Bowdoin College Computer Science

# Building a Better Bot:

## Evolving Game AI through Genetic Programming of Decision Trees

*Author*
Erik Wurman

*Advisor*
Stephen Majercik

May 15, 2019

# Acknowledgements

I would like to thank Professor Stephen Majercik for all the guidance he gave me during this project. His advice and knowledge were a great help to this project and I could not have done this without him. I would also like to thank my reader, Professor Sean Barker, for providing exceptional and timely feedback and advice despite having his plate full with a newborn son. I would like to give another thank you to Bowdoin's IT staff including, but not limited to, DJ Merrill, Samuel Tarr, and Jason Pelletier for their help in setting up the infrastructure to run my experiments. Sue O'Dell and the rest of the library staff deserve gratidute as well for their help in the research and publishing processes. Lastly, I would like to thank MIT and the *BattleCode* staff for answering my many questions and providing such an interesting game with such great documentation.

# Contents

# Abstract

This project explores the effectiveness of using genetic programming to evolve decision trees to play MIT's *Battlecode 2018* AI Challenge. *Battlecode 2018* is a time-limited turn based strategy game in which two AI bots compete to conquer the world. Genetic Programming is an evolution-inspired optimization algorithm that works by breeding higher fitness individuals in a population over many generations to create more fit individuals over successive generations. This project created a decision tree grammar to create players for *Battlecode 2018* and showed that while genetic programming may not be the best method for creating players in this grammar, it can be used to successfully improve upon basic players.


# 1 Introduction

## 1.1 Outline

This Honors Project sought 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 were 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. The rest of this section introduces genetic programming and *BattleCode*. I then explain my player structure and how genetic programming acts on these players. I then talk about related works and how they relate to this project. Next, I explain the experiments I ran and the results. I finish with some future extensions to this work.

## 1.2 Genetic Programming

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. Each generation, the individuals of this population are evaluated and rated by their fitness. The algorithm then uses the selection operator to select a group of individuals from the population to breed. Once the breeding pool is selected, crossover occurs between two random individuals in the breeding pool and creates new individuals for the successive generation until the new population is filled. To

add some random diversity, 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. The pseudocode for the typical genetic programming algorithm is in Algorithm 1.

**Algorithm 1** Genetic Programming

```
initialize random population P of size s
for each generation do
    for each idividual p ∈ P do
        evaluate fitness of p
    end for
    Breeding Pool B = Selection(P)
    Next Population P' = Crossover(B)
    Mutate(P')
    P = P'
end for
```

### 1.2.1 Selection

In genetic programming, there are many methods of selecting the individuals for the breeding pool. Some methods 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 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. Tournament selection is another common method that compares two random individuals from the population and adds the more fit individual to the breeding pool. The advantage of tournament selection is its simplicity and that the fitness evaluation need only be relative between the two individuals. This project used tournament selection.

### 1.2.2 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. Figure 1 shows two trees crossing over between subtree B and E.

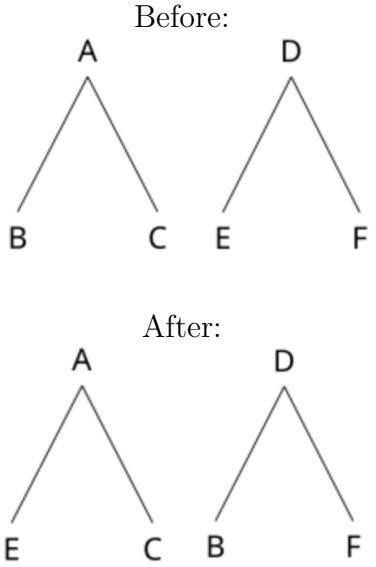


Figure 1: Crossover of two trees

### 1.2.3 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. This is typically done by randomly altering nodes in a tree at some small probability per node. Figure 2 shows a tree undergoing mutation on node B.

## 1.3 Battlecode 2018

### 1.3.1 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

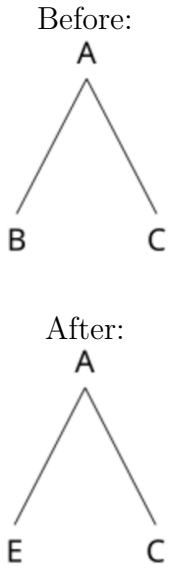


Figure 2: Mutation of a tree

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.

### 1.3.2 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]

### 1.3.3 Maps

The two planets, Earth and Mars, have different maps and karbonite deposits. Each game map is a 2 dimensional grid filled with locations. Not all grid spaces are valid locations, so units must be able to avoid invalid spaces when traversing the map. Each valid location has a fixed karbonite amount that can be harvested over the course of the game. Both teams start each game 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.

| 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 |

Table 1: Unit Statistics [5]

### 1.3.4 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. Table 1 outlines the unit statistics for the five unit types in *Battlecode 2018*.

### 1.3.5 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 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.

### 1.3.6 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, but a successful attack strategy involves more than just getting an early advantage. 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.

# 2 Decision Tree Players

## 2.1 Decision Tree Overview

A decision tree is a tree structure that takes an input and evaluates the input through a series of if nodes 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, I carefully constructed the grammar for these trees to simplify crossover. Without a strict grammar, crossover between trees could construct non-functioning trees.

## 2.2 Player Structure

I decided to sort each possible actions into one of four different action categories. These actions categories are move, build, attack, and harvest. A player can make one of these four possible actions each round. Each player has a decision tree, the Top Tree, that determines which of the action types a player will execute. Each of the action types has a decision tree that 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. 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.

## 2.3 Tree Structure

Each decision tree is made up of different types of nodes. These nodes are responsible for gathering information and representing different operations 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, ValueNode. Each Node inherits from the Node type, which has pointers to three children nodes that help define the tree structure.

### 2.3.1 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*.

### 2.3.2 Boolean Nodes

Boolean Nodes evaluate as true or false, often by querying information about the game and returning the evaluation 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. This node could also contain an operation and its children hold the information to query the game for its status. For example, the node could contain the less than operation, have a first child that is an information node containing the function *getRoundNumber* and a second child that is a Value Node containing 200. This node would evaluate as true when the round is less than 200. In practice, the operation of a Boolean Node is a numeric comparison such as the less than operation. 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.

### 2.3.3 Value Nodes

Value nodes contain integer values. These nodes have no children of their own, but they are useful as a child to a Boolean Node. For example, a Value Node can store the value 250 and could be compared to functions like *getRoundNumber* or *getKarbonite*.

### 2.3.4 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.

### 2.3.5 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

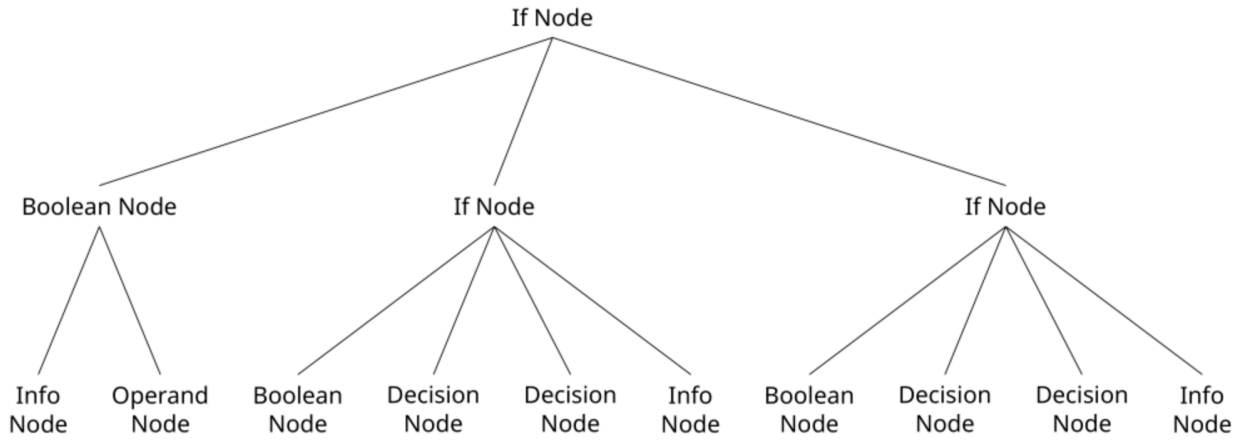


Figure 3: A two-layer decision tree structure

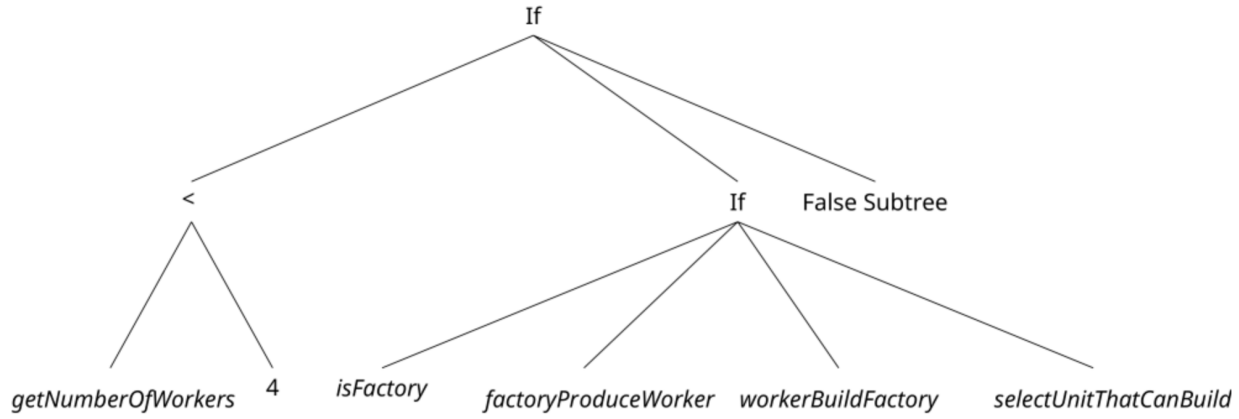


Figure 4: An example tree

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.

### 2.3.6 Example Tree

Figure 3 shows the structure of a tree with two layers of If Nodes and how each node type fits into the tree structure. Figure 4 shows an example tree that follows this structure. For space, the false subtree is not expanded in the example tree.

### 2.4 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 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.

## 3 Genetic Programming for Battlecode

### 3.1 Overview

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 tournament selection to select the better fit trees for breeding. I then have the breeding pool undergo crossover to generate the next population. This next population undergoes mutation before starting the next generation. 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.

### 3.2 Battlecode Selection

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 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 individuals 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*.

### 3.3 Battlecode Crossover

For each individual, I execute crossover on each tree separately. Each tree for a player has the same probability of doing crossover with the corresponding tree in the other player, and crossover only occurs for that tree if a random value is in the probability threshold. 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, naively 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. The node at which to swap in tree B is found the same way as in tree A, but only stops at like nodes. This search defaults to one of the like nodes in the lowest layer if the process does not stop earlier in the tree. 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.

### 3.4 Battlecode Mutation

Since mutation should only affect small subtrees in a given decision tree, 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, Value 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 *unitMoveTowardAllyBehavior* 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 Related Works

Creating AI players for complex games is nothing new. Much of research in artificial intelligence began 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 [1] 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.

More recently, a group of researchers [2] 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. This Planet Wars game is a much simpler game than *Battlecode 2018*, but it is still useful for gauging strategies to creating an AI bot for playing a game.

Kahn and Okada [3] 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. 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.

# 5 Experiments

## 5.1 Experimental Challenges

*Battlecode 2018* 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 with a 4.2 GHz Intel Core i7 processor and 16 GB 2400 MHz DDR4 memory. On these machines, a single game of *Battlecode 2018* takes about one minute to complete. For genetic programming to work effectively, it needs a large population and many generations. Completing the entire 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.

## 5.2 Preliminary Experiments

### 5.2.1 Preliminary Setup

The initial goal of this project was to evolve a player that could compete against a competitor's 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 [6] from the 2018 tournament 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
Probability of mutating a given node of 0.01

In total, I ran nine tests, one for each combination of the above parameters, to explore the basic genetic programming parameters effects on the process. I generated the initial random population by randomly generating each layer of the tree. For any bottom layer, I would create an Information Node that holds a *selectUnit* function that depends on the tree. For example, a random harvest tree would have one of the functions in the *selectWorker* grouping, while an attack tree would have a function in the *selectAttacker* grouping and the build tree would have a function in the *selectBuilder* grouping. The Boolean Node for this layer would hold a boolean function such as *isKnight* for the attack tree and *isFactory* for the build tree. Decision Nodes would depend on the boolean function, so the true Decision Node for the *isFactory* function would be a *factoryProduce* function. The false function, since if the build unit is not a factory so it must be a worker, would be a *workerBuild* function. For layers higher in the tree, Boolean Nodes contain the less than operator and have children that are Information Nodes containing one of the *getNumericGameInfo* functions and a Value node with an integer value.

### 5.2.2 Preliminary Results

The initial results for this process were mixed. Regardless of the parameters given to the process, each winner had severely limited behavior. The best players, 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 positions or attack. The worst players evolved only moved units around. A quick glance at the evolved winners' top trees confirmed these behaviors. The top tree for each winner had only one If Node 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 Node 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 were similar, and Figures 5−7 show the general trend of all tests.

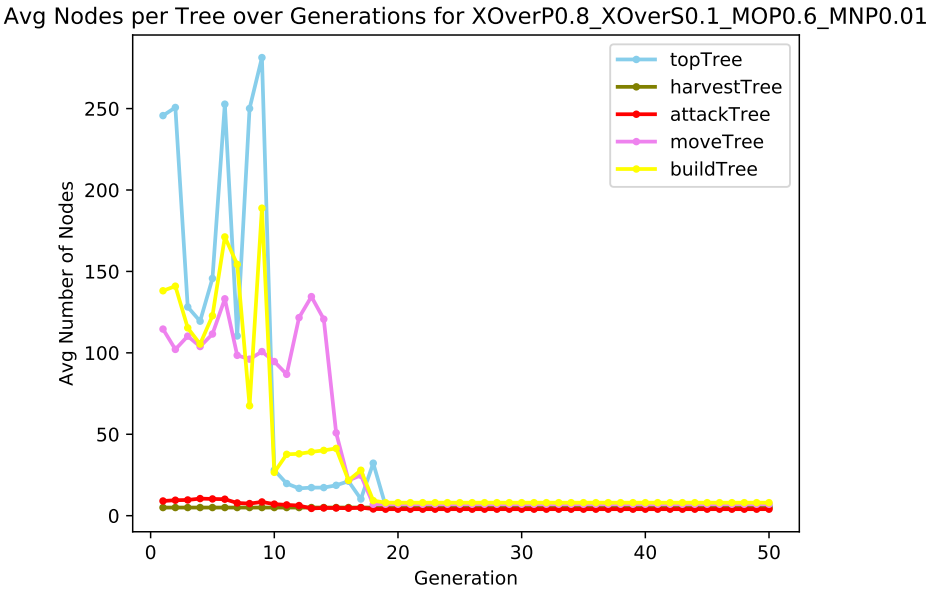


Figure 5:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.8 and a Mutation Occurrence Probability of 0.6

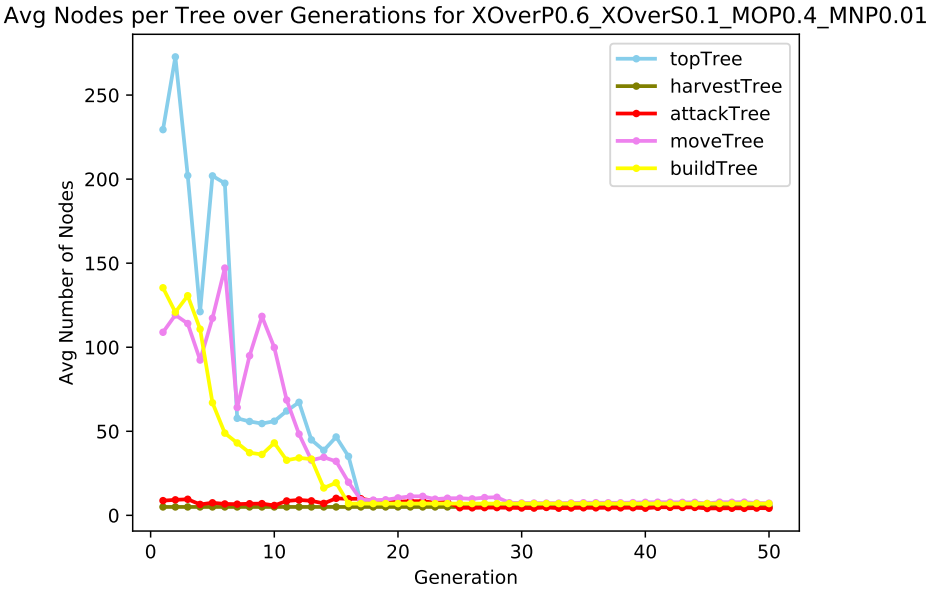


Figure 6:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.6 and a Mutation Occurrence Probability of 0.4

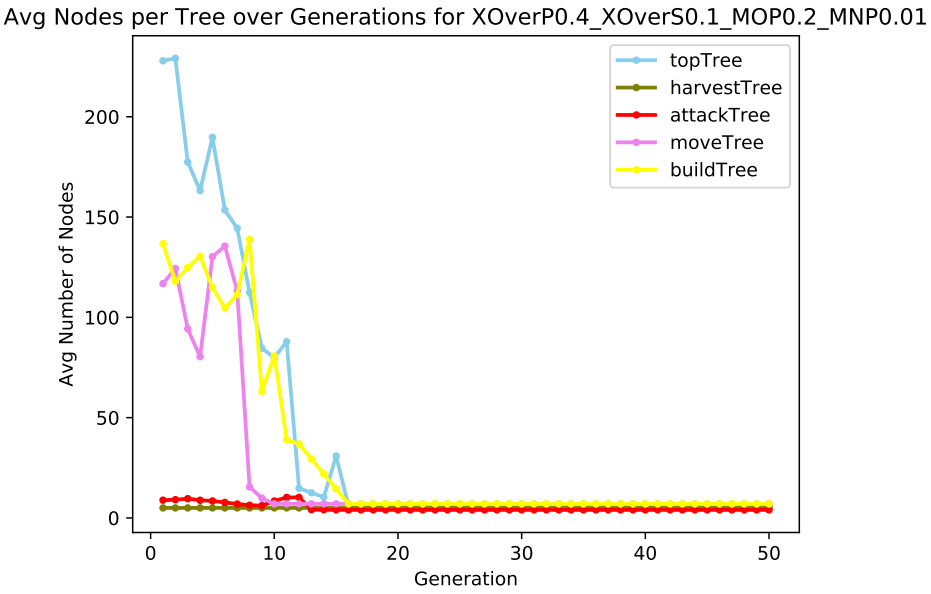


Figure 7:
Average Nodes per Tree in the Population for Crossover Occurrence Probability of 0.4 and a Mutation Occurrence Probability of 0.2

### 5.2.3 Preliminary Analysis

It was clear from these figures that I had to change my genetic operators 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 en-

courage getting to Mars. Because testing takes so long, I had to pick only one focus of developing an army or getting to Mars. By playing my bots only against other bots focussed on getting to Mars, I hoped to shrink the problem to a reasonable size so that I could evolve reasonably successful behavior.

## 5.3 Fixed Size Tree Experiments

### 5.3.1 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, Information Nodes are only found as children to If Nodes on the lowest layer and Boolean Nodes while If Nodes are now always symmetric. 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 Node 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.

Each tree is balanced, and I didn't want to create too much expressivity because all I wanted to produce was a player that arrived on Mars. However, changes to the tree structure 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.

### 5.3.2 Fixed Size Genetic Operators

Before I 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.

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 Value 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. If one considers 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 Value Node. All mutations to a tree must keep any Boolean Node comparisons matching these legal values. The mapping is shown 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 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.

| Function | Values |
|---|---|
| *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

### 5.3.3 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 4. 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 Value 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.

### 5.3.4 Curriculum Training Overview

Due to this project's time constraints, I could only run limited experiments with fixed size trees. Rather than attempting the more challenging task of evolving AI from scratch, I prioritized the more likely to succeed task of demonstrating whether genetic programming could be used to improve upon an existing player. 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 is shown in Table 3.

| 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

For the curriculum training tests, the size of each tree type is shown in Table 4. I chose these sizes to match the sizes of the trees in my hand-crafted player. I felt that these numbers were large enough to provide ample action diversity in the tree but not too large that the solution space is too big to evolve successful players.

| | |
|---|---|
| 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 4: Player Fixed Tree Sizes

## 5.4 Elitism

In addition to the regular genetic programming I was attempting with my curriculum training tests, I though it worthwhile to try the same tests while using elitism. Elitism is a common aspect of genetic algorithms that helps the best individuals of a population survive through to the next generation without losing the qualities that make it good through Crossover or Mutation. During the selection phase, a genetic algorithm using elitism will add the best of the current population into the next population without them having to go through crossover. These individuals can still be selected for breeding, but a copy of them is guarenteed to make it to the next generation. In order to implement elitism in my genetic programming, I first needed to create a function that evaluated trees for desirable characteristics so that I could establish which are the best.

I only implemented this function for the Top Tree, since this was the most important tree for a player and the easiest to quantify as desirable or undesirable. A desirable tree had qualities that made it easier for a player to get to mars. That means it would build, move, attack, harvest, check for rockets, and avoid duplicate boolean function checks down the same path. A perfect tree score was 2 to the power of the number of If Layers. I decided that building and moving are the two most important actions, so a Top Tree that lacked these action types lost 4 points from their fitness score for each action type missing. Attacking was the next most important action, so players would lose 2 points for missing the attack action type in a Decision Node. Missing the harvest action type was only a 1 point penalty since players get a karbonite income each round. A player would lose 1 point per path with a duplicate boolean function check. Lastly, players would lose 4 more points if they never check the number of Rockets they have. This penalty was meant to encourage the players to think about rockets and get to Mars.

### 5.5 Curriculum Training Results

The curriculum training is complete for Populations of 32 and 50 Generations with and without Elitism. I am still waiting on my tests with populations of 64 and 100 generations. I have not graphed the results of the first tests yet. The graphs will track the improvement made by the genetic programming by playing against a random player and counting how often it wins at different generations and graphing the win percentage over time. It will then do the same thing against the hand-crafted player, which will hopefully show improvement.

## 6 Future Work

Due to the time required to run tests, I was unable to test all the processes I planned. The next step of this project, after showing that genetic programming can improve upon existing players, is to try and develop a Fixed Size Tree player from scratch. If such a player could be evolved, then this method would show good promise for the future development of Game AI. I don't expect such a player to be anything more than mediocre, but even achieving mediocrity would be a huge step forward for emergent intelligent behavior.

Secondly, there are many avenues that this exploration into genetic programming could have pursued. The first decision I made was to separate a player into the four action types of harvest, move, attack, and build. This helped logically separate the trees and made it easy to understand how the player worked. However, most competitors in the original 2018 competition organized their decision making into phases. That is, if certain requirements were met in the game, the player would enter a new phase and act accordingly. I would be interested to see if organizing the player to act by phase provided better results than by action type.

Third, I decided to focus my second batch of tests on getting to Mars. Instead I could have focused on military dominance. This could be done by changing the hand-crafted player to focus on attacking rather than building rockets.

Lastly, it is likely to be the case that genetic programming and genetic algorithms are better at parameter optimization rather than this task. Therefore, I would like to see how well it could optimize phases in this game. This future project would take a decent hand-crafted player that separates its behaviors into phases. The player would not have to be in decision tree form either. Then a genetic programming algorithm would evolve the phase definitions to improve the performance of the player.

# 7 Conclusion

*Battlecode 2018* lives up to its reputation as a true AI challenge, as I was unable to create an effective player. However, I was able to partake in this educational, and I dare say fun, exploration into genetic programming and decision trees. I have experimented with different iterations of the crossover and mutation operators, and have come up with methods to avoid some of the pitfalls of my early attempts. After poor results from my first experiments, I had to impose a rigid tree structure to maintain the size of the trees for subsequent generations as my genetic programming algorithm proceeded. I was exposed to curriculum training and its strengths. However, due to the excessive time required to test, I was never able to fully explore the parameter space for any of my Fixed Size experiments.

The results of my tests are not promising for the future of genetic programming in game AI. Perhaps with more time, a larger population, and more generations, the results could provide better results. Despite the poor results, I still believe there is a use for genetic programming in game AI. I believe it could be useful in tuning parameters for hand-crafted behaviors.

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

[6] Salisk. (2018). A Python bot for the MIT BattleCode competition, MIT, Cambridge, MA. [Online]. Available: https://github.com/salisk/Battlecode_bot