School of Electronic Engineering
and Computer Science

Final Report

Programme of study:
Computer Science with Year Abroad

Project Title:
Online Evolutionary Planning and Evolutionary MCTS for Turn-Based, Multi-Action Games

Supervisor:
Dr Diego Perez-Liebana

Student Name:
Amrit Singh

Final Year
Undergraduate Project 2020/21

Queen Mary
University of London

Date: 04/05/2021

Abstract

A.I. agents for strategy games have been a big source of interest, this is due to the complex nature of strategy games and their seemingly endless game states. In this report, we look at Online Evolutionary Programming and Evolutionary MCTS. These are two state-of-the-art methods that are being explored as agents for strategical, turn-based, and multi-action games. In this report, I will try to push the boundaries of what has already been done by implementing these agents to a new framework called Tribes. The following report describes each agent and the work that has been done around them. This project builds the foundations for future work to be done with these agents by implementing these agents to the framework and testing their current performance. We find that the performance of these agents, within the applied framework and considering their current state, perform reasonably well. However, there are certain areas, which will be discussed within this report, that require modification to enhance the performance of these agents.

Contents

Chapter 1: **Introduction**

Strategy games are a genre of games where a player's decision-making skills greatly determine the outcome of the game. Strategy games can be broken down to having a game state and an action space that allows a player to progress to a different game state. Many simple strategy games have a low branching factor and therefore a relatively small amount of game states to search. Therefore, algorithms can and have been designed to explore these game states effectively and efficiently allowing them to play the game effectively.

The most ideal scenario is where an algorithm knows every possible game state and therefore always knows the best possible move to make. This is somewhat possible for simpler games that have a small number of game states however, this will not be possible for most games as the number of states to remember is too large. Therefore, we need more efficient ways to search possible states and asses them to see which is the best one.

There are strategy games that have extremely high branching factors and intern a large action set. Existing algorithms are not as effective in these games and this will be further explained in the literature review in chapter 2 in this report. Therefore, such games are a source of interest to produce an effective agent that can play them.

Research is being done in this field by creating agents that can play in strategy games with high branching factors. However, the agents created are for a specific game and therefore optimised for that game. Therefore, these agents need to be applied to different games and tested against existing ones. By adopting these methods, different modifications can be explored to test the limitations and capabilities of these agents.

The report aims to create an *Online Evolutionary Planning* (Justesen, Mahlmann and Togelius, 2016) and *Evolutionary MCTS* (Baier and Cowling, 2018) agents for *Tribes* (Perez-Liebana et al., 2020). These agents will then be put up against existing ones to test their performance and viability. Furthermore, the aim is to see what modifications need to be made to these agents so that they can handle a growing action sequence size.

This report has four main objectives:

- Design and implementation of an Online Evolutionary Planning (OEP) agent for Tribes
- Design and implementation of an Evolutionary Monte Carlo Tree Search (eMCTS) agent for Tribes
- Test and compare OEP and eMCTS with each and existing agents for the framework
- Propose modifications for OEP and eMCTS

With these objectives in mind, this project aims to answer if it is possible to apply OEP and eMCTS to a game with a variable action set. These two questions are important as the games were originally designed and made for games with a fixed number of actions per turn. Thus, the next obvious question is how these agents perform against other agents created for those games.

The main contributions of this work would be testing OEP and eMCTS, and laying the groundwork for future research on these agents. If successful, these agents can be used in the industry to create agents in all sorts of strategy games.

This report first explores the work that has already been done in this field within chapter 2. Chapter 2.1 first outlines Tribes. This is the framework that we will be using in this project and what will be used to create agents to test. Chapter 2.2 and 2.3 then go over Online Evolutionary Planning and Evolutionary Monte Carlo Tree Search, respectively. These are the main agents that this project is focused on and these chapters will explain the work that has been done with them. Chapter 2.4 then goes into opponent-pruning paranoid search and how it may have useful implications in this project.

Chapter 3 investigates the difficulties with this project and how they were solved/avoided. Chapter 4 explains how the agents are implemented. Chapter 5 looks at the results from these agents. Chapter 6 discusses possible research that can be done in the future.

Chapter 2: **Literature Review**

This chapter will define previous studies that have been conducted in this field. 2.1, 2.2 and 2.3 will go over the literature that are most relevant to this topic area.

2.1 Tribes

2.1.1 Overview

Tribes is an implementation of the popular game *The Battle of Polytopia* a (Midjiwan AB 2016). This is a strategy game where players spawn into a map with an initial capital city, a unit and a particular research element researched on the research tree. There are two game modes that players can choose from: *Capitals* or *Score*. The first game mode has all players competing until they are the last civilisation remaining on the map. The other game mode is where players play for a given number of turns and the player with the highest score wins the game. The details of the game will be covered in section 2.1.2.

Tribes is a framework that has implemented this game for research purposes. The main reason that this game, and subsequently this framework, is excellent for this type of research is due to the huge branching factor this game has. Furthermore, the game has a lot of strategic depth - a player must manage their economy, research and units all at the same time.

2.1.2 Game Rules

This sub-section will go over the rules for the game and therefore explain why the game has a huge branching factor. This report will briefly go over the rules and structure, but full details of the framework can be found in the original paper (Perez-Liebana et al., 2020).

2.1.2.1 The Game Map

The game area is a grid of squares. These squares are called tiles and each tile represents a piece of land. Players can only see a portion of the tiles at a time. This is because the rest of the tiles are covered by a fog of war (This is a term used when an area of the map is hidden to the player until the player discovers it). Players can extend how much of the map they can see by placing a player on the border of the fog of war. The player can see an extra 1 tile into the fog of war when a unit is present, this is however changed to 2 tiles when the unit is on a mountain. This feature can also be turned off within Tribes to give the player full observability.

Each tile in the game has some properties. The first being what type of terrain the land it is. Each type presents a different opportunity and playstyle for players. For example, if the players city is surrounded by shallow water tiles, players would find it more beneficial to research technologies that use shallow water tiles rather than mountain tiles. Tiles can also hold resources for players to use. If a resource falls within the city's borders, the player can use it to grow that city. Players can also find villages around the map. These villages can then be captured by a player to expand their civilisation.

2.1.2.2 City, Tribe and Unit Actions

As a player, you can categorise your possible actions into three groups as done within the original paper.

Tribe Actions

The first set of actions to consider are tribe actions. These actions include researching technologies in your technology tree, building roads on a non-enemy tile and ending your turn. Tribe Actions are actions that will generally have an effect over the entire player civilisation.

City Actions

The second set of actions are called City actions. These are actions that occur within city borders. There are a lot of different actions available here, therefore I will generalise them to cover as many as possible.

These actions include building and destroying buildings. Special buildings and monuments can be built with special requirements set. You can also gather resources within city borders, spawn available units and level up cities.

Unit Actions

Unit action is any action that can be performed by a unit. These actions include moving, attacking, and capturing. Some units can also be upgraded and can also discover special ruins in the game space.

Key Takeaways of the Game Rules

With its rules into consideration, there are a few points to be made about this framework. The game gives the user a lot of flexibility with the way they want to play. Therefore, it is easy to see why it is a difficult game to make an AI for. There is also a point to be made with pruning. When we consider the action space for this game, we also consider possibilities where two sets of actions lead to the same end game state. A simple example would be if a player spawned a unit on city A and then capture a village into a city. This would be the same as capturing the village and then spawning a unit. Therefore, some sort of pruning algorithm could work when looking through for possible moves and actions so that we do not waste time or computational power in visiting these states.

Logically, an agent should only end their turn when there are no more moves that they can perform. This is generally a good rule of thumb as it makes sense to try and maximise your chances of winning by playing as many moves as you can.

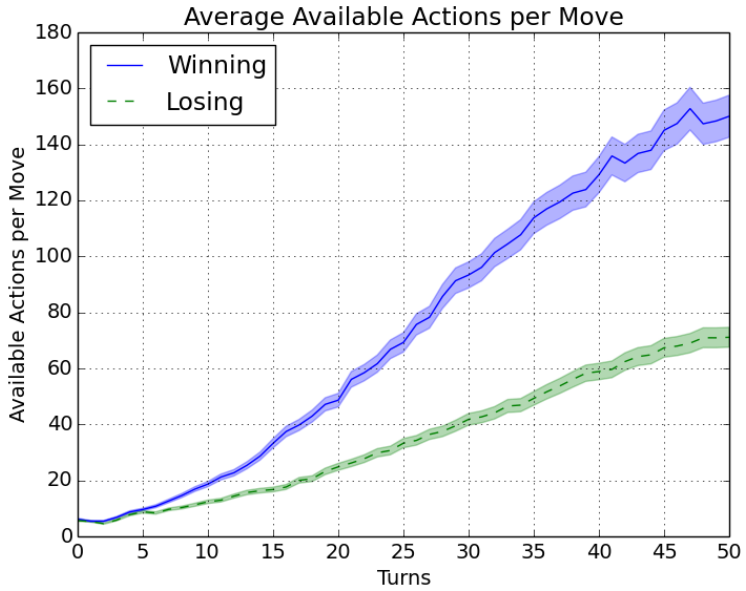

Figure 1. Graph of Available Actions per Move for the game Tribes (Perez-Liebana et al., 2020) – As shown by the graph, agents were more likely to win when they had more moves available to them.

2.1.3 Implemented Agents and Findings

After considering the game rules and set up in section 2.1.2, it is easy to see how this game can become quite complex. When the game first starts, the player only has a limited about of resources and therefore a limited number of moves they can make. However, when the user discovers more of the map and advance deeper into the game, it is easy to see how the possible number of moves grows very fast. This section will visit the agents that have been implemented into this framework and their initial findings.

The framework has 7 agents built for it; Do Nothing, Random, Rule-Based(RB), One-Step Look Ahead(OSLA), Monte Carlo(MC), Monte Carlo Tree Search(MCTS) and Rolling Horizon Evolutionary Algorithms(RHEA) (Perez-Liebana et al., 2020). Details about their implementation will not be discussed but their performances will be.

Overall, the Rolling Horizon Evolutionary Algorithm (RHEA) performed the best. This is probably due to RHEA best utilising Statistical Forward Planning (SFP). The random, RB and OSLA agents do not take full advantage of SFP. By nature, SFP is the process of looking ahead to possible future game states. This can be highly advantageous in strategy games when you are trying to do more complicated moves that take more than one atomic action to perform. OSLA does look ahead one move and determines the best possible outcome but you cannot be expected to perform any complicated strategies. However, it should be noted, even though the implemented Rule Based agent does not use SFP, it was the second-best performing agent by winning 65.8% (Perez-Liebana et al., 2020) of its game played. This does demonstrate that SFP is not required to make a good agent, however, a deep knowledge and understanding of the game is required to make a good RB agent. You also cannot account for every possible scenario for Rule-Based agents. The two other agents that tried to take advantage of SFP where the MC and MCTS agents. Both agents explore future game states in slightly different ways. MC explores future states at random and executes the set of actions that led to a state with the highest score. MCTS does it slightly more systematically. The MC and MCTS agents placed fourth (with 50.32% games won) and third with (60.44% of games won) (Perez-Liebana et al., 2020), respectively. MCTS is generally considered to be a “state of the art search framework” for games with branching factors of up to a few hundred (Baier and Cowling, 2018). Tribes is a game with an average branching factor of 10^{15} (Perez-Liebana et al., 2020) which is clearly far greater than a few hundred. Therefore, after

viewing the other agents in a little bit more detail, it is clear to see why RHEA may have performed better. The goal of this project will be to create an agent that has been researched in for other games with a vast action space and attempt to apply it to Tribes.

2.2 Online Evolutionary Planning

This section looks to introduce Online Evolutionary Planning (OEP) (Justesen, Mahlmann and Togelius, 2016) and the work that has been done with it so far.

2.2.1 How OEP Works

OEP is an approach that was inspired by RHEA. Evolutionary algorithms have been used to evolve controllers through a process of offline learning (Justesen, Mahlmann and Togelius, 2016). The idea behind these controllers is to evolve the controller while it plays the game.

OEP was first implemented for the game *HeroAIcademy* and we will discuss their implementation. For this explanation, it is important to understand that this is a game where a player has five action points and therefore can make a total of five moves per turn.

- The first step of this OEP implementation is to create a population of random individuals (Justesen, Mahlmann and Togelius, 2016). An individual is composed of random actions, using a forward model, that the player can make until they have no more action points remaining.
- All individuals are then rated using a heuristic function. This judges the actions on the game state they lead too. They are then ranked, and the worst individuals are then removed from the population.
- Out of the remaining individuals, each one is paired randomly with another and bred through uniform crossover.

Once your time allowance is then finished, the individual with the highest score is then executed.

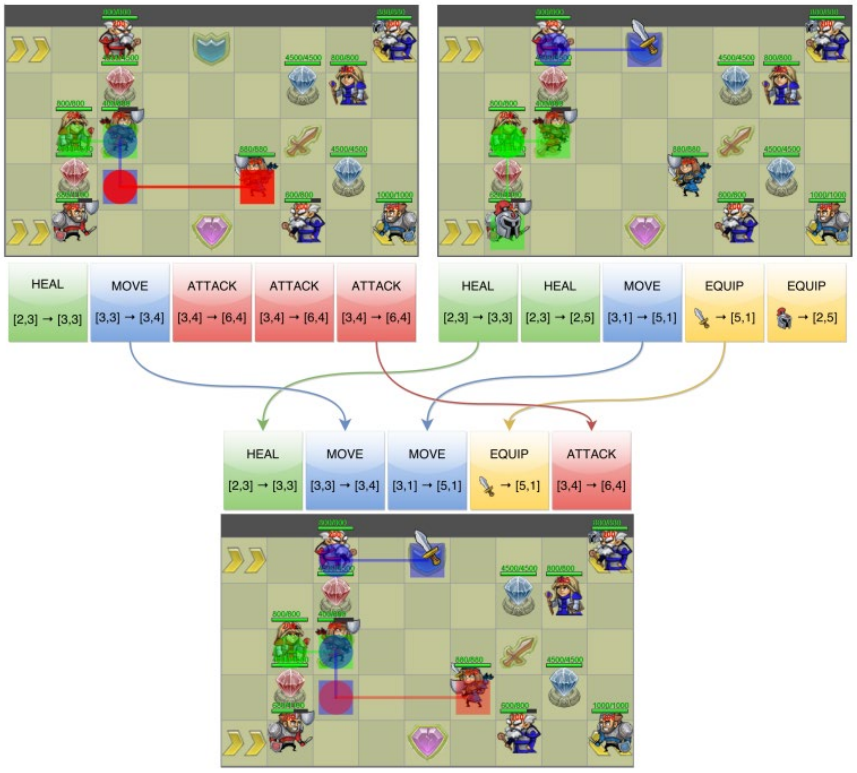

Figure 2. This is a visual representation of the uniform crossover in OEP implementation for Hero Academy. This visual representation was also taken from the same paper. (Justesen, Mahlmann and Togelius, 2016)

The original paper also further proceeds to explain that this can generate illegal action states. These illegal action sets are attempted to repair by using actions (genes) from the other parent and if it still does not work, then generate a random move that will fit. Some of the offspring have some of their genes mutated to introduce new actions into the gene pool. Again, if this led to an illegal action sequence, we then try to replace it with a random action that will keep it legal. It seems logical that the OEP is an agent that tries to find the best sequence of moves to perform. The algorithm is actively looking for the best sequence of steps to performs through evolution. The best action sets are always breaded to see if that action set can be used to create a better one. It is also therefore evident that the best genes are always passed on and therefore the best actions are generally likely to show up more in later generations. My main concern is that this method can easily be influenced by an action set that will help a player in the short term and not an action set that will help in the long term.

2.2.2 Potential Issues

A potential bottle neck for this method is the machine it is running on, but also the heuristic function that evaluates a given game state. OEP will only perform as well as the heuristic method that evaluates the population of action sets. If the heuristic method cannot evaluate game states well, it will cause OEP to generate a bad action set to perform. Therefore, the heuristic function should also be considered when evaluating this agent. The difficulty with OEP and implementing it in tribes is that Tribes has a variable number of moves that you can perform per turn. In its original paper OEP was implemented in a game that had a fixed number of moves per turn (5 moves per turn out of a variable number of options that you can perform). Tribes does not have any rule indicating that you can only perform a set number of moves in a turn that makes this project even more difficult.

2.3 Evolutionary MCTS

Evolutionary MCTS (eMCTS) (Baier and Cowling, 2018) is the other main algorithm that will be implemented into tribes. The following sections dive into eMCTS and explain the current state of this agent.

2.3.1 How eMCTS works

eMCTS was formulated by combining aspects of OEP and traditional MCTS. Traditional MCTS works by exploring a tree where nods represent game states and edges represent actions that get you from one game state to another. Once you reach a node that has not been discovered, you then expand that node and perform a rollout. This process is then repeated until the time has elapsed. eMCTS first requires you to start with a sequence of moves. Each node in an eMCTS tree is a move set and each edge is a mutation. This also therefore means that we no longer need to perform rollouts and evaluate them, we now evaluate the leaf nodes.

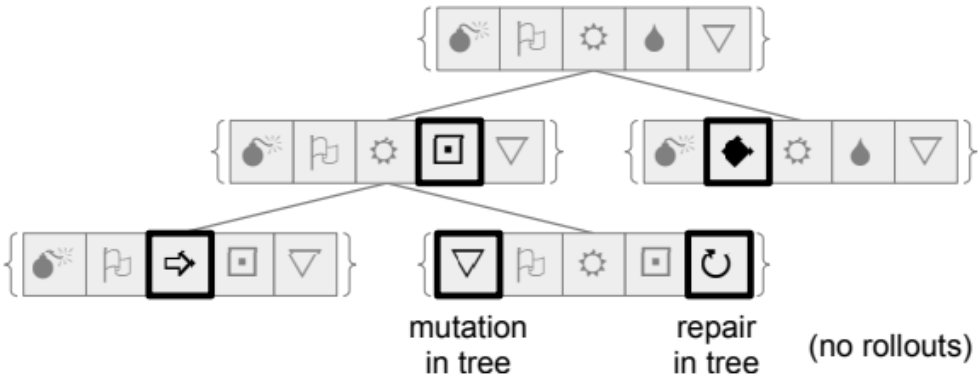

Figure 3. This is a visual example of eMCTS. (Baier and Cowling, 2018) – As described, eMCTS takes a sequence of actions and mutates them as you travel down the tree.

As the diagram shows, if a mutation causes an illegal action in the sequence, the illegal action is replaced with a legal move. This is the same system in place as OEP.

Bridge-burning was also applied in the original paper for this agent. This means that after every phase, the most promising mutation is executed. This helps reduce the enormous branching factor the tree would otherwise have.

2.3.2 Conclusion

In its original paper, eMCTS was compared against the same agents in OEP's original paper and an improved version of OEP called *greedy OEP* (Justesen, Mahlmann and Togelius, 2016) (Baier and Cowling, 2018). Where OEP was shown to be the best agent in its paper, eMCTS is shown to be even better. The paper shows results with two different parameters tweaked. The first being the time allocation and the second being the number of action points per turn.

Normal MCTS agents are generally great for games that only have one move that players can make at a time (Kozelek, 2009) (Ciancarini and Favini, 2010). eMCTS on the other hand, has taken the agent and has adapted it for a game with a fixed number of moves per turn. Therefore, this needs to be adapted for tribes where the number of moves is not one and nor is it fixed to an amount.

The biggest takeaway is that eMCTS outperformed vanilla OEP quite significantly in every significant area of testing. It should be noted that OEP did perform slightly better

with a larger time allowance. However, their improved *greedy OEP* seemed to be close to eMCTS. With a larger time, allowance, they were pretty much just as good as each other with eMCTS beating *greedy OEP* by 1.4-11.3% more (Baier and Cowling, 2018).

2.4 Opponent-Pruning Paranoid Search

The Battle of Polytopia (Midjiwan AB 2016) was designed to be played with 4 players on a single map. This fact further increases the difficulty of this game. Agents that use a forward model to go through future game states often do not simulate the opponents turn due to the complexity of the task. With three opponents, it is very unlikely for an agent to start to simulate their own turn again. This sub-section will go over the work that has been done to simplify this task and what potential it way has in this project.

Opponent-Pruning Paranoid Search (OPPS) (Baier and Kaisers, 2020) was a paper published that introduced a base technique of simulating opponent turns that then allows you to simulate your next turn. This approach a generalization of the search technique BRS+ (Esser et al., 2013), which was an improvement to BRS (Schadd and Winands, 2011) (Baier and Kaisers, 2020).

The idea of BRS+ is that all the opponents are somewhat working against the root player. This is done by searching through what moves the opponents can make and picking the opponent that can make the worst move for the player. This move is then performed with all the other opponents performing a move that is best for them. This essentially simplifies the problem of having many opponents to only one entity. OPPS generalises this by giving control of how “paranoid” and to the degree you want to prune the tree. Aspects of this approach are useful to look at, especially for tribes which has 4 players playing against each other. The downside of this approach is that it has been made for games with one move per turn. Tribes on the other hand has multiple moves per turn that a player can make and this approach will therefore need to be tweaked. Furthermore, this method can only be applied when we want to simulate future turns and with a game like tribes, it is unlikely we can go that deep. It is possible that OEP and eMCTS are adjusted so that they search for a shorter amount of time and then start looking into future turns with this approach. This will need to be tested further but can have a major impact when combined with these agents.

2.5 Other Relevant Research

Other agents have also been proposed in this field of interest and this section discusses these agents and why they may not be suitable for this context.

We first look at Portfolio Greedy Search (Churchill and Buro, 2013). This agent was created for large scale simulations where players may need to control many units. This, at first, looks like a good start for a game like tribes but it can only be useful for late game scenarios. Tribes is unlikely to have over 30 units per player for a reasonable size map compared to StarCraft (Blizzard Ent., 1998) which was the game this agent was initially tested on (Churchill and Buro, 2013)) and therefore this agent does not seem as useful. Aspects of their research and methods can be applied but for the sake of simplicity, they have been omitted.

Naïve MCTS (Ontanón, 2017) is also a viable option that we can utilise at as an agent. This approach uses a naïve assumption with a multi-armed bandits (MCB) approach to find the best move for a given set of rewards and reward functions (Sironi et al., 2018). It is possible to split tribes down so that we can teach an agent to make “good” moves

by rewarding them. This then becomes a more complicated challenge as Tribes has a lot of factors to account for. Therefore, this approach should not be favoured as it can get complicated quickly.

Chapter 3: **Difficulties and Solutions**

In this chapter, I will go over what difficulties this project faces and how I planned and ended up solving those problems.

3.1 Variable Node/Individual Size

One of the main challenges in this project is coming up with a solution to how to deal with a variable node size. This is a challenge because in the original papers (Justesen, Mahlmann and Togelius, 2016; Baier and Cowling, 2018), both agents were designed for a game that had a fixed node/individual size. This meant that every turn, the agent must make a fixed number of given moves (This was 5 in the original papers). This was because this was a core game mechanic where players can make up to 5 moves.

However, Tribes does not have this restriction in its game play. A player is allowed to make 4 moves in their first turn, 6 in their next and maybe 30 in their last turn. Thus, a new solution needs to be created so that these agents can run in this new framework. I have planned out two solutions to this problem for which I implemented the second.

3.1.1 Fixed Node Sizes - Original Method

The first implementation method I considered was to keep it like the original papers and keep a fixed node length that increases on a per-turn basis. The biggest advantage of using this method is that it keeps it as close as possible to the original paper and their methodology. This meant that all nodes/individuals created in each turn will all have the same length and as the game progressed, this fixed number will increase.

The biggest downside to this, and the main reason why I decided against this, was that there seems to be no fair way to decide on what this number should be. Based on what situation a player is in, this would dictate on what they can do and thus how many actions they can perform. Every game would be different to each other and there is no way to know what situation a player is in based on the turn number alone. One can argue that this can be done by performing a random set of actions and this could be our fixed size. This would work, but it suffered from two problems. The first is that it is not fair as one sequence of moves may lead you to performing more actions in the same turn, but this will not be allowed with a fixed length. The second problem is that the sequence may pre-maturely end the turn too soon as randomly picking actions in this framework includes the one to end your turn.

3.1.2 Variable Node Sizes – Implemented method

The second method planed was allowing any node/individual to be of any size in a given turn. This is a lot fairer than the previous version as it allows nodes to consider longer action sequences. The downside, however, is that I need to edit the way uniform crossover is performed for OEP. The original method was straight forward as both individuals are always have the same length and thus there is no issue when performing uniform crossover.

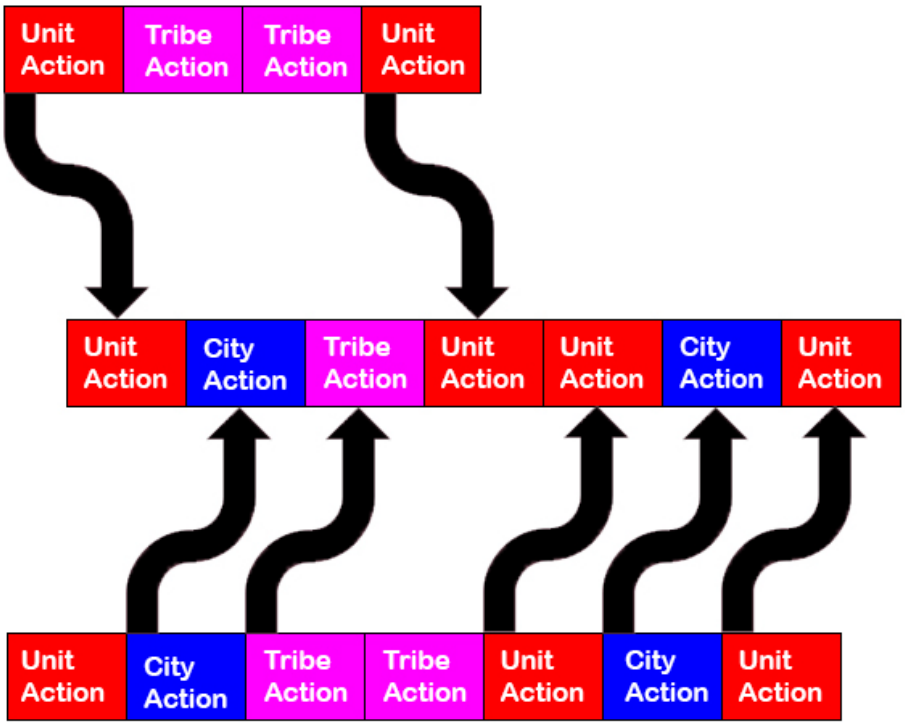

Figure 4. An example of the modified Uniform Crossover Used in this project.

The Figure above shows how my modified Uniform crossover works. I have modified it so that the uniform crossover works normally for the size of the smaller individual. In the example above, we have individuals of size 4 and 7. We perform Uniform crossover normally for the first 4 actions in each individual, respectively. The modification is that after the point where uniform crossover occurs normally, we take all the extra actions from the longer individual and add it to the child. The implication of this implementation is that it will naturally favours longer individuals. I chose to take this approach as the more moves you make, the more likely you are to perform well. This decision was also influenced by the original paper for Tribes (Perez-Liebana et al., 2020). Part of the findings in the original paper showed that winning agents on average had a larger branching factor per turn than the losing agent. This implies that winning agents have more moves available to them to perform. Thus, I decided to favour longer individuals so that they can potentially benefit from this.

3.2 Simulating Opponent Moves

In this section, I will go over my reasoning for why I made design decisions towards simulating opponents turns.

At first, I created these agents with no intention to simulate the opponents turns. This would seem like a disadvantage, but I was primarily trying to get the agents working well. Furthermore, I believed that an agent focusing on its own actions will perform better than an agent using resources to simulate what the opponent will do. This approach made the agent easier to implement, but difficult to change to accommodate for this. The extra layer of difficulty in this is that you need to consider that the opponent will be playing for themselves rather than for you. It will be relatively easy for me to add the opponents moves as extra actions within the individual, but OEP will only consider sets of actions that the opponent can make that will be beneficial to you as a player. The reality is that

the opponents will likely not make those moves and make moves that work against the player and this can potentially decrease the performance of the agent.

The correct approach would be to have the opponents turn as a separate individual that you are trying to maximise for them. Thus, you are likely to see what moves that opponents are likely to make and thus can forward plan for yourself. This approach is strong on paper but suffers from two problems. Firstly, Tribes is a game where up to 4 players can play against each other. This therefore means that you will need to simulate 3 other opponents before you can get to yourself and consider what you would need to do. As we talked about in the Literature Review, we can use OPPS so make the opponents turns easier to simulate. This would fix this problem relatively well but the second problem, related to this one, will still cause a decrease in performance. The second problem is that simulating opponents turn takes resources to do. The ideal goal is that you have infinite time and resources and thus your agent can theoretically reach the most optimum actions to make. This is not a viable, and thus Tribes has 3 options to choose when to stop the agent and return its best actions: time, number of iterations and number of forward model calls. Other than number of iterations, the others will greatly suffer when trying to simulate opponents turns as they will take up those resources and thus fewer iterations could be made decreasing your performance. This has not been tested in this project, but this is definitely an area for future research with these agents and how they can be possible improved.

3.3 Metrics for Comparing Agents

The end goal of this project is to see the viability of these agents implemented in a new game. Thus, this section will outline what is being done to ensure that there are fair comparisons between the agents.

The framework I have been using already has a tournament system implemented. This system allows two agents to play against each other on pre-defined level seeds and settings. I will be using the same settings as the ones used in the original paper (Perez-Liebana et al., 2020). This means that there are 25 seeds where each seed will be played 20 times which comes to 500 games. Agents do swap positions and tribes making it as fair as possible. We will also be using the same game mode as the original paper. This was “Capitals”. This game mode has agents playing until one is wiped out or the game reaches turn 50 (whichever comes first).

3.3.1 Parameters and Stop Type

For a fair comparison, all agents will be using the same heuristic. The framework has three different heuristics implemented; we would be using the “Diff Heuristic”. This is also the same heuristic that was used in the original paper. This heuristic evaluates the a given state to an original and returns a score that is dependent on the parameters in the heuristic. Having the same heuristic means that every agent will evaluate each game state equally and thus no agent should have an unfair advantage.

It was also ensured that every agent uses the stop type “FM Calls”. FM Calls refers to the amount of forward model calls the agent makes. All the agents that use these inherit basic parameters one of which is the limit of FM Calls. This limit dictates the amount of times an agent can use an FM Call. Therefore, any test will look at how efficiently an agent uses these calls and thus which is better that way. Using time may also have been fair but an agent may use 5000 calls in that time whereas another could use 2000 which does not seem like a fair comparison.

3.3.2 Same Node/Individual Sizes

Originally, OEP and EMCTS were created with the freedom to have nodes/individuals of unlimited size. This then meant that in turn 1, they could have one of size three or four, and by turn 30, they could have a node or individual with a size of 40.

After performing some tests, OEP and EMCTS were performing well. However, I was informed that all other agents are capped at 20. This meant that my agents could have nodes at around 30 whereas RHEA (as an example) was stuck at 20. This was a clear unfair advantage to my agents and thus a cap was also introduced that is set to 20 ensuring that they also do not exceed 20.

Chapter 4: **Final Agents**

In this chapter, we will go over the final agents made and parts of the code that perform specific functions.

4.1 OEP

This agent created is composed of three classes: OEPAgent, Individual, OEPParams. The agent class is responsible for the bulk of the work of performing what needs to be done and returning actions to be performed. The Individual class is a representation of the list of actions and has the functionality of storing the end game state and value of the node.

Both agents have a main act function with the signature

```
public Action act(GameState gs, ElapsedCpuTimer ect)
```

This is how a given agent in the framework returns an action it wants to perform. Therefore, this method must perform two important functions. The first is that it must find the sequence of moves it wants to perform and the second is that it must return actions from a selected sequence one at a time.

Thus, this is a long method broken down into sections that perform the following tasks:

- Return the next action if in the process of doing so
- Otherwise, create a new population of Individuals
- Until your selected budget has expired:
  - Perform Uniform Crossover
  - Evaluate all Individuals
  - Kill the lower end of the individuals
  - Fill the rest of the population with random individuals

Budget checks do occur between stages, and at the end of every iteration. The code for this method is not crucial but how they perform their tasks is and is discussed below.

4.1.1 Random Actions

The method ‘randomActions’ is a simple method that creates an individual consisting with random actions that are available. The method also ensures that the size of the individual does not exceed the defined limit in its parameters. The method was heavily inspired by other agents in the framework (Perez-Liebana et al., 2020).

```
1:    private Individual randomActions(GameState gs){
2:         ArrayList<Action> individual = new ArrayList<>();
3:         while ((!gs.isGameOver() && (gs.getActiveTribeID() ==
getPlayerID())) && (individual.size() < (params.NODE_SIZE-1))){
4:             ArrayList<Action> allAvailableActions = this.allGoodActions(gs,
m_rnd);
5:             Action a =
allAvailableActions.get(m_rnd.nextInt(allAvailableActions.size()));
6:             if(!(a.getActionType() == Types.ACTION.END_TURN)){
7:                 advance(gs, a);
8:                 individual.add(a);
9:             }else if(allAvailableActions.size() == 1){
10:                advance(gs, a);
11:                individual.add(a);
```

```
12:             }
13:         }
14:         //if one less then max node size
15:         if(individual.size() == (params.NODE_SIZE-1) &&
(individual.get((params.NODE_SIZE-2)).getActionType() !=
Types.ACTION.END_TURN)){
16:             ArrayList<Action> allAvailableActions = this.allGoodActions(gs,
m_rnd);
17:             Action end = null; ;
18:             for(Action a : allAvailableActions){
19:                 if(a.getActionType() == Types.ACTION.END_TURN){
20:                     end = a;
21:                     break;
22:                 }
23:             }
24:             individual.add(end);
25:         }
26:         Individual in = new Individual(individual);
27:         in.setGs(gs.copy());
28:         return in;
29:     }
```

4.1.2 Uniform Crossover

The next functions that will be discussed will be with how the agent performs uniform crossover. There are 4 methods associated with performing this action.

The first one that we will look at is ‘procreate’. This method is responsible for deciding what two individuals procreate. Here you would notice that tournament Selection is used to pick which individuals would perform uniform crossover with each other. The reasons and the code associated with this will be discussed in the evaluation. After selection, individuals are divided into groups so that individual at position x in group 1will cross with individual at position x in group 2.

```
1:     private ArrayList<Individual> procreate (GameState gs,
ArrayList<Individual> population){
2:         ArrayList<Individual> group1 = new ArrayList<>();
3:         ArrayList<Individual> group2 = new ArrayList<>();
4:
5:         while(population.size() > 0){
6:             Individual[] winners = tournamentSelection(population);
7:
8:             group1.add(winners[0]);
9:             group2.add(winners[1]);
10:
11:            population.remove(winners[0]);
12:            population.remove(winners[1]);
13:        }
14:        for(int i = 0; i < group1.size(); i++){
15:            population.add(crossover(gs.copy(), group1.get(i),
group2.get(i)));
16:        }
17:        return population;
18:    }
```

The next function is the one that performs the crossover. As stated in the last chapter, the agent can consider two individuals of different lengths. It does this by finding the smaller individual and then performing uniform crossover for that length. The agent then adds all actions remaining for the longer individual.

```
1:  private Individual crossover(GameState clone, Individual individual1,
Individual individual2){
2:          ArrayList<Action> in1 = individual1.getActions();
3:          ArrayList<Action> in2 = individual2.getActions();
4:
5:          ArrayList<Action> child = new ArrayList<>();
6:          boolean ind1 = true;
7:          boolean sameSize = false;
8:          int smallSize = in1.size();
9:          if(smallSize > in2.size()){
10:             ArrayList<Action> temp = in1;
11:             in1 = in2;
12:             in2 = temp;
13:             smallSize = in1.size();
14:         }
15:         else if(smallSize == in2.size()){
16:             sameSize = true;
17:         }
18:         smallSize --;
19:         if(!sameSize && in1.size() > 0){in1.remove(in1.size() - 1);}
20:         int in1amount =(int)(in1.size() / 2);
21:         int in2amount = in1.size() - in1amount;
22:         for(int i = 0; i < in2.size(); i++){
23:             if(i >= smallSize){
24:                 child.add(in2.get(i));
25:             }else{
26:                 if(in1amount == 0){
27:                     child.add(in2.get(i));
28:                 }else if(in2amount == 0){
29:                     child.add(in1.get(i));
30:                 }else{
31:                     int temp = m_rnd.nextInt(100);
32:                     if(temp < 50){
33:                         child.add(in1.get(i));
34:                         in1amount--;
35:                     }else{
36:                         child.add(in2.get(i));
37:                         in2amount--;
38:                     }
39:                 }
40:             }
41:         }
42:         child = repair(clone, child);
43:         Individual in = new Individual(child);
44:         in.setGs(clone);
45:         return in;
46:     }
```

This method would cause a lot of errors. Thus, at the end of this previous method, we call a method called ‘repair’. The role of this method is to take a given individual and fix it so that it is legal. If it finds an illegal move, it will replace it with a legal one. This method is also then responsible for calling mutations with a given probability.

```
1:     private ArrayList<Action> repair(GameState gs, ArrayList<Action>
child){
2:         ArrayList<Action> repairedChild = new ArrayList<>();
3:         boolean mutated = false;
4:         for(int a = 0 ;a < child.size(); a ++) {
5:             if(!(gs.getActiveTribeID() == getPlayerID())){return
repairedChild;}
6:             int chance = m_rnd.nextInt((int)(params.MUTATION_RATE * 100));
7:             if((m_rnd.nextInt(100) < chance) && !mutated){
8:                 Action ac = mutation(gs);
9:                 advance(gs, ac);
10:                repairedChild.add(ac);
11:                mutated = true;
12:            }else{
13:                GameState copy = gs.copy();
14:                boolean added = false;
15:                try {
16:                    boolean done = checkActionFeasibility(child.get(a),
gs.copy());
17:                    if (!done) {
18:                        ArrayList<Action> allAvailableActions =
this.allGoodActions(gs.copy(), m_rnd);
19:                        Action ac =
allAvailableActions.get(m_rnd.nextInt(allAvailableActions.size()));
20:                        advance(gs,ac);
21:                        repairedChild.add(ac);
22:                        added = true;
23:                    } else {
24:                        repairedChild.add(child.get(a));
25:                        added = true;
26:                        advance(gs, child.get(a));
27:                    }
28:                } catch (Exception e) {
30:                    if(added){repairedChild.remove(repairedChild.size()-
1);}
31:                    gs = copy;
32:
33:                }
34:            }
35:        }
36:        if(!(gs.getActiveTribeID() == getPlayerID())){return
repairedChild;}
37:        ArrayList<Action> allAvailableActions = this.allGoodActions(gs,
m_rnd);
38:        Action end = null; ;
39:        for(Action a : allAvailableActions){
40:            if(a.getActionType() == Types.ACTION.END_TURN){
41:                end = a;
42:                break;
43:            }
44:        }
45:        repairedChild.add(end);
46:        return repairedChild;
47:    }
```

```
1:    private Action mutation(GameState gs){
2:       ArrayList<Action> allAvailableActions = this.allGoodActions(gs, m_rnd);
3:                                                                    return
allAvailableActions.get(m_rnd.nextInt(allAvailableActions.size()));
4:    }
```

All major functions have been discussed for OEP. The rest of the functions responsible for various tasks like evaluation of individuals or re-filling the population are very straight forward or use functions in the framework that is already provided to complete their functionality.

4.2 EMCTS

By nature, EMCTS has a lot of similar functions and methods to OEP. Thus, these similarities would be discussed but not explicitly shown here.

This Agent is composed of three classes: EMCTSAgent, EMCTSTreeNode and EMCTSParams. The agent class is the one which is used to perform all the tasks required and perform actions. Params is a very basic class that stores variables that can tweak the performance of the agent. The TreeNode class is a simple class used to simulate a node in a EMCTS Tree. This stores the following main information and methods to do with them:

```
1: private ArrayList<Action> sequence;
2: private EMCTSTreeNode parent;
3: private ArrayList<EMCTSTreeNode> children;
4: private double value; // the heuristic value of the node
5: private double score;// the exploration score of the node
6: private int times_visited;
7: private GameState gs;
```

As stated earlier, EMCTS has an 'act' method responsible for performing all relevant actions and returning actions one by one for them to be executed. Thus, this method has the following structure:

- Return the next action if in the process of doing so
- Otherwise, create a new root node to expand from
- Until your selected budget has expired:
  - Find a node to expand
  - Expand that node until a given depth
  - Update Scores and Check Values

The method for EMCTS to create its initial node is also called 'randomActions'. Like OEP, it returns a list of random legal actions that the agent can perform. This method is almost very similar as the one in OEP and thus will not be shown here.

4.2.1 Node to Expand

One of the main problems in EMCTS is how to figure out what node needs to be expanded. This was solved by using the score value of the nodes. This value was based on the formula: $Value + bias \times \sqrt{\log\left(\frac{No.\ times\ Parent\ Visited}{times\ Visited}\right)}$. The method is not responsible for calculating or setting this score. It just goes down the tree looking for the biggest score and returning that node. Thus, this method is purely searching for a node with the biggest score.

```
1:      private EMCTSTreeNode nodeToExpand() {
2:          ArrayList<EMCTSTreeNode> children = root.getChildren();
3:
4:          EMCTSTreeNode toMutate = null;
5:
6:          if (children.size() == 0) {
7:              toMutate = root;
8:          } else {
9:              boolean found = false;
10:             double nodeScore = 0;
11:             EMCTSTreeNode nodeOn = root;
12:             nodeScore = nodeOn.getScore();
13:             while (!found) {
14:                 if (nodeOn.getChildren().size() == 0) {
15:                     found = true;
16:                     toMutate = nodeOn;
17:                 } else {
18:                     boolean bigger = false;
19:                     for (EMCTSTreeNode node : nodeOn.getChildren()) {
20:                         if (node.getScore() > nodeScore) {
21:                             bigger = true;
22:                             nodeOn = node;
23:                             nodeScore = node.getScore();
24:                         }
25:                         if (!bigger) {
26:                             found = true;
27:                             toMutate = nodeOn;
28:                         }
29:                     }
30:                 }
31:             }
32:         }
33:         return toMutate;
34:     }
```

4.2.2 Mutate and Repair

The mutate function for EMCTS is more complex than the one for OEP. This is because this method finds a random action in the node's sequence, then finds a new action to mutate it to and then call the repair function to fix any illegal moves.

The repair function however is similar to the one in OEP and thus will not be shown here.

```
1:      private EMCTSTreeNode mutate(EMCTSTreeNode node, GameState gs) {
2:          GameState clone = gs.copy();
3:          ArrayList<Action> seq = node.getSequence();
4:          int moveToMutate = m_rnd.nextInt(seq.size());
5:          ArrayList<Action> newSeq = new ArrayList<>();
6:          for (int i = 0; i < moveToMutate; i++) {
7:             advance(gs, seq.get(i));
8:             newSeq.add(seq.get(i));
9:          }
10:         ArrayList<Action> allAvailableActions = this.allGoodActions(gs,
m_rnd);
11:         Action a =
allAvailableActions.get(m_rnd.nextInt(allAvailableActions.size()));
12:         newSeq.add(a);
13:         for (int i = moveToMutate + 1; i < seq.size(); i++) {
14:             newSeq.add(seq.get(i));
15:         }
16:
17:         newSeq = repair(newSeq, clone);
18:         EMCTSTreeNode newNode = new EMCTSTreeNode(newSeq, node);
19:         node.addChild(newNode);
20:         return newNode;
21:     }
```

Chapter 5: **Evaluation**

In this chapter, we will look over the results obtained from OEP and eMCTS and discuss the results. First, we will look over the results obtained by playing OEP and eMCTS against the other agents in the framework. We will also discuss the improvements made to OEP over the base version from its original paper.

5.1 Results Overview

		Simple	RHEA	OSLA	MCTS	OEP	EMCTS
OEP	% win rate	40.6	33	48.8	60.2	-	61
	Wins	203	165	244	301	-	305
	avg Score	8498.1	9568.55	9909.2	10714.42	-	10964.83
	avg Tech	20.64	21.4	22.02	22.16	-	23.25
	avg Cities	1.48	2.07	2.19	2.74	-	2.53
	avg Pro	12.9	17.25	18.27	22.49	-	21.43
EMCTS	% win rate	36	18.6	37.4	43.6	39	-
	Wins	180	93	187	218	195	-
	avg Score	7913.35	9160.91	9505.58	9630.06	9503.9	-
	avg Tech	20.98	21.8	22.23	21.69	22.56	-
	avg Cities	1.41	1.89	1.99	2.29	1.94	-
	avg Pro	11.48	15.41	16.67	18.45	16.18	-

Figure 5. Results for OEP/eMCTS against other agents

Figure 5 above shows OEP and eMCTS score's against the other agents at the top. The results above seem to indicate that OEP and EMCTS are not strong agents, but this may not be the case after looking into the other statistics.

		Simple	RHEA	OSLA	MCTS
OEP	% win rate	59.4	67	51.2	39.8
	Wins	297	335	256	199
	avg Score	7789.08	12220.01	10005.86	8949.84
	avg Tech	17.87	23.8	22.27	22.34
	avg Cities	3.01	2.34	2.16	1.49
	avg Pro	21.14	19.25	17.19	10.76
EMCTS	% win rate	64	81.4	62.6	56.4
	Wins	320	407	313	282
	avg Score	8150.63	13300.69	10869.6	9672.83
	avg Tech	18.43	23.81	22.84	22.37
	avg Cities	3.08	2.59	2.46	1.89
	avg Pro	22.44	22.14	20.31	13.74

Figure 6. Results for other agents against OEP/eMCTS

The most interesting result to point out here is between OEP and the Simple agent. OEP and the Simple agent are quite even with regards to the number of wins each agent got with Simple being slightly better. When assessing the average score for each agent through the 500 games, you will see that OEP had a higher average then Simple. This means that OEP was very good at achieving a higher score, but something else was happening that was causing it to lose.

After looking at all the other results, another pattern appears. It commands that agents capture, on average, 2 or more cities. The exception to this rule is when one agent beats another by a significant amount. However, this is not the case for OEP and Simple. OEP is only getting an average of 1.48 cities, whereas Simple has an average of 3.01.

It seems like the Simple agent is excellent at combat and thus war at this game. As simple is a rule-based agent, it can be coded to be as aggressive as you would like it to be. Thus, it seems that OEP and eMCTS do not seem to see the long-term benefit of fighting and thus do not engage in it as much. The benefit of going to war in these games is that the opponent then needs to focus their resources on getting new troops and thus cannot focus on expanding their empire or other research that may improve score more significantly.

RHEA does simulate opponent moves too. This may be an explanation to how it outperforms Simple as it can see the benefits of fighting and capturing cities. This said, its sill beats OEP based on score quite significantly and not by capturing cities.

eMCTS seems to be the weakest agent. This likely due to eMCTS suffering from the same problem as OEP by not simulating opponents turn. Furthermore, eMCTS did not have any improvements added to it like OEP. The slandered version was implemented from its original paper, thus no “Bridge Burning” (Baier and Cowling, 2018).

Another reason why these agents may not be performing as well may be due to their implementations. Attempts were made to have OEP and eMCTS as efficient as possible with how they use FM calls. Originally, every time an individual in OEP was evaluated, FM calls were used to get to the game state that an individual leads to. A change was implemented so that the end game state was saved in the individual when it is created thus eliminating the need to use these extra FM calls. A similar attempt for eMCTS, but this was unsuccessful and could also be a reason as it is not performing as well.

One final factor that can be the cause of these agents underperforming is that they may not be using optimal parameters. OEP has many parameters that can be tweaked (Parameter : default value used, Population Size : 50, Mutation Rate : 0.1, Kill Rate : 0.5, Tournament Size : 0.5). Tweaking these parameters can help this agent perform much better. The same is true for eMCTS. It has two main parameters (Parameter : default value used, Bias : 1, depth : 10).

5.2 OEP Improvements

Initially, OEP was weaker than seen in the results in the previous section. Thus, some improvements were implemented in an attempt to make it stronger. These improvements will be discussed in the following sections and their code respectively. The results without these improvements can be found in Appendix A. Both improvements were inspired by RHEA doing them.

The first improvement made to OEP is that we implemented Tournament Selection (Miller, B. and D. Goldberg, 1995). This is the process on how we decide what individuals would crossover with each other. This was seen in the previous chapter under the

method “Procreate”. This was a logical improvement as OEP naturally choses two individuals at random to crossover. Tournament selection, on the other hand, states that we hold two tournaments of random individuals and the best one from each will be chosen to crossover. This method has the chance that the best two individuals are chosen together but it is not guaranteed. Individuals are put into tournaments randomly and the best individual is based on their heuristic score.

```
1:      private Individual[] tournamentSelection(ArrayList<Individual> pop){
2:          if(pop.size() < params.TOURNAMENT_SIZE){
3:                      if(pop.size()  ==  1){return  new  Individual[]{pop.get(0),
pop.get(0)};}
4:              Collections.sort(pop);
5:              return new Individual[]{pop.get(0), pop.get(1)};
6:          }
7:          ArrayList<Individual> tournament1  =  new ArrayList<>();
8:          while(tournament1.size() < params.TOURNAMENT_SIZE){
9:              Individual temp = pop.get(m_rnd.nextInt(pop.size()));
10:              if(!(tournament1.contains(temp))){tournament1.add(temp);}
11:           }
12:         Collections.sort(tournament1);
13:
14:         ArrayList<Individual> tournament2  =  new ArrayList<>();
15:         while(tournament2.size() < params.TOURNAMENT_SIZE){
16:             Individual temp = pop.get(m_rnd.nextInt(pop.size()));
17:             if(!(tournament2.contains(temp))){tournament2.add(temp);}
18:         }
19:         Collections.sort(tournament2);
20:         return new Individual[]{tournament1.get(0), tournament2.get(0)};
21:     }
```

The code above is what is used for tournament selection. This seems to be a clear improvement as it is very likely that in the base version, a very good individuals crosses with a very bad one and that generates an average individual. This version of tournament selection is without replacement; therefore, every individual gets the opportunity to crossover. The second improvement that was implemented was a change to the kill method of the agent. Originally, the agent is just supposed to kill a portion of the population and then replace them with random ones. This was changed to how RHEA performed this.

The method first takes the population and finds the best individual. We then shift that individual, so all its actions move one space forward and we then mutate it so that half of the population is a variant of that individual. The other half are then random individuals. This approach is similar to Greedy OEP (Baier and Cowling, 2018). Here, we are favouring high scoring individuals and combining eMCTS and OEP in a way that half of the population is a search to find a better variant of that individual.

Two main methods are used in this process, the first is a “shiftPop” that shifts the best individual and a shift method that performs the rest of the tasks. A new mutate method was also created and that can be found in Appendix B.

```
1:   private ArrayList<Individual> shiftPop(GameState gs, ArrayList<Individual>
population){
2:         ArrayList<Individual> newPop = new ArrayList<>();
3:         shift(gs.copy(), population.get(population.size()-1));
4:         newPop.add(population.get(population.size()-1));
5:         for(int i = 1; i < (params.POP_SIZE/2); i++){
6:              Individual ind = mutateInd(population.get(population.size()-1),
gs.copy());
7:             newPop.add(ind);
8:         }
9:         for(int i = newPop.size(); i < params.POP_SIZE; i++){
10:            newPop.add(randomActions(gs.copy()));
11:        }
12:
13:        return newPop;
14:    }

1:   private void shift(GameState gs, Individual individual){
2:         GameState clone = gs.copy();
3:         individual.shift();
4:         boolean feasible = true;
5:         int j = 0;
6:         while(feasible && j < individual.getActions().size())
7:         {
8:             Action act = individual.getActions().get(j);
9:             feasible = checkActionFeasibility(act, gs);
10:            if(feasible)
11:            {
12:                advance(gs, act);
13:                j++;
14:            }
15:        }
16:        int i = j;
17:        while((!gs.isGameOver() && (gs.getActiveTribeID() == getPlayerID()))
&& i < params.NODE_SIZE)
18:        {
19:                 ArrayList<Action>allAvailableActions = this.allGoodActions
(gs.copy(), m_rnd);
20:                          Action  ac=allAvailableActions.get  (m_rnd.nextInt
(allAvailableActions.size()));
21:            individual.getActions().add(ac);
22:            advance(gs, ac);
23:            i++;
24:        }
25:        double score = heuristic.evaluateState(clone,gs);
26:        individual.setValue(score);
27:        individual.setGs(gs.copy());
28:    }
```

These improvements overall, helped OEP perform better, especially against the Simple and RHEA Agents.

Chapter 6: **Conclusion**

Overall, OEP and eMCTS are two agents with potential to be far better. This project has created the foundations for future projects to expand on what has already been created and improve upon. Throughout the report, I have mentioned what areas of these agents can be improved and that will be summarised here.

Further work and research can be done with regards to having OEP and eMCTS extend further into the future and simulate opponent moves. This is where OPPS would be used as discussed in the background research chapter. A further avenue of research would be how these two agents can be changed to use this method and see how well it would perform. A start would be a simple implementation of OEP or eMCTS will look through opponent moves that favour them. This of course is not ideal but is a good start to look for performance boosts.

The biggest improvement needed for this implementation of OEP and eMCTS would be to find the most optimal parameters and investigate why they are the most optimal parameters. As mentioned before, the parameters used in my experiments were originally thought of as good bases but are by no means optimal. Changing these values can be the sole reason for these agents to start out performing the others on the framework.

One area of interest could be looking into other metrics to compare these agents. Currently, FM calls are used to stop an agent. Changing this to time or number of iterations can be of interest as this would not consider efficiency of using FM calls, but how they perform overall. People in industry may be interested to see how they can perform with 1 second of run time or 10 second etc. These may be areas where these agents excel in compared to others.

After these modifications have been tested, one of the last areas to test would be to test other general improvements like bridge burning eMCTS or greedy OEP. The end goal of these agent would be to see which are most viable in what scenarios and thus how they can be implemented in industry. As it stands right now, the most reliable agent still seems to be some rule-based agent. The downside to such an agent is of course the time and expert knowledge it takes to create them. But given the resources, you can definitely make very strong rule-based agents.

References

1. (Justesen, Mahlmann and Togelius, 2016) Justesen, N., Mahlmann, T. and Togelius, J., 2016, March. Online evolution for multi-action adversarial games. In *European Conference on the Applications of Evolutionary Computation* (pp. 590-603). Springer, Cham.

2. (Baier and Cowling, 2018) Baier, H. and Cowling, P.I. (2018). Evolutionary MCTS for Multi-Action Adversarial Games. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*.

3. (Perez-Liebana et al., 2020) Perez-Liebana, D., Hsu, Y.-J., Emmanouilidis, S., Dewan Akram Khaleque, B. and D. Gaina, R. (2020). Tribes: A New Turn-Based Strategy Game for AI Research. *(2020): Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1).

4. (Midjiwan AB 2016) Midjiwan AB. 2016. The Battle of Polytopia.

5. (Konami 2016) Konami. 2016. Yu-Gi-Oh! Duel Links.

6. (Baier and Kaisers, 2020) Baier, H. and Kaisers, M., 2020, September. Opponent-Pruning Paranoid Search. In International Conference on the Foundations of Digital Games (pp. 1-7).

7. (Esser et al., 2013) Esser, M., Gras, M., Winands, M.H., Schadd, M.P. and Lanctot, M., 2013, August. Improving best-reply search. In *International Conference on Computers and Games* (pp. 125-137). Springer, Cham.

8. (Schadd and Winands, 2011) Schadd, M.P. and Winands, M.H., 2011. Best reply search for multiplayer games. *IEEE Transactions on Computational Intelligence and AI in Games*, *3*(1), pp.57-66.

9. (Ciancarini and Favini, 2010) Ciancarini, P. and Favini, G.P., 2010. Monte Carlo tree search in Kriegspiel. *Artificial Intelligence*, *174*(11), pp.670-684.

10. (Kozelek, 2009) Kozelek, T., 2009. Methods of MCTS and the game Arimaa.

11. (Churchill and Buro, 2013) Churchill, D. and Buro, M., 2013, August. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)* (pp. 1-8). IEEE.

12. (Sironi et al., 2018) Sironi, C.F., Liu, J., Perez-Liebana, D., Gaina, R.D., Bravi, I., Lucas, S.M. and Winands, M.H., 2018, April. Self-adaptive mcts for general video game playing. In *International Conference on the Applications of Evolutionary Computation* (pp. 358-375). Springer, Cham.
13. (Blizzard Ent., 1998) Blizzard Entertainment. 1998. StarCraft.

14. (Ontanón, 2017) Ontanón, S., 2017. Combinatorial multi-armed bandits for real-time strategy games. *Journal of Artificial Intelligence Research*, *58*, pp.665-702.

15. (Miller, B. and D. Goldberg, 1995) Miller, B. and D. Goldberg. “Genetic Algorithms, Tournament Selection, and the Effects of Noise.” *Complex Syst.* 9 (1995): n. pag.

Appendix A – Pre-Improvements Results

The results shown here are the results of OEP against other agents before the improvements were added. These improvements are explained in the Evaluation chapter with how they affected the results.

		Simple	RHEA	OSLA	MCTS	EMCTS
OEP	% win rate	39	39.6	50.4	63	63
	Wins	195	198	252	315	315
	avg Score	8123.82	9913.83	9752.34	10680.74	10409.94
	avg Tech	20.84	21.62	22.19	22.42	22.84
	avg Cities	1.5	2.26	2.35	2.87	2.55
	avg Pro	12.35	18.73	18.75	22.69	20.29

Figure 7. Results for OEP against other Agents

		Simple	RHEA	OSLA	MCTS	EMCTS
OEP	% win rate	61	60.4	49.6	37	37
	Wins	305	302	248	185	185
	avg Score	7794.46	11925.07	9530.66	9021.59	9125.4
	avg Tech	18.04	23.51	214.81	22.84	22.09
	avg Cities	2097	2.11	2.06	1.34	1.79
	avg Pro	21.35	17.72	16.23	10.05	15.15

Figure 8. Results for other agents against OEP

Appendix B – Mutate Individual

The following code is one that randomly mutates an individual so that every move has a chance to be mutated. This was used in the second improvement stated in the Evaluation chapter.

```
1:    private Individual mutateInd(Individual individual, GameState gs){
2:         ArrayList<Action> child = new ArrayList<>();
3:         for(int a = 0 ;a < individual.getActions().size(); a ++) {
4:             if (!(gs.getActiveTribeID() == getPlayerID())) {
5:                 Individual in = new Individual(child);
6:                 in.setGs(gs.copy());
7:                 return in;
8:             }
9:             int chance = m_rnd.nextInt((int) (params.MUTATION_RATE * 100));
10:            if ((m_rnd.nextInt(100) < chance) ) {
11:                Action ac = mutation(gs.copy());
12:                advance(gs, ac);
13:                child.add(ac);
14:            }else{
15:                   if(checkActionFeasibility(individual.getActions().get(a),
gs.copy())){
16:                    advance(gs,individual.getActions().get(a));
17:                    child.add(individual.getActions().get(a));
18:                }
19:                else{
20:                ArrayList<Action> allAvailableActions = this.allGoodActions(
gs.copy(), m_rnd);
21:                        Action ac = allAvailableActions.get(m_rnd.nextInt(
allAvailableActions.size()));
22:                    advance(gs,ac);
23:                    child.add(ac);
24:                }
25:            }
26:        }
28:        Individual in = new Individual(child);
29:        in.setGs(gs.copy());
30:        return in;
31:    }
```