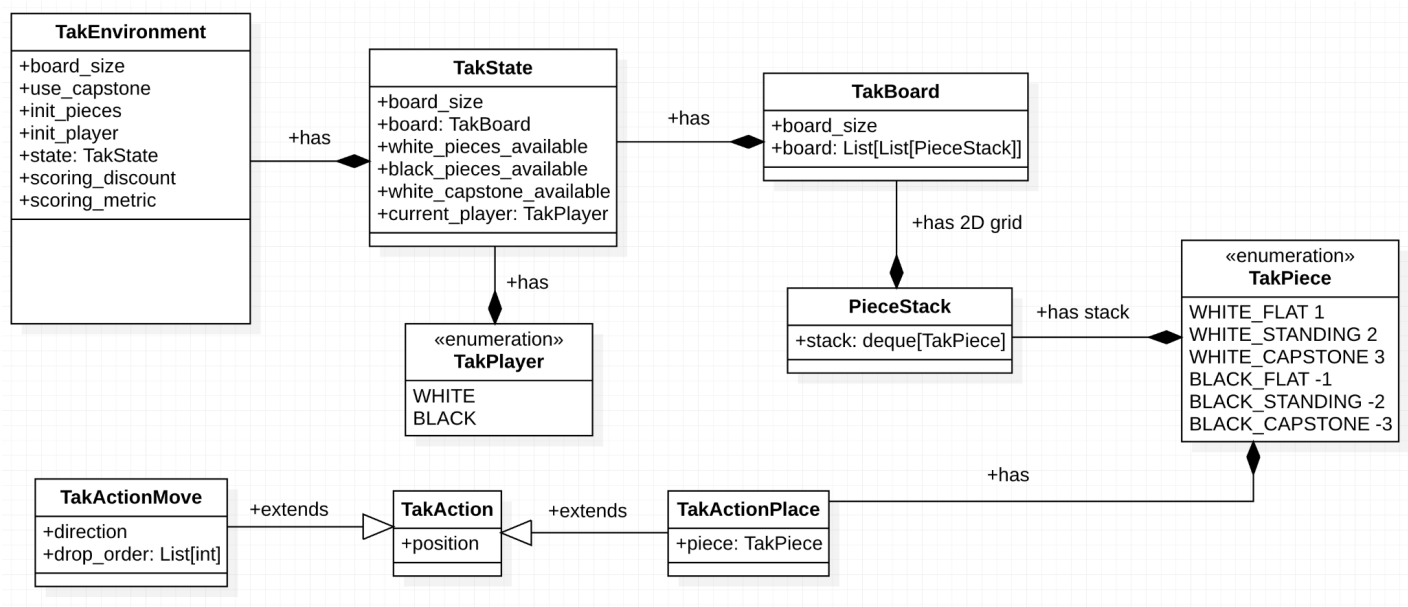


Figure 1: Class Diagram for Tak Environment

The board is an instance of the *TakBoard* class, which has a 2D list of *PieceStacks*. This is a wrapper class with various convenience methods for querying the state and transitioning from one board to another (given an action). The stacks contain a *deque* of *TakPieces*, which is an enumeration where the negative values represent black pieces and the positive values represent white pieces.

The actions are represented in two types, *TakActionMove* and *TakActionPlace*, which both extend the abstract class *TakAction*. The move actions are defined as the *from_position*, their *direction*, and a *drop_order*, while the place actions are defined by their *position* and the piece type being placed. Note that the move actions don't describe the pieces being moved, this is because the action is applied (if valid) to a given state.

Additionally, most of the behavior and methods for the environment, the state, and the actions are UNIT tested. This is done to validate the implementation and that the rules of the environment match the game rules. UNIT tests are only done for the game environment itself, the learning algorithms are tested manually.

Complexity of the Game

Since the game is zero-sum and full-knowledge, a basic Min-Max algorithm could, in theory, always optimally solve the game. However, this would require exploring a huge tree of possible states for the game. The total number of states depends on the size of the board. The size of the board determines the number of stones available to each player. Then each position can have a stack of unlimited height (up to all the stones in the game). To give an idea of the complexity of the game, the formula below gives the total number of possible *actions* at any state. Note that depending on the state most of these actions would not be valid.

$$actions = 3b^2 + 4b^2 \sum_{i=1}^{b} count\big(ordpartitions\ (i)\big) = 3b^2 + 4b^2 \sum_{i=1}^{b} 2^{i-1}$$

Figure 2: Formula the for the total number of actions

This formula is given by the board size (*b*). For each position on the board (b^2) there are 3 possible place actions (*flatstone, standing*, and *capstone*). Plus, for each position on the board (b^2) there are four possible directions to move on (*up*, *right*, *down*, *left*). Then for each move's origin and direction, there are a number of possible "*pickup counts*", from 1 to the size of the board (*b*). For each of those, there are a number of possible "*drop orders*". The *drop orders* can be calculated by counting the number of "ordered partitions" of the *pickup count*.

The *partitions* of a number *n* are the different ways integers can sum up to that number *n*. The *ordered partitions* count each different "sorting" of integers. Example: *partitions(3)={(3), (2+1), (1+1+1)}* and *ordered_partitions(3)={(3),(2+1),(1+2),(1+1+1)}*. I found that the number of ordered partitions follows the formula 2^{n-1}.

Rendering the Game

The game has a 3-dimensional component, this makes displaying the state of the board on the screen complicated. First, a 2D text grid can only show one "layer" of the board at the time, which is hard to read for a human. I decided to use the PyPlot library to render a basic, rendering of the game. The advantage of using this library is that the 3D plot is displayed on a web browser window and can

be rotated to different perspectives. This is not a “playable” rendering of the game, just a static state of the game, mostly used for debugging the environment and the player algorithms.

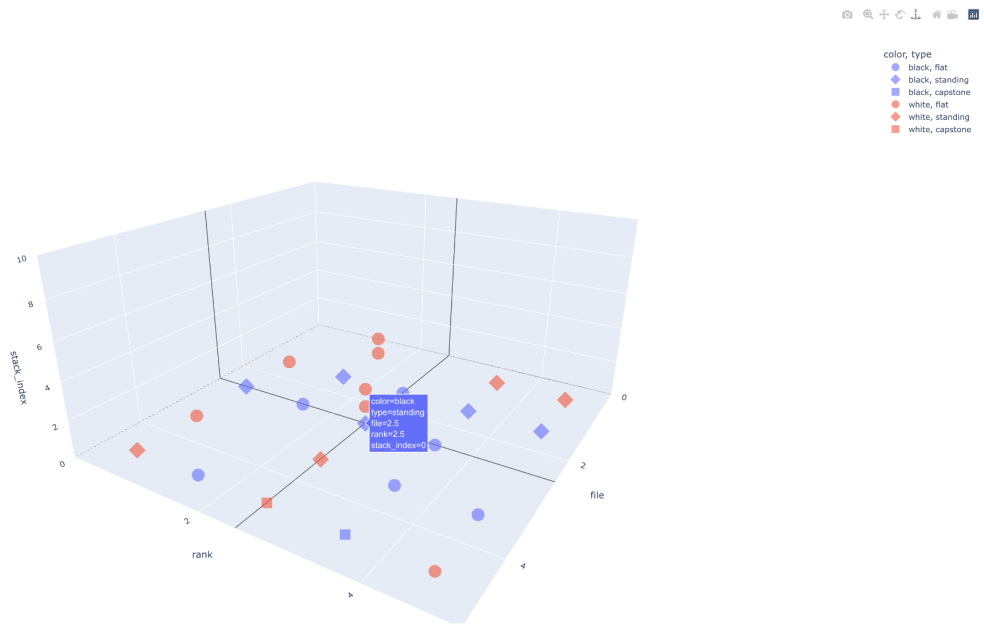


Figure 3: Example game state 3D rendering

AI Player for Tak

The optimal move can be found by expanding the state tree through all actions until reaching terminal states. Then choosing the action that maximizes the reward, taking into account the goal of the opponent, using a Min-Max algorithm. However, as the size of the state tree grows, it becomes more expensive. My approach uses Monte-Carlo Tree Search (MCTS) to intelligently select actions without expanding the whole tree. The method stores information from all past games and its estimates should be better over time. Due to time constraints for the project, I mainly focused on the MCTS method with three different rollout policies. I also tested each of the rollout policies on their own over a number of games.

Implementing “PlayerAgents”

Each method for playing the game is implemented as a “*PlayerAgent*”, this is simply a Python class that the methods implement so there is a single interface for using the different methods. The player agents have an **initialization** constructor and a **select action** method. This method is given a state and uses the agent’s policy to select the next action for it.

Implementation Challenges

- Initially, I implemented the methods to generate a list of all ***valid*** actions. This resulted in being slow and continuously generating the same actions many times per game. In the most complex methods, the same actions were generated many times in a row for a single step of the game.
- The first try at optimizing this was to use *memoization* to store the previous compute results and reuse them. This worked initially, but as the number of memorized values grows with the visited states of the game, the memory required to store them grows exponentially. After visiting many states, through a few games, the required memory passed the available real memory in my system. In one test, after running for about 2 hours the space used by the *python* processed over 90Gb. While the theoretical time complexity of this was lower, the real-time needed to retrieve the values from compressed memory was too slow. This made the overall process slower.
- The solution I came up with was to pre-generate all possible actions, and test whether they are valid at each point in the game given the current state. This is slightly less efficient, but the total actions can be stored in memory, and validating them with a given state is a faster operation. Overall, this resulted in usable times for running the experiments I wanted a few times.

Using the environment with the PlayerAgents

Since this is a two-player game, using the environment has a slight modification to the normal "*step while not done*" process. The first part is to initialize both agents, for the white and black players. Then the first player (usually white) selects their action and the environment steps with it. If the resulting state is not terminal, the second player selects their action, and the environment steps with it. If that results in a terminal state the loop ends, if not the code loops back.

Algorithms

I implemented two types of players: players using a **normal policy** and players using **planning**. The former refers to players that use the random policy or an e-greedy policy based on some Q-values for state/action pairs. The latter refers to players that explore different actions and their outcomes in some way. The planning methods may use other policies or state evaluation methods internally.

Policies

- **Random Policy**: simply selects one of the *valid* actions purely at random.
- **Weighted Random Policy**: first selects whether to choose a *place* or *move* action, with a given probability *p* (and *1-p*), then selects one of the *valid* actions of the chosen type.
    - I added this method after some initial testing with the random policy. At the start of the game, the number of valid *place* actions is a lot more than the valid *move* actions. So the random policy tends to do a lot more *place* actions. This does not perform great and most of the games end with no paths at all and result in ties.

- **E-Greedy Policy**: selects a random action with probability *e*, and best action with probability *1-e*. The best action is computed by the Q-values of the state/action pairs for the current state.
    - The *e-greedy policy* I implemented uses Q-values that initialize at 0 for all pairs.
    - The policy has an extra *update* method that updates the Q-values with respect to the rewards taking the selected action. This update is done in the method explained in the SARSA Policy iteration below.

Learning & Planning Algorithms

I implemented two learning algorithms that learn through the results of many game runs. The first is based on Q-Values and policy iteration using the SARSA method. The second is based on Monte-Carlo Tree Search, inspired by the work in AlphaGo Zero [1]. These are implemented with some variations specific to the TakBoard game.

SARSA Policy Iteration

I implemented SARSA Policy Iteration as a learning method. One key distinction between the algorithm studied in class and the book is how the update is learned. SARSA chooses the next action and uses it as part of the update. However, since this is a multi-agent environment, the next action for one agent cannot be selected until the other agent takes their turn. I implemented the algorithm so that the update to the Q-values would happen after selecting the action in the following turn, but before taking that action. Following is some high-level pseudocode on how this works:

```
1. Initialize the e-policy and agents for each player (white and black)
2. Initialize a prev_step tuple for each player. The tuple has three positions: the
   previous state, the action taken on that state, the resulting reward from that
   action (which is 0 unless the action resulted in a terminal state).
3. Initialize the state # (the start of the game)
4. While not done:
      a. Select action for the first player
      b. If the prev_step for the first player has data:
            Update the Q-values using SARSA update:
Q[prev_state,prev_action] += alpha*(reward + gamma*q[state, action] - q[prev_state,
prev_action])
      b. Take selected action, and observe reward and next state
      c. Store the state and action in the prev_step for the player
      d. If the new state is terminal, break the loop
      e. If not, update the current state to the new state
      f. Select action for the second player
      g. If the prev_step for the second player has data:
            Update the Q-Value using SARSA update
      h. Take selected action, and observe reward and next state
      i. Store the state and action in the prev_step for the player
      j. If the new state is terminal, break the loop
      k. If not, update the current state to the new state
5. Update the Q-values using SARSA update for the last player to take a turn
```

The last step (5) in the algorithm above is necessary since it is the only time a non-zero reward is actually observed. This algorithm is implemented in the file *experiments/sarsa.py*. The hyperparameters for this algorithm are *epsilon* for the e-greedy policy, the *alpha* learning rate, and the *gamma* discount rate.

Monte-Carlo Tree Search Learning (MCTS)

This method does both a kind of planning for the current game and learning from the previous game, as well as from the *rollout runs* during its execution. This method was suggested by the teaching staff when brainstorming the project. I implemented this from scratch based on the lectures and some preliminary research. This is one of the methods used by DeepMind's AlphaGo Zero [1]. I combined this with some domain knowledge about the *Tak* board game.

This method uses a tree (or graph) of information about the game. This could start off as empty or with information from previously played games. As the agent plays more games, the information on the *knowledge graph* grows and gets "better".

The action selection using MCTS has five parts: 1) **exploring** the knowledge tree by selecting and taking greedy actions, starting at a node that represents the current state of the game; 2) Once a leaf state is reached, or a given *max exploration depth* is reached, **expand** leaf by taking an action; 3) Then, **simulate** various games using a fast *rollout policy*; 4) Use the results of the simulated to **backpropagate** of states that lead to this leaf, up to the starting node; 5) Finally, **select** the action with the highest number of wins.

I applied some variations to the MCTS algorithm explained above. These are based on some intuitions about how the board game works and some interesting research on the topic:

1. Changed the action selection in the exploring stage (1) to use e-greedy selection: with some probability *e* it selects a random action (for which we might have no information), potentially growing the graph; With some probability *1-e* it chooses a greedy action, based on the wins/losses information for each known action (each edge from the current node).
2. The order of the actions and states does not matter for choosing an optimal action in this board game: "the best action is determined only by the current state". This means the information about wins/losses from the simulation might be "reused" on different "paths" of the game. This, combined with [2] and [3], inspired me to change the *knowledge tree* for MCTS to a *knowledge graph*. Here, the nodes in the graph represent unique states, which can potentially be reached from various previous states, by taking different actions. Additionally, this is a *directed* graph, where the edges go from previous states to next states. This has no effect on the exploring, expansion, simulation, and select stages (1, 2, 3, and 5). However, the backpropagation (4) is modified so that, Instead of propagating the results only to the *parent* of a node, it is propagated to every node that can be reached by following the directed edges in "reverse".

The implementation for the knowledge graph is a class in the *agents/TakMCTSPlayerAgent.py* file. This class has various convenience methods and stores the graph information using a *Graph* data

structure from the *python-igraph* package [4]. The execution of the rollouts in the simulation stage (3) can be done in parallel and is implemented with a flag to enable/disable multithreading.

Additionally, the *TakMCTSPlayerAgent* supports any policy to be used as a *rollout policy*, as long as it follows the expected interface. This can be given during the initialization of the agent. The experiments use this method with three different policies: the *random policy*, the *weighted random policy*, and the *e-greedy policy with SARSA policy iteration.* In the case of the *SARSA policy iteration*, the simulation stage (3), is modified to call the policy's update method during execution. The policy iteration is not necessary to use the *e-greedy policy* and could be replaced with a policy with fixed, predetermined Q-values.

The hyperparameters for this algorithm are the *mcts_depth* (max depth for the exploring stage, 1), *mcts_epsilon* (*e* value for the exploring stage, 1), the *mcts_iterations* (the number of times to repeat stage 1 through stage, 4 before the final selection), the *rollout_runs* (number of rollouts for the simulation stage, 3), and the *rollout_policy* to use. Plus any hyperparameters for the rollout policy itself.

Experiments & Results

I tested the two random policies, SARSA Policy Iteration with an *e*-greedy policy, and MCTS (with the random and *e*-greedy methods as rollout policies) in various experiments. I ran the experiments using the code in the *experiments* folder and stored the results in *.csv* files. Then, I analyzed the results and generated plots using a Jupyter Notebook. I ran various experiments throughout the implementation and debugging of my algorithms, which I explain and analyze below. Most of the experiments were done with a board of size *3*, as the methods slow down significantly with larger boards.

Random Policy & Weighted Random Policy Players

The first two experiments consist of running the game many times using the *random policy* and the *weighted random policy* for both players. The game was run *300* times per configuration, where each configuration is a combination of game boards with sizes 3, 4, or 5, and the starting player is the white or black player. Taking into account which player starts the game is important since there is a slight advantage to the first player (though due to time constraints not all experiments consider this variation). The two plots in *Figure 4* show the results of these games. Using the purely *random policy*, most of the games result in ties, and about the same number of games are won by the white and black players, regardless of which one started. However, when applying the *weighted* variation, we get some interesting results.

First, note that in some cases the number of ties is increased. I believe this is due to the *weighted* version having a higher chance to choose "place" actions in advanced states of the game, which would lead to termination of the game by the secondary condition (refer to rules above), leading to a tie. The same thing that gives the *weighted random policy* at the beginning of the game may lead to more ties. A possible solution would be to vary the weight used throughout the progress of the game.

Second, these results (on the right of Figure 4), show that the player that started the game wins more often, which is expected due to the advantage mentioned before.

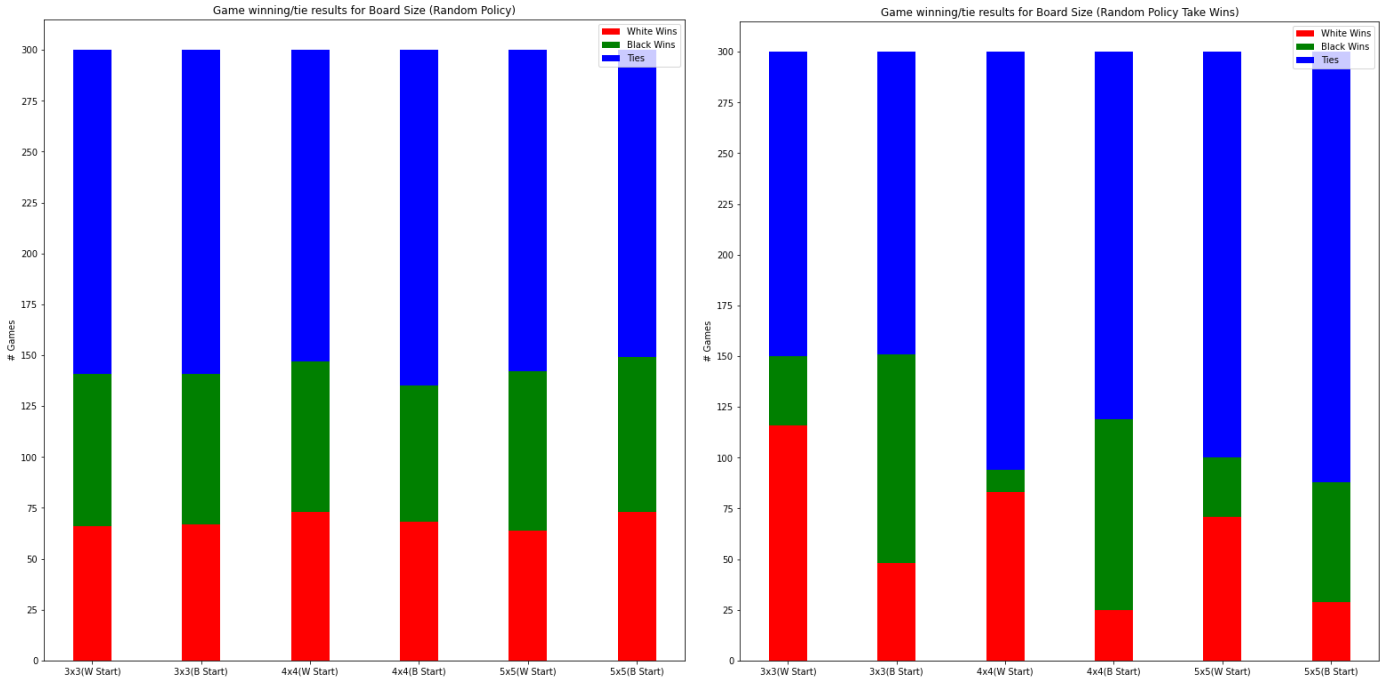


Figure 4: Comparison of the results for the *random policy* and the *weighted random policy*. The plot on the left shows the results of *300* games played with both players using the *random policy*. The plot on the right shows the results of *300* games played with both players using the *weighted random policy*. The blue sections represent *ties*, the green sections games *won by the black player*, and the red sections games *won by the white player*.

SARSA Policy Iteration

My next experiment was testing the SARSA Policy Iteration for learning Q-Values with an *e*-greedy policy. Due to time constraints and unforeseeable circumstances, I was only able to run these with a single value for the *epsilon* (0.1), *alpha* (0.99), and *gamma* (0.99) hyperparameters. The plot in Figure 5 below shows the results of using policy iteration for 1000 episodes. The test was run 10 times with the same parameters, given the shown confidence bands (at 95%). This method does not seem to learn enough through the tests 1000 episodes, the average reward resulting from each episode varies between *0* and *5*.

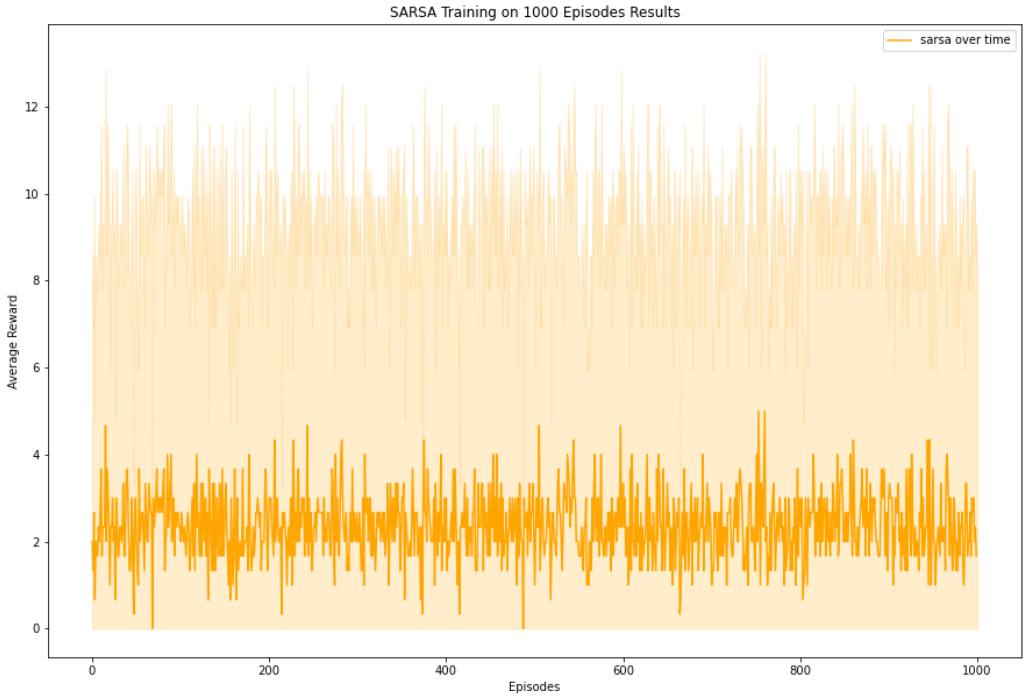


Figure 5: shows the average reward obtained on the *n*th episode, for SARSA Policy Iteration. As the episodes progress the method should improve. Shows mean reward and a confidence band on the positive side (negative reward is not possible).

MCTS Parameters Playing vs Random Policy Player

The main method of interest was MCTS. The first experiment using this method was comparing one player with the MCTS versus a player using the Random Policy. The plot below in Figure 6 shows the cumulative reward for each player through the progress of 100 games, through which the MCTS method learns. The amount of learning can be seen on the graph by looking at the "inclination" of the lines, unfortunately, I did not have time to plot the derivatives. The different colors in the plot show variations on the *mcts_epsilon* and number *rollout_runs* hyperparameters. The first three (red, green, blue) show the purely-greedy method. The best performing parameters (purple) were *mcts_epsilon = 0.9* and *rollout_runs = 32*. A first observation is that adding the *e*-greedy selection can be effective at making the MCTS selection perform better. Second, as expected, the higher *rollout_runs* perform better. Note that the amount of rewards for the last 20 games adds up to over 150, while the previous 80 runs add up to about 450. This shows that the MCTS method learns over time.

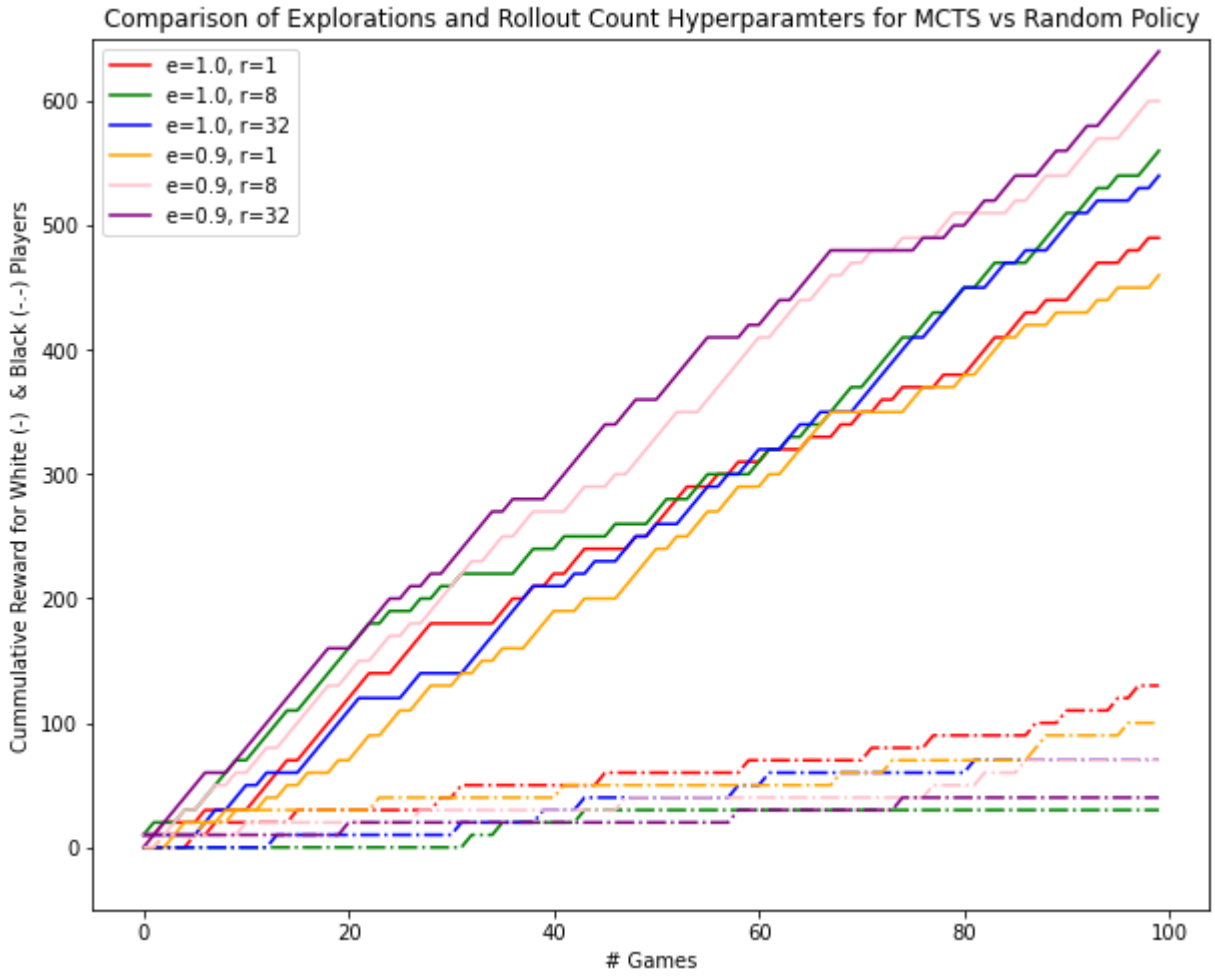


Figure 6: shows the cumulative reward for an MCTS player, shown with the full lines, against a Random Policy player, shown with the dashed lines. Each color represents different runs of this same trial using different values for the *mcts_epsilon* and *rollout_runs* hyperparameters for MCTS.

Comparing MCTS Rollout Policies

Finally, I wanted to compare the performance of MCTS playing against another MCTS player, while also varying the number of *rollout_runs (*between *16* and *32*) and, most importantly, the *rollout_policy*. I tested three versions of rollout policies, the fully *random policy*, the *weighted random policy*, and the *e-greedy policy with SARSA Policy Iteration*. Unfortunately, some time constraints limited how many runs of the methods I could do. First, I only run each variation in one trial shown on the results. Ideally, this would have been many trials (*~50*) plotted as the mean with confidence bands. The second limitation was that the number of games (episodes) was limited to only *10*. I would have preferred to test these methods through at least *100* episodes.

The results are shown in Figure 7 below. The plot on the left shows the cumulative reward results through the *10* games for each variation. Since these are for a single run, I can't draw any conclusions. However, the models with more *rollout_runs* performed better in these specific runs. Specifically, the model using the *e-greedy policy with sarsa policy iteration* as the rollout had the highest returns. However, as shown on the right side of the plot, this advantage came with the runtimes to

select the next action. The two methods using *SARSA* had the highest times, which is expected due to the complexity of the rollout policy. Similarly, the runs using a higher number of rollout runs also take longer than the others.

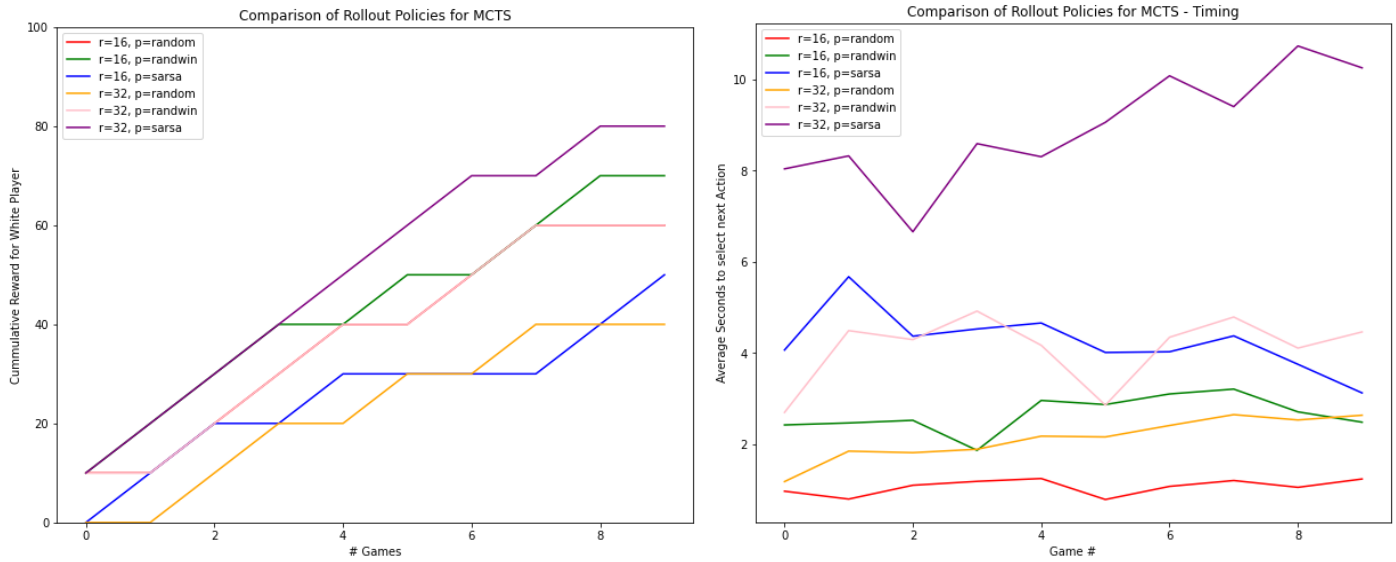


Figure 7: The left plot shows the returns throughout *10* episodes for a single run of a MCTS agent playing versus another MCTS agent. The different colours show returns for varying hyperparameters used. The right plot shows the average time to select an action at each episode.

Discussion

Considering the methods I implemented and tested for the player agents on the *Tak* board game and the analysis above, I believe this project is a solid first step towards using learning methods in this environment. The 3D component of the board game adds an interesting dimension to the normal two-player board game environments.

Key Takeaways

- The MCTS method seems to perform well on the *Tak* board game.
- Using better, more complex, policies in the simulation stage of MCTS leads to better learning results.
- Using a higher number of rollout runs per simulation leads to better estimates for the actions.
- Using e-greedy exploration and action selection in MCTS can lead to some improvement.
- The E-Greedy Policy by Q-Values through SARSA Policy Iteration does not learn good q-values. fast, though better hyperparameter tuning and longer training might improve this result.

Limitations & Advice

- Some of the experiments were run with very limited trials and episodes. This was due to personal unforeseeable circumstances, unrelated to the project, on the last day of the project. I would advise future students to focus on getting all results as soon as possible

- The performance of the Environment in Python might not be optimal, it might be interesting to explore implementations that run natively on the computer or that can take advantage of GPUs.
- I could not explore various hypermeter values for the SARSA Policy Iteration method.

Future work

- Explore the current experiments through more trials and more episodes, validating the observed behaviors.
- Explore hyperparameter tuning for SARSA Policy Iteration.
- Explore hyperparameter tuning for MCTS methods.
- Implement a *dynamic* version of the weighted random policy, where the probability of choice *place* and *move* actions varies with respect to the number of pieces on the board or the number of controlled spaces on the board.
- Explore using MCTS with pre-trained Q-values (possibly through policy iteration) for the e-greedy policy.
- Implement SARSA with Q-values for aggregated states/action pairs, rather than state/action pairs. Specifically, by only using the "top" piece in each stack, not the full 3-dimensional board.
- Explore more advanced rollout policies, including methods based on DQN and state aggregation.

Conclusion

The MCTS method implemented shows a lot of promise, and further work on it would be a good research direction for building good learning player agents for the *Tak* board game, and possibly other two-player zero-sum games with full knowledge.

References

1. Silver, D., Schrittwieser, J., Simonyan, K. *et al.* Mastering the game of Go without human knowledge. *Nature* 550, 354–359 (2017). https://doi.org/10.1038/nature24270
2. Czech, Johannes, Patrick Korus, and Kristian Kersting. "Improving AlphaZero Using Monte-Carlo Graph Search." Proceedings of the International Conference on Automated Planning and Scheduling. Vol. 31. 2021. https://ojs.aaai.org/index.php/ICAPS/article/view/15952
3. Leurent, E. & Maillard, O.. (2020). Monte-Carlo Graph Search: the Value of Merging Similar States. *Proceedings of The 12th Asian Conference on Machine Learning*, in *Proceedings of Machine Learning Research* 129:577-592. https://proceedings.mlr.press/v129/leurent20a.html
4. *Get started with igraph in python*. igraph. (n.d.). Retrieved December 15, 2021, from https://igraph.org/python/#docs

Appendix - Larger Plots

Figure 4

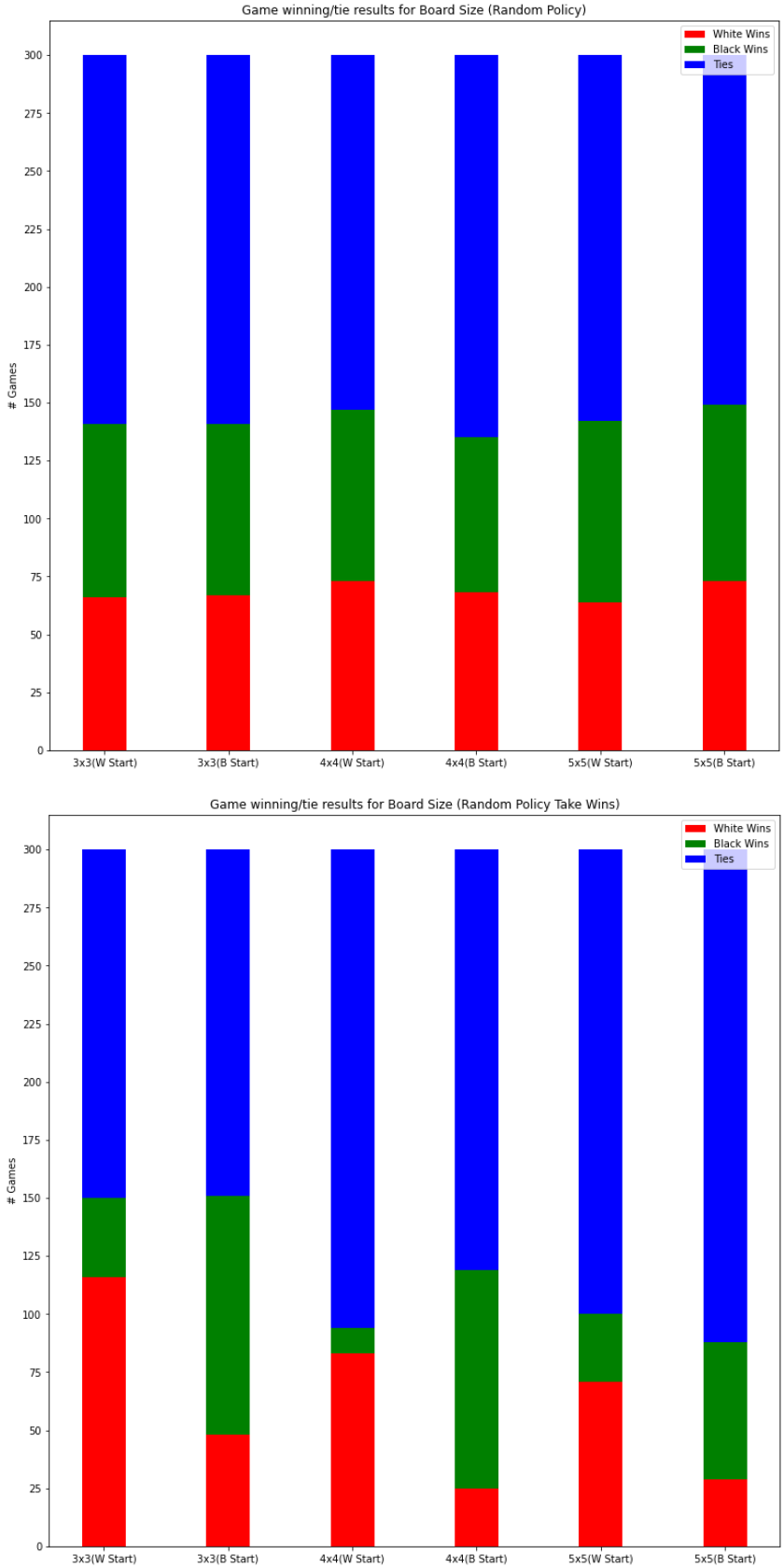


Figure 5


SARSA Training on 1000 Episodes Results
sarsa over time
Average Reward
Episodes
0
2
4
6
8
10
12
0
200
400
600
800
1000

Figure 6

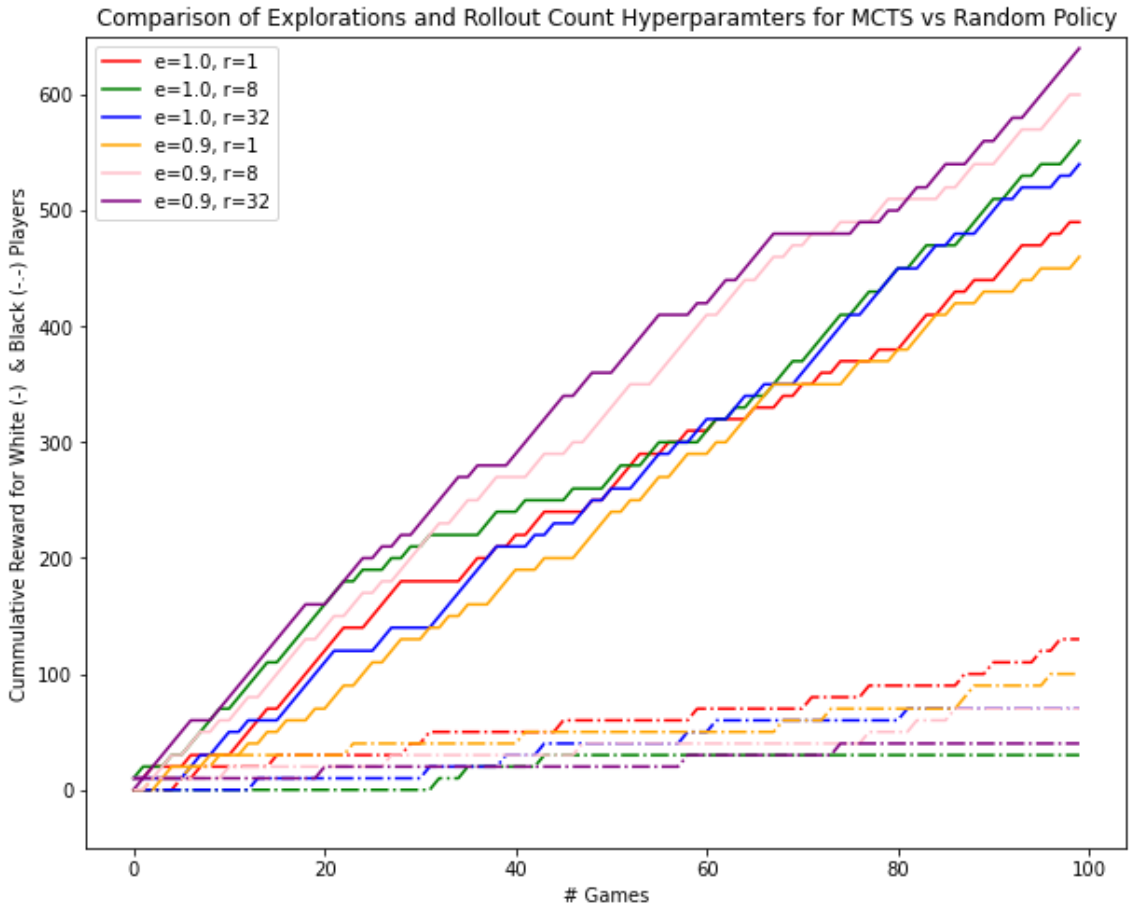
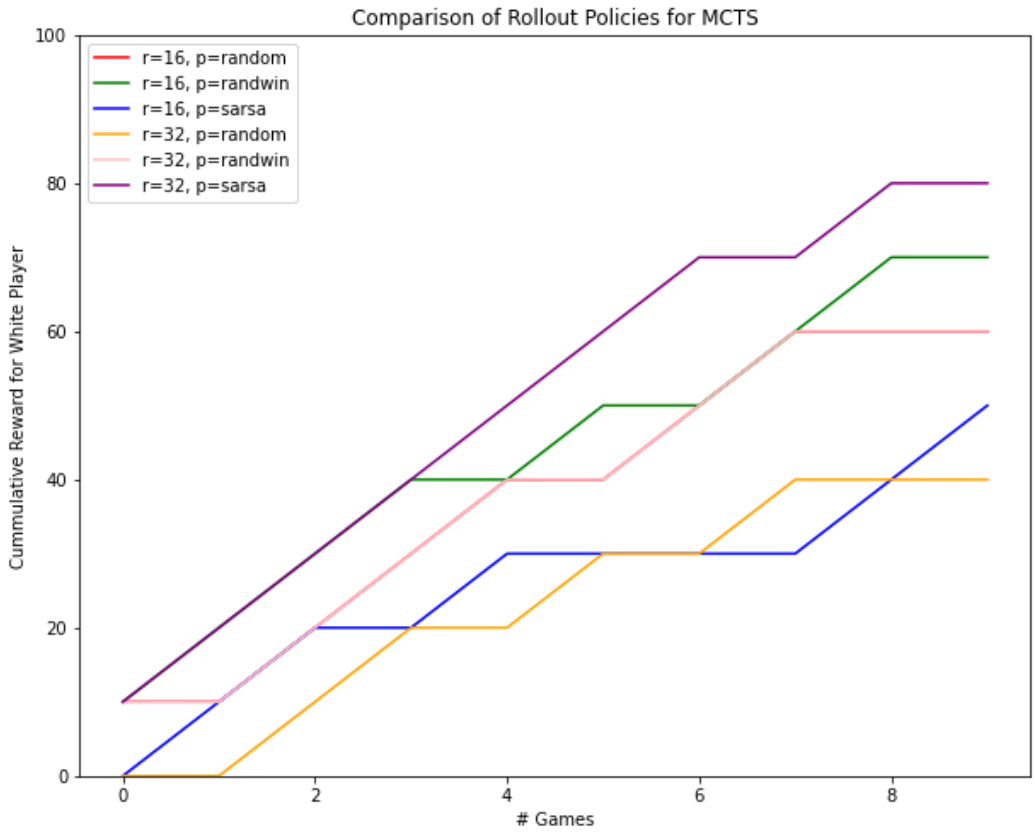
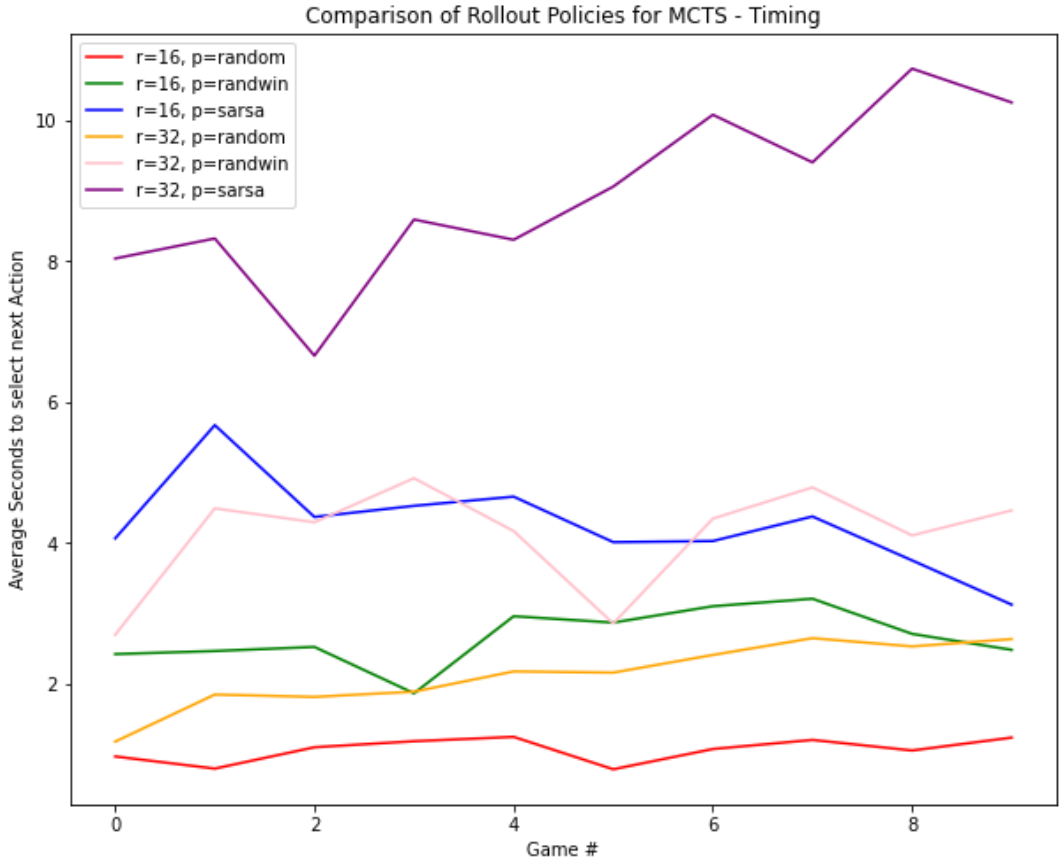

Comparison of Explorations and Rollout Count Hyperparamters for MCTS vs Random Policy
e=1.0, r=1
e=1.0, r=8
e=1.0, r=32
e=0.9, r=1
e=0.9, r=8
e=0.9, r=32
Cummulative Reward for White (-) & Black (- -) Players
0
100
200
300
400
500
600
0
20
40
60
80
100
Games

Figure 7


Comparison of Rollout Policies for MCTS
r=16, p=random
r=16, p=randwin
r=16, p=sarsa
r=32, p=random
r=32, p=randwin
r=32, p=sarsa
Cummulative Reward for White Player
Games
0
20
40
60
80
100
0
2
4
6
8


Comparison of Rollout Policies for MCTS - Timing
r=16, p=random
r=16, p=randwin
r=16, p=sarsa
r=32, p=random
r=32, p=randwin
r=32, p=sarsa
Average Seconds to select next Action
Game #
2
4
6
8
10
0
2
4
6
8