Building a Better Bot:

Evolving Game AI through Genetic Programming of Decision Trees

Author

Erik Wurman

Advisor

Stephen Majercik

May 3, 2019

Contents

Abstract

TODO

1 Introduction

This Honors Project seeks to create an Artificial Intelligence Player for a time-limited Turned Based Strategy Game. The game is *BattleCode: Escape to Mars*, which was created by MIT for its 2018 AI challenge. The goals of this project are twofold: to determine if genetic programming can develop a player from a randomly generated initial population, and if not, then to determine if genetic programming is a valid mechanism to improve a hand-crafted decision tree to play this game.

Genetic programming is a nature inspired algorithm under the umbrella of genetic algorithms that are based on evolution. A genetic algorithm is an optimization algorithm that begins with a population of individuals. Each generation, the individuals of this population are evaluated and rated by their fitness. Then, the algorithm selects a certain number of these individuals to breed. This selection process is created in a way that makes it more likely to pick better fit individuals. Once the breeding pool is selected, two individuals undergo a crossover operation to create new individuals for the successive generation. These individuals then undergo mutation. The crossover and mutation operations allow the algorithm to make large and small steps around the solution space as it searches for more optimal solutions.

2 Battlecode 2018

2.1 Overview:

Battlecode 2018: Escape to Mars is a time-limited Turned Based Strategy Game created for MIT's AI Challenge where two teams compete for resources and fight for control of both Earth and Mars. The competition was held in January of 2018. In the game, there is one resource, karbonite, that is dispersed across the maps of Earth and Mars. There are 1000 rounds per game with each round consisting of 4 turns per player. Each player has 10 seconds plus an additional 50 milliseconds per round to complete all four moves. The winner of the game is the team that destroys all the other team's units or, in the case that neither team is completely eliminated, the team with the highest value of units at the end of the game. On round 750, Earth floods and everything on it is destroyed. If both teams die in the flood, then the team that was winning before the flood wins the game. If only one team is

destroyed, the surviving team wins. Otherwise, the two teams play the remaining 250 turns exclusively on Mars. A single game take approximately three minutes to complete on my laptop and about one minute on Bowdoin's Searles 224 Lab computers.

2.2 Karbonite

Karbonite is the only resource in *Battlecode 2018*. It can be mined on both Earth and Mars. In addition to mining revenue, each team receives a basic karbonite income each round. This income is inversely related to the team's current store of karbonite. A player gets 10 karbonite per round, decreased by 1 karbonite per 40 karbonite they have in stock to a minimum of 0. Both teams start the game with 100 Karbonite. [5]

2.3 Maps

The two planets, Earth and Mars, have different maps and karbonite deposits. Each game map is a 2 dimensional grid filled with locations. Each grid location is part of the map, meaning units can exist there, or they are blocked, creating obstacles for units to avoid. Each valid location has a fixed karbonite amount that can be harvested over the course of the game. Both teams start exclusively on Earth. The Earth map is always symmetric to avoid giving one team an unfair starting advantage. Mars is not symmetric since players can choose where on Mars they wish to land. Structures cannot be built on Mars.

2.4 Units and Buildings:

There are 5 types of units and 2 types of buildings in *Battlecode 2018*. There are workers, which can mine karbonite, construct buildings, and clone themselves. There are knights, which have high health but can only attack adjacent squares. There are rangers, which have low health but can attack from a distance. There are mages, which have very low health, can attack from a medium range, and have splash damage. There are also healers, which can heal nearby friendly units. The buildings are factories, which construct units, and rockets. Units can be garrisoned in a rocket, and the rocket can take off to Mars, effectively transporting units to Mars. All buildings and units cost karbonite to create. Figure 1 outlines the unit statistics for the five unit types in *Battlecode 2018*.

2.5 Actions and Cooldowns:

Each turn, a player can perform 4 actions. Possible actions for workers include moving, harvesting, replicating, and building or repairing structures. Possible actions for military units include moving and attacking or healing. All units have a movement cooldown and an

Robot	Worker	Knight	Ranger	Mage	Healer
Factory Cost	50 (*60 if replicated)	40	40	40	40
Max health	100	250	200	80	100
Damage	0	80	30	60 (*damage is also dealt to adjacent squares)	-10 (*Healers must heal() to deal "attack")
Attack range	0	2	50 (*cannot attack within range 10)	30	30
Vision range	50	50	70	30	50
Movement Cooldown	20	15	30	20	25
Attack Cooldown	100	20	20	20	10
Passive Abilities*	None :(	Defense	Far-sighted	Explosive shot	Positively Charged
Active Abilities*	Replicate	Javelin	Snipe	Blink	Overcharge
Active Ability Cooldown*	500	100	200	250	100
Active Ability Range*	2	10	Infinity	8	30

Figure 1: Unit Statistics

attack cooldown that limit how often a single unit can move and attack over a given number of turns. Factories can construct each type of unit for a karbonite price. Rockets can takeoff from Earth and land on Mars.

2.6 Common Strategies:

MIT videotaped and publicized the day of the competition. In this video, MIT's commentators outline some basic strategies that many of the contestants follow. On small maps, teams rarely make it to Mars, so the strategy seems to be to build and army and attack early in the game. These armies are a small group of military units, usually knights with some mages or rangers. An early military advantage generally leads to success on these small maps.

On larger maps, some teams take the same approach, although there is more coherency with the attack strategies. Instead of a few units, there are many. Armies are made up almost entirely of either mages or rangers. The abilities of these groups tend to stack together. For example, a large group of mages with splash damage can take out a large group of enemies while an army of rangers can surround and corner an opposing army by continually attacking the closest enemy until the rangers reach the buildings of the enemy. It seemed that a ranger army is more effective earlier in the match while the mage army is more effective in the later part due to a researchable upgrade to the mages that allows for local teleportation, called blink, that allows them to move easily around obstacles and avoid danger.

In addition to the armies, when playing on a bigger map a player spends more time developing the economic side of their team. This includes creating more factories for building units, creating more workers for getting more resources, and building rockets to get their units to mars. The player that gets to mars first tends to have an advantage since moving troops out of a rocket costs a turn each, and the opposing team can then attack the units coming out of the rockets without the full army in the rocket being able to fight back. Therefore players tend to try and move their army to mars earlier that the other team, but doing so comes at a cost to resources on Earth and that development.

3 Decision Tree Players

3.1 Decision Tree Overview:

A decision tree is a tree structure that takes an input and evaluates the input through a series of if statements to determine the course of action. Most linear programs can be represented as decision trees in one form or another. For this project, the decision trees will take the

game as input and output the desired action. However, due to the crossover operation in genetic programming that combines two trees, a grammar for these decision trees had to be carefully constructed. Otherwise, crossover between trees could construct non-functioning trees.

3.2 Player Structure:

A player can make one of four possible actions each round. These actions are move, build, attack, and harvest. Each player has a decision tree for each of these actions, which is responsible for determining what specific movement, attack, building, or harvesting should be done. For example, a rocket taking off would be under the Move category while a healer healing a nearby unit could fall under the Attack category, and a factory constructing units would be a Build action. An additional tree, the Top Tree, will determine which of the four action types the player should perform. Therefore, each player will contain five separate decision trees. In order to decide what move to make for its turn, the player will first evaluate the Top Tree and then evaluate the corresponding action tree to make its move.

3.3 Tree Structure:

Each decision tree is made up of different types of nodes. These nodes are responsible representing different information in order for the decision tree to operate and evaluate to a decision. Each node is one of the following node types: DecisionNode, IfNode, BooleanNode, InformationNode, OperandNode. Each Node inherits from the Node type, which has pointers to three children nodes that help define the tree structure. Figure 2 shows the structure of a tree with two layers of If Nodes.

3.3.1 Decision Nodes:

Decision Nodes hold a function to evaluate. In the case of the Top Tree, this function tells which action tree to evaluate. Otherwise, this function is a function that performs an action in the game. Some examples of these types of functions are *unitAttackClosestPossibleEnemy*, *factoryProduceWorker*, and *unitMoveIntoClosestRocket*.

3.3.2 Boolean Nodes:

Boolean Nodes evaluate as true or false, often by querying information about the game and returning the evaluating of a boolean expression. The Node itself contains either a binary function such as *isRanger* or *isFactory* and takes in a unit as a parameter, or it contains an operation and its children hold the information to query the game for its status. In practice, the operation is a numeric comparison such as the less than operation. However, Boolean

Nodes can also be chained such that the top node holds a binary operation and the children are both Boolean Nodes. Almost all Boolean nodes with binary functions are found on the lowest level of the tree, while those with children are found higher in the tree.

3.3.3 Operand Nodes:

Operand nodes contain integer values. These nodes have no children of their own, but they are useful as a child to a Boolean Node.

3.3.4 Information Nodes:

Informations Nodes hold functions that take no parameters and return a value about the game or a unit in the game. There are two uses for an Information Node. The first is as a child for a Boolean Node to give information about the game in numeric form. The second is as the optional fourth child of an If Node to select a unit of the evaluation of the If Node. For example, the function *getNumberOfWorkers* returns the number of workers a player has, and would be held by an Information Node that is the child of a Boolean Node. In contrast, the function *selectUnitWithLeastLifeThatCanAttack* returns the unit with the least health that can attack an enemy and would be held by an Information Node that is the fourth child of an If Node.

3.3.5 If Nodes:

If nodes are the driving force of decision tree. They hold no data of their own but provide the structure that allows for simple tree evaluation. Each If Node has three children with an optional fourth Information Node child. The first child of each If Node is a Boolean Node. If the If Node is the bottom-most If Node in the tree, then the second and third children are Decision Nodes and it has the optional fourth child Information Node. Otherwise, if the If Node is in the middle of the tree, then its second and third children are If Nodes and it has no optional Information Node. The second child is the true child of the If Node, so if the Boolean Node evaluates true the If Node would evaluate its second child. The third child is the false child and is evaluated similarly for a false Boolean Node evaluation.

3.4 Tree evaluation:

In order for a decision to be made, a tree must be evaluated. To evaluate these decision trees, the player iterates down the If Nodes of the tree until it gets to a Decision Node. To do this, the tree has each If Node evaluate their Boolean Node and set the current node as the appropriate true or false child. On the lowest level If Node, which has the optional
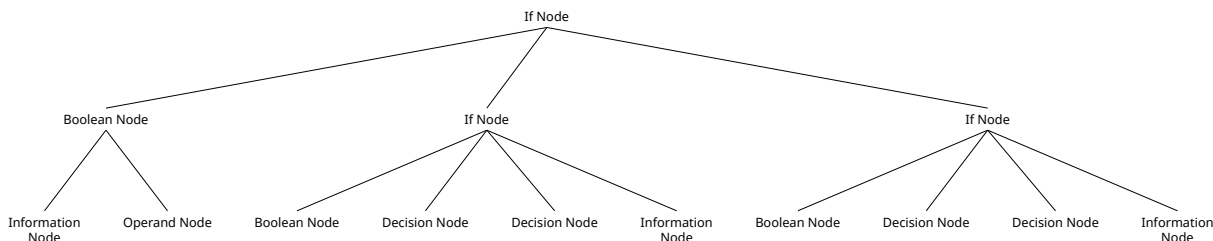
Figure 2: A Two-Layer Decision Tree Structure

Information Node, the evaluation is slightly different. The If Node first evaluates its fourth child, the Information Node, which gives a unit that it passes as a parameter to the Boolean Node. The If Node then evaluates the appropriate child Decision Node, which executes the function that interfaces with the game.

4 Genetic Programming

4.1 Overview

Genetic programming is an evolution-inspired optimization algorithm that evolves programs to work more effectively. As in this project, programs evolved by a genetic programming algorithm are often represented by trees. The typical genetic programming algorithm has three main operators: Selection, Crossover, and Mutation. The algorithm begins with a randomly generated population of individual programs. In this project, I randomly generate the decision tree players, which each have the 5 randomly generated decision trees. Each generation, I evaluate the fitness of each individual and use Selection to select the better fit trees for breeding. Breeding is conducted by selecting two individuals from the breeding pool and having them undergo Crossover to create a new individual for the following generation. Before continuing to the next generation, each of the new individuals undergo Mutation. The pseudocode for the typical genetic programming algorithm is below:

4.2 Selection:

In genetic programming, there are many methods of selecting the individuals for the breeding pool. Some methods will draw probabilistically with replacement from the population based on the fitness of each individual. Drawing probabilistically has the drawback that the fitness must be a numeric score, that a high fitness individual will be chosen often so that

---

Algorithm 1 Genetic Programming

 1: initialize random population P of size s
 2: **for** each generation **do**
 3:     **for** each ididividual $p \in P$ **do**
 4:         evaluate fitness of p
 5:     **end for**
 6:     Breeding Pool $B = \text{Selection}(P)$
 7:     Next Population $P' = \text{Crossover}(B)$
 8:     $\text{Mutate}(P')$
 9:     $P = P'$
10: **end for**

---

duplicate copies of one individual exist in the breeding pool, and that the population can converge prematurely to a suboptimal individual. Other methods rank the individuals by fitness and then draws probabilistically from the population with replacement based on the fitness rank of each individual. This again requires a numeric score for an individuals fitness, but has less bias toward a relatively dominant fitness score. Due to the variable nature of games in *Battlecode 2018*, it is difficult to construct a fitness evaluation with a numeric score based on tree structure or gameplay statistics. Instead, fitness is evaluated during Selection by using a selection method called Tournament Selection.

Tournament Selection selects two individuals from the population and has them compete. The winner of the competition is selected for the breeding pool. In this project, these two individual will go head to head playing *BattleCode 2018* against one another. Tournament Selection guarantees the worst fit individual will not be in the breeding pool because it will lose against any other individual in the population. It also is a natural selection method for a binary outcome of winner and loser in a game like *Battlecode 2018*.

4.3 Crossover:

The goal of Crossover is to create individuals that take relatively large steps around the solution space in search of better solutions. It is the genetic operator that allows for the most exploration of the solution space. Crossover works by selecting two individuals in the breeding pool and swapping subtrees between them to create new individuals. This swapping works better on smaller subtrees because it produces less radical changes. I execute crossover on two trees by beginning at the roots of each tree with a small probability of crossing over. As I traverse the tree toward the leaves, I increase the probability of stopping to

execute crossover at that node. Thus, it is more likely for crossover to occur lower in the tree.

However, navely swapping subtrees will oftentimes invalidate a tree. Therefore, in order to make it more likely that Crossover will produce legal trees, swapping subtrees only occurs on like nodes. For example, if the Crossover function decides to swap at an If Node in tree A, then it can only select an If Node in tree B to swap with. This requirement ensures that the structure of the tree does not change so that it will still be able to execute, and it helps ensure that decision nodes have the correct parameters for their functions and avoid errors in decision tree evaluations.

4.4   Mutation:

While Crossover allows the genetic programming algorithm to search the solution space, Mutation allows for local search around an individual in that solution space. In order to perform local search, Mutation must only affect small subtrees in a given decision tree. Therefore, the probability of mutation on a given node is very small and increases only slightly with the depth of the node in the tree. Mutation can only affect Boolean Nodes, Operand Nodes, Information Nodes, and Decision Nodes. In order to ensure that Mutation does not break the functionality of the tree, the genetic operator only operates on nodes that contain a function. Mutation acts upon the node by replacing that function with a similar, but not the same, function. For example, in an Information Node that holds *unitMoveTowardAlly-Behavior* may mutate to hold *unitMoveTowardEnemyBehavior*. Each function has a unique group of functions that are similar, and no function belongs to more that one group. A list of all these groups allows for easy lookup of similar functions during the mutation operation.

4.5   Genetic Programming to Battlecode Player:

Through genetic programming, I wanted to evolve a single player for *Battlecode 2018*. Therefore, I would record the best individual at each generation by having all individuals compete in a tournament. At the end of all the generations, I would have each generation's winner enter the winner's tournament to determine the best individual bot.

5 Related Works

Creating AI players for complex games is nothing new. Much of research in artificial intelligence begins with games. However, less research has been done with Real Time Strategy (RTS) or time-limited Turn-Based Strategy games. In 2005, JinHyuk Hong and Sung-Bae Cho created a simple RTS game and developed a reactive model to make decisions with limited information. They used co-evolution and genetic algorithms to develop these reactive models. To evaluate the fitness of their agents, Hong and Cho evaluated three separate measures of a game; winning, economic development, and opponent units killed. This let them rank each individual model in their populations. They had success with their model on simple maps, but it struggled on maps with obstacles. [1]

More recently, a group of researchers compared genetic programming to a genetic algorithm for playing Planet Wars, a simple RTS game about conquering opponents planets using ships. Each conquered planet produces ships for the team. They evaluated fitness by winning and time required to win. The results showed that the bots created through genetic programming was superior to the genetic algorithm bot. The genetic programming bot also performed well against opponents it wasn't trained against. This work was instrumental in my decision to use genetic programming [2]. This Planet Wars game is a much simpler game than *Battlecode 2018*, but it is still useful for gauging strategies to creating a an AI bot for playing a game.

Kahn and Okada researched how genetic programming could be used to create separate families of individuals where individuals within a family share characteristics. They did this for generating NPC faces in games with different clans of a given race. Their approach was successful at randomly generating alike faces while saving memory [3]. This idea of separate families of individuals in the population is an interesting one, as we could have families of players that focus on different strategies. Defining what those strategies or families are though, is difficult.

In 2016, a group of researchers created a method they dubbed 'Online Evolution' for the multi-action turned-based strategy game *Hero Academy*. Each turn, they create a population of move combinations and use a genetic algorithm to find the best series of moves. They evaluated individual sets of moves by evaluating the board as if the moves were played. Their results showed this method outperformed Monte Carlo Search and two separate greedy algorithms [4]. This is an interesting work because *Hero Academy* has a similar strategy as *Battlecode 2018*. However, because *Battlecode 2018* has a much shorter time limit, the strategy of running a genetic algorithm each turn is not feasible.

6 Experimental Challenges:

For genetic programming to work effectively, it needs a large population and many generations. Because each game of *Battlecode 2018* takes about one minute to complete, completing the genetic programming algorithm with a population of 32 and 50 generations takes around a day. This makes extensive testing of parameters take a very long time. Additionally, the game was unable to run on Bowdoin's High Performance Cluster despite the administrator of the cluster's best efforts in finding and installing the required dependencies. Therefore, testing was limited to a couple Searles 224 Lab computers, which makes extensive testing rather time consuming.

7 Preliminary Experiments

7.1 Preliminary Setup:

The initial goal of this project was to evolve a player that could compete against a real player. In order to evolve though, the genetic program would have its population play itself and select using Tournament Selection. I downloaded some of the entrants' players from the 2018 tournament to be able to test against should the evolved bots look successful. These players also served as building blocks for the functions I wrote for the decision trees to interface with the game.

However, in order to create great players, the algorithm must first create better than random players, then medium players, then good players. Therefore, I originally planned to run the algorithm for a set number of generations starting with a randomly generated population of players. Then, I would repeat this process but add some of the players created by the first process to the initial population to give the algorithm a head start in finding decent players. This process would repeat, but at the end of each run of the genetic programming algorithm, the players created would be tested against the next benchmark player. That is, the first players created would be tested against a random player. Once there are players that consistently beat a random player, the players would be tested against the medium player. Medium players can be created by modifying *Battlecode 2018* applicants' code and changing weights and removing nuances in the code. Once the players beat the medium players they would challenge applicant's unmodified bots.

The first tests were run with the following parameters:

50 generations,

32 individuals in the population,
Crossover Occurrence probabilities of 0.4, 0.6, or 0.8,
Mutation Occurrence probabilities of 0.2, 0.4, or 0.6,
Crossover stop traversing down the tree probability of 0.1
Mutation probability of mutating this node probability of 0.01

Nine tests were run in total to explore the basic genetic programming parameters effects on the process.

7.2 Preliminary results:

The initial results for this process were mixed. Regardless of the parameters given to the process, each winner was severely limited in its behavior. The best players evolved, that is the best of the winners of each process, acted by replicating the workers and building factories. The factories would occasionally create military units, but these units would not move to attack positioning or attack. The worst players evolved only moved around. A quick glance at the evolved winners' top trees confirmed these behaviors. The top tree for each winner had only one if statement and two decision nodes, so at most the player would only ever do two of the four possible actions. This was clearly not the desired behavior, so I started collecting data about the process to understand why our trees shrank from the original population's top trees with three or four if layers.

This led me to measure the average size of the trees as a function of the generation length. The results for all nine preliminary results can be seen in Figures 3−11.
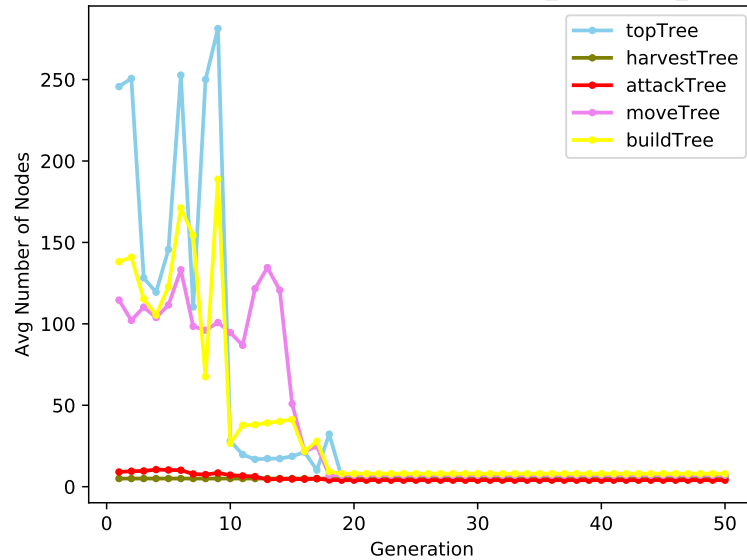
Figure 3:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.8 and
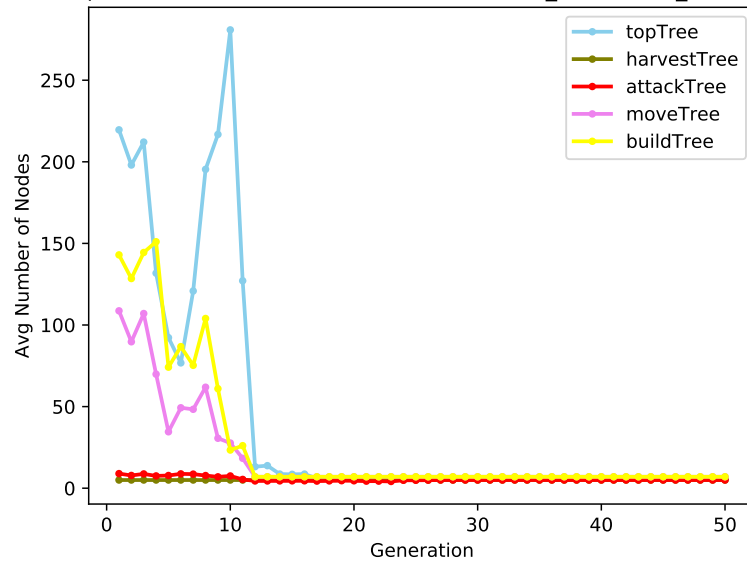a Mutation Occurrence Probability of 0.6

Figure 4:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.8 and
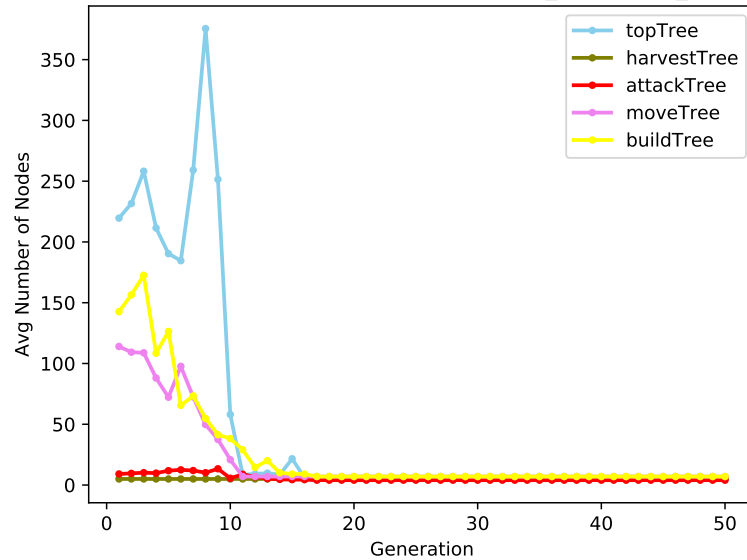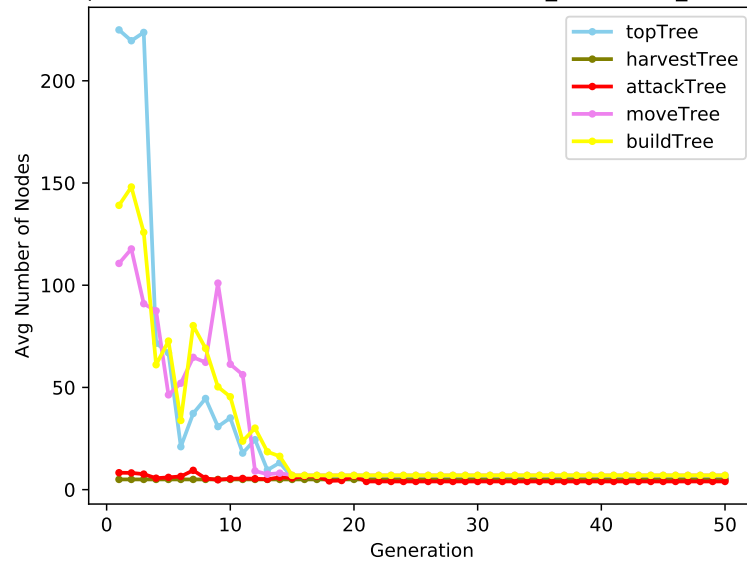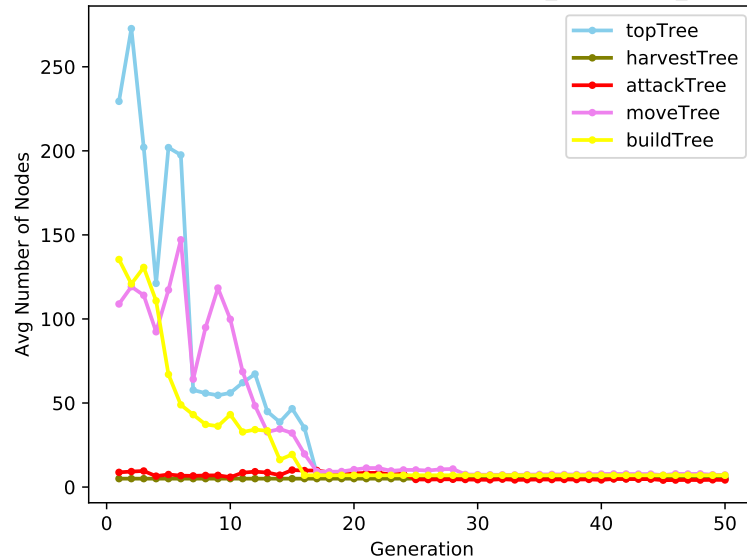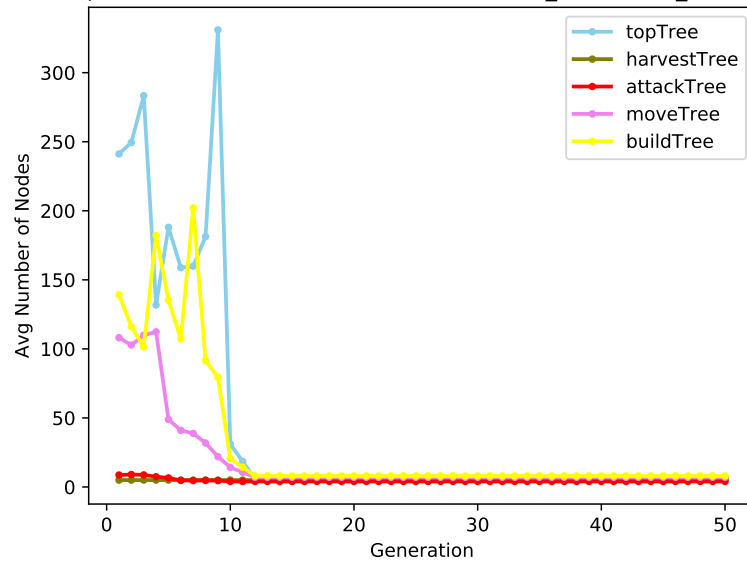a Mutation Occurrence Probability of 0.4

Figure 5:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.8 and a Mutation Occurrence Probability of 0.2

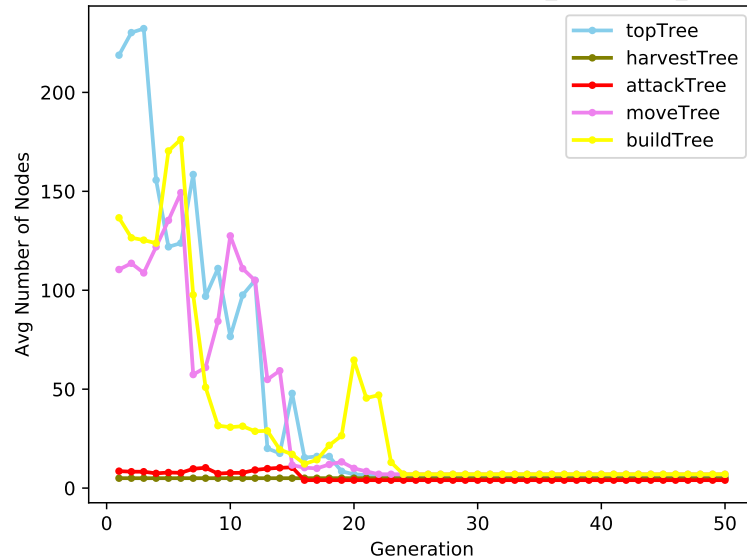

Figure 6:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.6 and a Mutation Occurrence Probability of 0.6

Figure 7:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.6 and a Mutation Occurrence Probability of 0.4

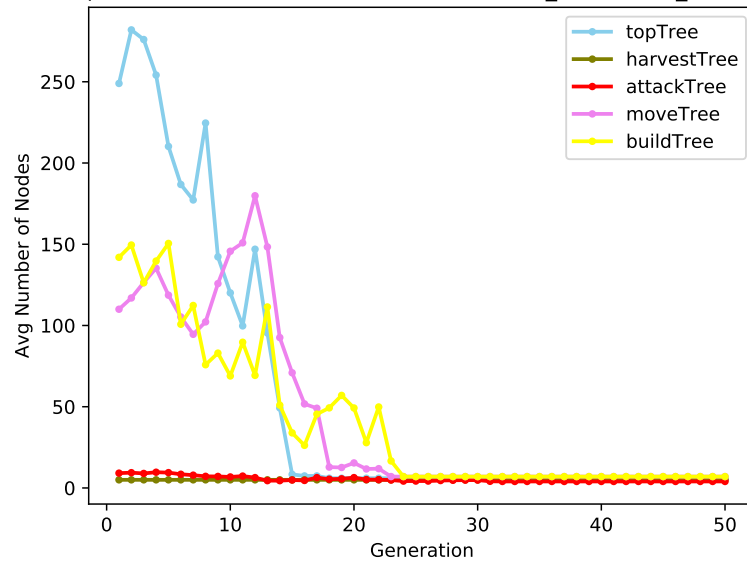

Figure 8:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.6 and a Mutation Occurrence Probability of 0.2

Figure 9:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.4 and
a Mutation Occurrence Probability of 0.6

Figure 10:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.4 and
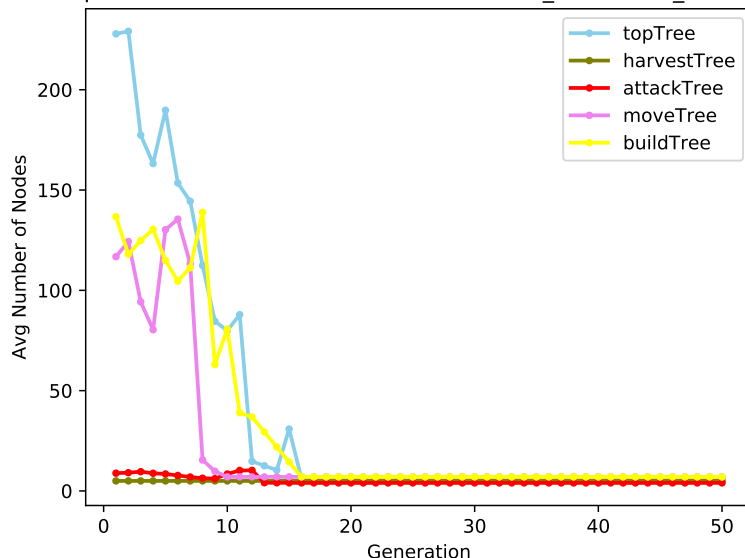a Mutation Occurrence Probability of 0.4

Figure 11:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.4 and
a Mutation Occurrence Probability of 0.2

7.3 Preliminary analysis

It was clear from these figures that changes to my genetic operators must occur for this project to produce any reasonable behaviors. Coincidentally though, there were two rather successful players from this bunch of tests. The first player's top tree only had build outcomes, so the player would build constantly, getting the necessary resources from the automatic income. This resulted in a very large economic score at the end of each game. Since the bot was not playing against any bots with functional militaries, the player just kept growing. Because the tiebreaker for winning if both players die in the flood is the economic score, this player turned out to be successful without exhibiting the desired intelligent behavior, since units couldn't move or attack, just spawn outwards like an amoeba. The other successful player was similar, only had one build decision, with the other decision as harvest. However, the If Node's Boolean Node child examined if the current round was less than round 3. If it was past round 3, the player would build. Thus, these two behaviors exhibited the same success when playing other bad players.

These players could be beaten, however, by a player that successfully gets to Mars, or a player that successfully attacks. I decided that as I fixed the genetic operators to avoid losing tree depth I would focus on constructing the genetic programming structure to encourage

getting to Mars. Because testing takes so long, I had to pick only one focus of developing an army or getting to Mars.

8 Fixed Size Tree Experiments

8.1 Fixed Size Trees

In order to avoid converging into small trees, I decided to keep each tree in a player a constant size. While this restriction made it so that the genetic programming process may not find the optimal tree configuration, if it exists, the restricted size does greatly shrink the solution space that the process must search. This makes it easier for the process to develop semi-intelligent behavior. Because the tree structure is fixed, I could impose stricter limits on the types of nodes that show up. For example, Boolean nodes only have the less than operation and are not chained while Information Nodes are only found as children to If Nodes on the lowest layer and Boolean Nodes. Previously, If Nodes were able to have a Decision Node child if they were part of an asymmetric If Node when one child was a decision and the other another If Node. This was because the Decision Node whose parent was the Information Node needed to evaluate with a unit, provided by the Information Node.

For all the following tests, the size of each tree for any individual can be found in Table 1.

Top Tree	4 If Layers
Harvest Tree	1 If Layer
Attack Tree	2 If Layers
Move Tree	4 If Layers
Build Tree	4 If Layers

Table 1: Player Fixed Tree Sizes

Each tree is balanced, and I didn't want to create too much available expression because all I wanted to produce was a player that arrived on Mars. However, changes to the tree structure limits necessitated changes to the genetic operators of Crossover and Mutation.By examining the raw data on mutations and crossovers from the preliminary tests, I was also able to gauge how often mutation occurred and where in the tree crossover occurred. These numbers helped shape the changes I made to the fixed size genetic operators.

8.2   Fixed Size Genetic Operators

8.2.1   Fixed Size Crossover

Before we fixed the size of a tree, the crossover operator would take two trees and find like nodes in each tree and crossover at these two unrelated instances. Now, in order to keep the tree heights the same before and after Crossover, the operator must do Crossover at the same location in both trees. By swapping nodes at the same location, the crossover operation can introduce new paths down a tree without changing the overall tree structure. Over generations, this maintains the depth of the tree so that a later generation individuals can still be complex enough to have multiple actions for an action type. That is, when completing a Build action, the player should be able to differentiate when to build a unit, factory, or a rocket rather that always building one.

8.2.2   Fixed Size Mutations

One of the challenges of the mutation operator is to be able to make small but meaningful changes to a tree. The operator is totally ineffective if the changes it makes are minuscule. For example, if the mutation process is trying to mutate a Boolean Node that compares the round number with an Information Node holding the *getRoundNumber* function to number 100 in an Operand Node, and all mutation does is alter the number 100 to 101, then there is no effective change in the tree's behavior. The preliminary Mutation function picked numbers randomly, which, in the case of round number, gave 1000 possible values that the tree had to explore. Now if we consider the scales of all other information functions like how much karbonite a player has (0-400 at least) and others, the solution space of all these factors grows tremendously. Therefore, I decided to add more structure to the mutation operator for these Fixed Size Trees.

In order to add a solution-space-limiting structure to the mutation operation, I mapped each possible function that can be held in a Boolean Node's Information Node child to a list of legal values that could be held by the Boolean Node's other child, an Operand Node. All mutations to a tree must keep any Boolean Node comparisons matching these legal values. The mapping can be found in Table 2.

Additionally, the Mutation function itself changed to only operate on Boolean Nodes, Decision Nodes, and Information Nodes on the lowest Layer. I also changed it to only perform once on a tree, given that it occurs according to the mutation occurrence probability. To do this, this new Mutation function traverses the whole tree and stores references to all eligible

getRoundNumber	50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 600, 700, 750
getNumberOfWorkers	1, 2, 4, 7, 11
getNumberOfFactories	1, 2, 3
getNumberOfKnights	1, 2, 3, 5
getNumberOfRangers	1, 3, 5, 8
getNumberOfMages	1, 2, 3
getNumberOfHealers	1, 2, 4, 7, 11
getNumberOfRockets	1, 2
getKarbonite	50, 100, 150, 200, 250, 300, 350, 400
getNumberOfAttackers	1, 4, 8, 12
getMaxNumberOfUnitsInEarthRocket	1, 3, 5

Table 2: Game Information Functions' Legal Values

nodes in the tree that it may mutate. It then picks one of these nodes randomly. Note that this has a built-in bias toward mutating nodes lower in the tree since there are twice as many nodes in the next layer as the current layer at any point in the tree. If a Boolean Node is chosen from an upper layer, Mutate picks a new game information function and associated legal value randomly. If it is a bottom layer Boolean Node and carries a function such as *isKnight* or *isWorker*, then it will replace them with like functions. If the node chosen is a Decision Node or an Information Node, then they live on the lowest layer and their functions are replaced by random functions in their unique similarity group. In the case that a Decision Node is chosen while mutating the Top Tree, the type of action is changed to a random type.

8.3 Generating Initial Fixed Size Population

In order to generate the initial population for genetic programming, I first must be able to generate a random player. This is done by creating the appropriate number of If Nodes for all layers for each Tree. The number of layers for each tree can be found in Table 1. We build each tree from the bottom up, so half the If nodes are used to create the bottom layer. The Boolean Nodes for these are made from the *isUnitType* function group and each If Node has an Information Node as a fourth child that selects a unit such as *selectRandomAttacker* or *selectRandomBuilder*. The next layer up takes two If Nodes from the lower layer and has a Boolean Node that has a random game information function in the Information Node child and a corresponding legal value for that function in the Operand Node child. This process continues until all layers have been created. However, during the iterative building process, I make sure not to repeat game information functions that are lower in the tree. This helps diversify the population by allowing more of the available functions to be represented by

some individuals in the starting population.

8.4 Curriculum Training Overview

Due to this project's time constraints, the experiments I was able to run with fixed size trees were limited. I therefore prioritized demonstrating whether genetic programming could be used to improve upon an existing player, rather than the more challenging task of evolving AI from scratch. To attempt to demonstrate improvement, I used a technique called Curriculum Training common in AI training. Curriculum Training works by learning steps of a process one at a time rather than all at once. It mirrors how a child might learn from the school curriculum. In this project, I used curriculum training to evolve one tree of a player at a time.

The first step in beginning curriculum training is to have a curriculum to train on. I therefore wrote a player in this decision tree structure that builds a rocket and launches it if the rocket fills with units. However, the player could not fill the rockets with units before Earth floods, and subsequently dies in the flood. This player is the base of my curriculum training, so if the genetic programming could evolve a player that can fill the rocket, or decide to launch when partially filled and fill to the threshold, I would have clearly demonstrated the improvement through genetic programming that I sought.

Curriculum training began with evolving the Top Tree. Every player in the population had the same lower trees that come from my hand-crafted player, and the genetic programming only acted on the Top Tree each generation. After all generations have completed, the training begins again on the next tree. However, each successive round, we use the evolved tree's from the winners of the previous generations. So in the second round of curriculum training, each individual in the population has three trees that come from the hand-crafted player, a Top Tree that was evolved and comes from the previous round's winner, and a tree they are currently evolving. The progression of tree evolution in my curriculum training can be found in Table 3.

8.5 Curriculum Training Results

9 Future Work

1. fixed size tree from scratch

Curriculum Round	Top Tree	Move Tree	Build Tree	Attack Tree	Harvest Tree
1	*evolving*	Hand-Crafted	Hand-Crafted	Hand-Crafted	Hand-Crafted
2	**evolved**	*evolving*	Hand-Crafted	Hand-Crafted	Hand-Crafted
3	**evolved**	**evolved**	*evolving*	Hand-Crafted	Hand-Crafted
4	**evolved**	**evolved**	**evolved**	*evolving*	Hand-Crafted
5	**evolved**	**evolved**	**evolved**	**evolved**	*evolving*

Table 3: Curriculum Training Tree Progression

2. Many of the competitors in the original 2018 competition organized their decision making by phase of the game.

3. Focus on developing military rather than Mars

4. Take a built player and evolve the phases

List of Figures

List of Tables

References

[1] JinHyuk Hong and Sung-Bae Cho. *Evolving Reactive NPCs for the Real-Time Simulation Game.* IEEE Symposium on Computational Intelligence and Games, 2005.

[2] A. Fernndez-Ares, P. Garcia-Sanchez, A.M. Mora, P.A. Castillo, and J.J. Merelo. *Designing Competitive Bots for a Real Time Strategy Game using Genetic Programming.* Dept. of Computer Architecture and Computer Technology, CITIC-UGR, University of Granada, Spain, 2014.

[3] Umair Azfar Khan and Yoshihiro Okada. *Genetic Algorithm (GA)-Based NPC Making.* In: Lee N. (eds) Encyclopedia of Computer Graphics and Games. Springer, Cham. 2015.

[4] Niels Justesen, Tobias Mahlmann, and Julian Togelius. *Online Evolution for Multi-action Adversarial Games.* In: Squillero G., Burelli P. (eds) Applications of Evolutionary Computation. EvoApplications 2016. Lecture Notes in Computer Science, vol 9597. Springer, Cham

[5] (2019, May 3) *Battlecode 2018 Gameplay Specifications v1.2.0* [Online]. Available: https://s3.amazonaws.com/battlecode-2018/specs/battlecode-specs-2018.html