View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Duquesne University: Digital Commons

Duquesne University
Duquesne Scholarship Collection

Electronic Theses and Dissertations

Spring 2016

A Genetic Programming Approach to Solving Optimization Problems on Agent-Based Models

Anthony Garuccio

Follow this and additional works at: https://dsc.duq.edu/etd

Recommended Citation

Garuccio, A. (2016). A Genetic Programming Approach to Solving Optimization Problems on Agent-Based Models (Master's thesis, Duquesne University). Retrieved from https://dsc.duq.edu/etd/569

This Immediate Access is brought to you for free and open access by Duquesne Scholarship Collection. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Duquesne Scholarship Collection. For more information, please contact phillipsg@duq.edu.

A GENETIC PROGRAMMING APPROACH TO SOLVING OPTIMIZATION PROBLEMS ON AGENT-BASED MODELS

A Thesis

Submitted to the McAnulty College and Graduate School of Liberal Arts

Duquesne University

In partial fulfillment of the requirements for

the degree of Masters of Science in Computational Mathematics

By

Anthony Garuccio

May 2016

Copyright by

Anthony Garuccio

2016

A GENETIC PROGRAMMING APPROACH TO SOLVING OPTIMIZATION PROBLEMS ON AGENT-BASED MODELS

By

Anthony Garuccio

Thesis approved April 6, 2016.

Dr. Rachael Miller Neilan Assistant Professor of Mathematics (Committee Chair)	Dr. John Kern Associate Professor of Mathematics (Department Chair)
Dr. Donald L. Simon Associate Professor of Computer Science (Committee Member)	Dr. James Swindal Dean, McAnulty College and Graduate School of Liberal Arts Professor of Philosophy

ABSTRACT

A GENETIC PROGRAMMING APPROACH TO SOLVING OPTIMIZATION PROBLEMS ON AGENT-BASED MODELS

By

Anthony Garuccio

May 2016

Thesis supervised by Dr. Rachael Miller Neilan

In this thesis, we present a novel approach to solving optimization problems that are defined on agent-based models (ABM). The approach utilizes concepts in genetic programming (GP) and is demonstrated here using an optimization problem on the Sugarscape ABM, a prototype ABM that includes spatial heterogeneity, accumulation of agent resources, and agents with different attributes. The optimization problem seeks a strategy for taxation of agent resources which maximizes total taxes collected while minimizing impact on the agents over a finite time. We demonstrate how our GP approach yields better taxation policies when compared to simple flat taxes and provide reasons why GP-generated taxes perform well. We also look at ways to improve the performance of the GP optimization method.

Contents

Chapter 1

Agent-Based Models

1.1 What is an Agent-Based Model (ABM)?

An agent based model (ABM) is a computational model that simulates the local interactions of autonomous individuals (or agents) and their environment. Generally, ABMs are comprised of one or more types of agents and an environment containing parameters exogenous to the agents. The agents are given an internal set of attributes and a set of rules that determine their behavior as they interact with each other and the environment. ABMs can be run many times under different controlled conditions, allowing for researchers to observe system-level dynamics and test hypotheses.

Researchers in nearly all domains use ABMs to investigate how the behaviors of a complex systems arise from the characteristics and interactions of its individual components. Some examples of emergent behaviors described by ABMs are listed here.

- Brown et al. used an ABM to determine the effects of heterogeneity in residential preferences on urban sprawl. The ABM approach allowed them to study the effects of heterogeneity among agents on the overall system behavior to find that heterogeneity significantly affects aggregate patterns of urban sprawl and

clustering [2].

- Mi et al. used an ABM approach to model healing of diabetic foot ulcers to test the expected effectiveness of three treatments. They found that the expected effect of administering (1) a neutralizing anti-TNF antibody, (2) an agent that would increase the activation of endogenous latent TGF-β1, or (3) latent TGF-β1 all would have similar effects [11].

- Castella et al. used an ABM in combination with GIS data to understand land-use changes in the northern mountains of Vietnam based on the interactions between local farmer's decisions and institutional policies on resource use. This allowed them to identify geographic regions with similar trajectories and predict the effects on the natural resource base [3].

ABMs are becoming increasingly popular because they do not require "simplification" of model assumptions like traditional mathematical models. Equation-based mathematical models are limited in the complexity of the systems they describe due to the need for the equations to be solved analytically or by numeric approximation. For example, ordinary differential equation models often assume the environment is homogeneous and rates of change are uniform among groups of individuals. In contrast, ABMs can accommodate individuals that are different from one another and individual attributes can change with time or adapt to the environment. ABMs can simulate heterogeneous environments and allow for the environment to be modified by inhabiting agents. Therefore, ABMs are particularly attractive for describing complex system-level dynamics that evolve as the result of the agent level actions.

There is no particular programming language essential to the simulation of an ABM. It is possible to write an ABM in any language, including C, Fortan, and Python. However, programming languages specific to agent-based modeling exist and are widely used. Most notably, NetLogo [12] is the most commonly used because of its

accessibility to programmers of all abilities and its graphical interface. Constructing an ABM in non-agent-based programming language, such as Python, often requires more model development time, but it can lead to shorter simulation time and the ability to incorporate non-standard features [18].

1.2 The Sugarscape ABM

The agent-based model (ABM) that we use in our study is a modified version Epstein's Sugarscape ABM [6] that is described in detail in Oremland and Laubenbacher [14]. The model follows individual agents, called ants, as they move between patches on a two-dimensional spatial domain collecting and metabolizing sugar (i.e., wealth). Ants may be viewed as rational economic agents with the simple objective to maximize their personal sugar wealth. Taxation is included in the Sugarscape ABM such that a percentage of wealth is taken from each ant on a regular basis. If an ant's sugar wealth is depleted (through either metabolism or taxation), the ant dies.

Along with initial placement and initial wealth, there are two intrinsic attributes, vision and metabolism, that determine an ant's ability to accumulate sugar wealth over time.

1. **Vision.** An ant's vision indicates the number of spatial patches the ant can see in any direction from its current patch. Vision defines the ant's ability to relocate to an area with greater resources. The higher the ant's vision, the more likely it is that the ant will see and move to a patch with a higher sugar content.

2. **Metabolism.** An ant's metabolism is equal to the amount of sugar consumed by the ant each time step. Ants with high metabolism values will lose more sugar than ants with low metabolism values, thus making it less likely to accumulate sugar wealth over time and more likely to die.

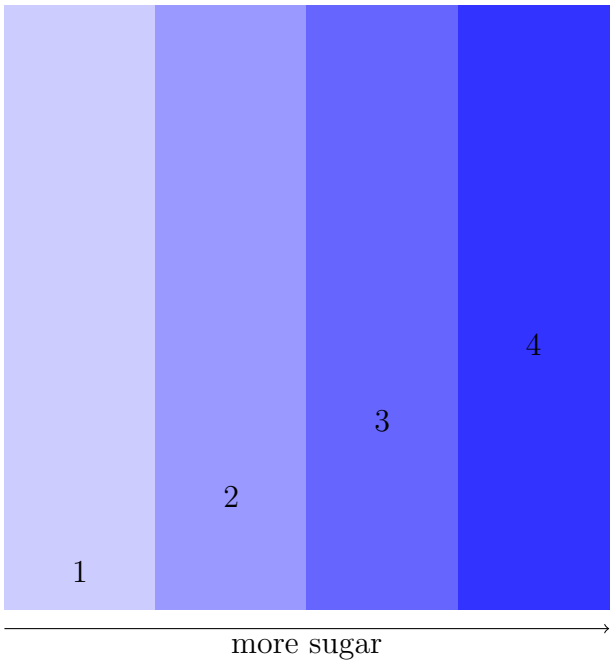

Figure 1.1: **Spatial Domain.** Ants are located on 48×48 patch grid and move between patches based on sugar availability. Regions 1, 2, 3, 4 describe amount of sugar available to inhabiting ants.

The spatial domain is a 48×48 patch grid on which each patch has a specified amount of sugar available to inhabiting ants. The Sugarscape ABM presented in Oremland and Laubenbacher [14] assumes the spatial domain is divided into four equal, vertical sections describing the amount of sugar available to inhabiting ants. As seen in Figure 1.1, the vertical regions (each of size 12×48) are arranged from least sugar on the left to most sugar on the right. Ants move independently at each time step to a patch within their vision that has the most sugar. Upon moving to a patch, an ant collects the full amount of sugar available. Sugar within a patch is never depleted and multiple ants may occupy one patch at a time. Ants may not cross the vertical and horizontal boundaries of the spatial domain.

Values of parameters used in the Sugarscape ABM are provided in Table 1.1. Each simulation begins with an initial population of 400 ants. Each ant is initialized by randomly selecting values for the ant's vision, metabolism, and initial sugar (see possible values in Table 1.1). An ant's metabolism and vision values remain the same throughout the simulation. The initial location of each ant is randomly selected from one of the 48×48 patches. The taxation rates to be applied to each ant are specified at initialization.

Figure 1.2 is an overview of the rules applied at each time step, or tick, during simulation of the Sugarscape ABM. Ants are randomly ordered at the beginning of each tick. The following rules are then applied to all ants, one at a time. The ant's status (dead or alive) is determined by checking if the ant's sugar wealth is greater than 0. If the ant is dead, no further action is taken on the ant. If the ant is alive, then the ant moves to a patch within its vision with the highest sugar content. If there are multiple patches with the highest sugar value, then the ant chooses one randomly. After the moving stage, the ant collects the sugar from its current patch, i.e. updates its sugar wealth by adding the sugar value of the patch to it is current sugar wealth. Next, the ant metabolizes sugar, i.e. updates its sugar wealth by subtracting the ant's metabolism value from its current sugar wealth. The ant's status (dead or alive) is again determined by checking if the ant's sugar wealth is greater than 0. If so, a

Table 1.1: **Model Parameters.** Values used in our ABM

Variable	**Description**	**Value**
Population size	Initial number of ants	400
Final time	Final tick of the simulation. The tick variable starts at 0.	4
Initial sugar	Possible initial sugar wealth for each ant	$\{1, 2, \ldots, 10\}$
Vision	Possible vision values for each ant	$\{1, 6\}$
Metabolism	Possible metabolism values for each ant	$\{2, 4\}$
Tax Rate	Proportion of sugar removed from an agent each tick	$[0, 1]$

percentage of the ant's sugar wealth is collected as a tax. The tax rate applied to each ant is a value between 0 and 1, inclusively. The simulation continues for 5 ticks or until all ants are dead. Note that the simulation runs for 5 ticks, so the value of the tick variable will be $\{0, 1, 2, 3, 4\}$.

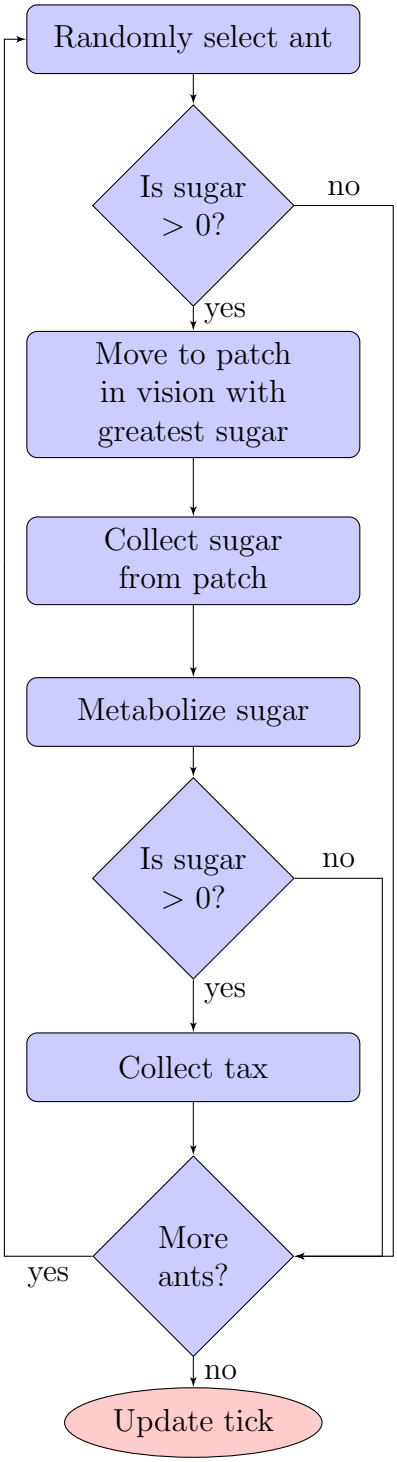

Figure 1.2: **Model Overview.** Diagram of rules applied to individual ants during each time step

Chapter 2

Optimization for Agent-based Models

2.1 An Optimization Problem for the Sugarscape ABM

Taxation of individual ant wealth is considered a control variable in the Sugarscape ABM. The model can accommodate a vast array of taxation policies. For instance, we can implement a simple flat tax (e.g., all ants are taxed 10%) or a complex taxation policy that depends on an ant's individual properties such as location, metabolism, vision, current wealth, etc. There are infinitely many tax policies that one can implement within the Sugarscape ABM.

The optimization problem posed in this study is to determine a taxation policy to be applied to ants in the Sugarscape ABM that will maximize the total sugar tax collected and the average lifespan of the ants in the population. Further, we want to keep the population-wide average tax rate low. The components of our objective are inversely related with respect to the control, i.e. taxation. Higher tax rates can yield a larger total tax collected, but can decrease the average lifespan of the population.

Lower tax rates generally lead to less tax collected, but result in a longer average lifespan of the population.

Formally, we express our optimization problem in terms of the quantity

$$J = C_1\ A_T + C_2 \sum_{i=0}^{T} \sum_{j=1}^{A_i} \text{Tax}_{i,j} - C_3\ \left[\sum_{i=0}^{T} A_i\right]^{-1} \sum_{i=0}^{T} \sum_{j=1}^{A_i} [(\text{TaxRate}_{i,j})^n] \tag{2.1}$$

where T is the final tick in the simulation, A_i is the number of ants alive at the ith tick, and $\text{Tax}_{i,j}$ is the total amount of sugar collected via taxation from the jth ant during tick i. The term $(\text{TaxRate}_{i,j})^n$ is the tax rate (between 0 and 1) of the jth ant during tick i raised to the nth power ($n \geq 1$). The value of n defines the relationship between the penalty of taxation and the rate of taxation. For $n = 1$, the relationship is linear such that a small increase in tax rates results in the same penalty increase regardless of the tax rate. For $n > 1$, the relationship is strictly convex such that small increases in tax rates lead to larger increases in the penalty term for larger tax rates. In equation (2.1) the final term is equivalent to taking the average of $(\text{TaxRate}_{i,j})^n$ over all ticks ($i = 0...T$) and all ants alive during each tick. Parameters C_1, C_2, and C_3 are balancing coefficients used to weight the three terms in the objective.

Since the Sugarscape ABM is stochastic, under any given tax structure the variables A_i, $\text{Tax}_{i,j}$, and $\text{AvgTaxRate}_{i,j}$ in equation (2.1) are not deterministic values, but rather random variables. Therefore the quantity J is also a random variable. Since ranking two random variables is not possible, we use the mean value of J obtained over a large number of simulations whenever we compare the performance of multiple controls. The optimization problem can be written as

$$\max_{\text{tax}} \bar{J} \tag{2.2}$$

where the maximum is taken over all possible tax structures and $\bar{J}$ is the mean value

of J obtained from 10 simulations of the ABM.

2.2 Previous work

There is no standard framework for solving optimization problems described by ABMs, such as the one in Section 2.1. Simulation and evaluation of all possible controls is often impractical or impossible. Several previous attempts at constructing optimal controls for an ABM have been made [4, 7, 8, 13, 14]. These approaches rely on the use of an equation-based approximation model. In this technique, the average temporal or spatiotemporal dynamics of the ABM are approximated by a system of mathematical equations (e.g., differential equations, polynomials). Control variables are included in the equations to mimic control in the ABM. Known optimization techniques and mathematical theory are applied to the equation model to construct controls that optimize the objective function as it is written in terms of the equation model. The resulting optimal control (generated by the equation model) is then transformed for implementation within the ABM and evaluated within the framework of the ABM.

Examples of equation-based approximation models and methods of optimization include

- Hinkelmann et al. [8] describe a framework for translation of ABM rules into polynomial dynamical systems. The polynomial dynamical system is optimized using known algebraic optimization methods.

- Federico et al. [7] approximate a predator-prey ABM using ordinary differential equations (ODEs) and apply optimal control theory to the ODE model to estimate optimal predator harvesting rates. The same ABM is approximated by Oremland et al. [13] using difference equations and Pareto optimization is applied to construct optimal controls.

- Christley et al. [4] approximate the Sugarscape ABM with a system of partial differential equations (PDE). The PDE accounts for movement of ants across the spatial environment and accumulation of sugar wealth over time. Optimal control theory is applied to the PDE model to construct optimal tax rates.

The technique of using an approximation model has been shown to work for some simple ABMs, but has failed to work when complex features are included in the ABM. Specifically, errors can occur when attempting to implement optimal controls from the approximation model within the ABM. For example, when spatial heterogeneity is added to the predator-prey ABM, the approximation model in [7] fails to yield an accurate representation of ABM dynamics essential to the optimization problem. Optimal controls from the approximation model are no longer near optimal when implemented in the ABM. Similarly, when the spatial domain of the Sugarscape ABM is modified from the one pictured in Figure 1.1, the methods of Christley et al. [4] fail becuse construction of the PDE approximation model relies heavily on the simple sugar landscape. In general, complex ABMs cannot be approximated well by closed-form equations. In fact, one of the main reasons that ABMs are chosen to model a system is that the rules governing the system are too complicated to be expressed using other, non-simulation based, classical techniques.

The main motivation for our research is to develop and explore a simulation-based optimization technique that can be applied to a broad range of ABMs. Our approach uses genetic programming and is demonstrated in this thesis on the optimization problem for the Sugarscape ABM (Section 2.1). Sugarscape is an excellent prototype ABM because it includes agents with different attributes and a heterogeneous environment. As far as we know, this is the first attempt at using genetic programming for constructing optimal controls for an optimization problem described by an ABM.

Chapter 3

A Novel Solution Approach

3.1 What is Genetic Programming (GP)?

Genetic programming is a metaheuristic for constructing computer programs for the goal of optimizing a given fitness landscape. By mimicking the processes of biological evolution and natural selection, genetic programing evolves a population of computer programs to produce one or more programs with optimal (or near optimal) fitness. Each individual (i.e. computer program) in the population is an expression tree constructed from a set of values and a set of operations to be applied to the values. An individual computer program can be thought of a tree comprised of non-leaf (i.e. non-terminal) nodes from the set of operations and leaf (i.e. terminal) nodes from the set of values.

Once a population is formed, each individual in the population is evaluated in terms of its fitness. In this context, fitness is a real-valued measure of how well an individual performs with respect to a given optimization problem. After forming a population and evaluating individual fitness values, a tournament-like step creates a new population from the old such that the probability of an individual advancing to the new population is dependent upon its fitness. In other words, individuals with

better fitness values are more likely to remain in the population whereas individuals with worse fitness values are more likely to be eliminated.

Once a new population is formed, genetic operations are applied to evolve individuals within the new population. These operations can include the following.

1. **Expression Mutation.** A subtree of the individual is selected and replaced by a new, randomly generated subtree.

2. **Mating (Crossover).** Two individuals are selected, then within each a random node is selected. The two subtrees with the selected nodes as roots are swapped between the individuals.

After applying genetic operations, the process of evaluating, selecting, and evolving a new population is repeated until a specified stopping criteria has been met. Algorithm 1 displays psuedo-code for genetic programming. Our implementation of genetic programming is performed using Distributed Evolutionary Algorithms in Python (DEAP). Specific details regarding the DEAP package are provided in Section 3.2.

Genetic programming, like most metaheuristics for solving non-convex problems,

Algorithm 1 Genetic Programming

```
initialize population
while not stopping-criteria do
  for individual in population do
    evaluate individual fitness
  end for
  for i < population-size do
    randomly select 4 individuals from population
    copy best individual to new-population
    increment i
  end for
  population ← new-population
  apply crossover operation
  apply mutation operation
end while
```

must balance the tasks of moving towards better known solutions while avoiding being trapped in local optima. This task is accomplished using a probabilistic tournament feature in which all individuals have a positive probability of moving on to the next generation, with larger probabilities assigned to individuals with better fitness. While the tournament method moves the population towards known local optima, the genetic operations–such as mutation–introduce new genetic material so that the metaheuristic can explore different parts of the global search space. Mating, or crossover, is a unique way to explore the search space, where parts of existing, well-performing individuals are spliced together with the possibility that effective pieces from each will combine into an individual that is greater than either parent.

Genetic programming is increasingly used for optimization in many disciplines. For example, optimization of database query can be performed using genetic programming, and this method is preferred over alternative optimization methods, like combinatorial search, due to its robustness and efficiency [17]. Construction of equations that mimic biological processes is another another application of genetic programming. In [16], the authors model a gene regularity network using a system of ordinary differential equations in which the right-hand sides were inferred from time-series data using genetic programming. Genetic programming is even used in the arts. One user constructed a genetic programming algorithm to replicate the Mona Lisa using polygons evolved to mimic the real image [9] .

Despite its growing popularity, genetic programming has yet to be employed for the construction of optimal controls in an agent-based model. In the next section, we describe a novel approach to solving the Sugarscape optimization problem using genetic programming. We use the Sugarscape ABM and optimization problem as a benchmark for testing our approach, but the approach can be easily adapted for the optimization of other ABMs.

3.2 Application to the Sugarscape Optimization Problem

3.2.1 Defining the Population

The goal of the Sugarscape ABM optimization problem is to construct a tax structure that maximizes the total sugar tax collected and the average lifespan of the ants in population, while minimizing the average tax rate. In the context of genetic programming, each possible tax structure is an individual that may be included in the population. An individual (i.e. tax structure) can be viewed as a string of functions or as an expression tree for better visualization. For example, consider the tax structure that suggests each ant with a metabolism equal to 2 be taxed 20% of its current sugar wealth; otherwise, do not tax the ant. This individual can be represented by the string if-then-else(eq(metabolism,2), 0.2, 0) or visualized as the tree depicting in Figure 3.1.

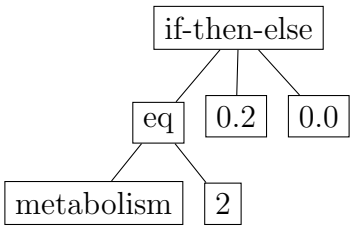

Figure 3.1: Expression tree for the tax structure that applies a 20% tax rate to all agents with metabolism equal to 2 and no taxation of all other agents.

Individuals are formed from the set of all possible operators and the set of all possible terminal nodes defining a tax structure. These two sets define the functional space, i.e. domain, that the genetic programming algorithm will search for a tax structure. Table 3.1 lists the set of all nodes used in our simulations, while Table 3.2 lists the set of all operators used in our simulations.

Table 3.1: **Terminal Set** All possible terminal nodes for tax structures in the Sugarscape ABM

Variable	**Description**
sugar	Current sugar-wealth of the ant
vision	Vision value of the ant
metabolism	Metabolism value of the ant
x	Horizontal value of the patch the ant is occupying
y	Vertical value of the patch the ant is occupying
patchSugar	Amount of sugar on the patch the ant is occupying
tick	Time-step of the simulation
Constant	Constant selected from $\{0, 0.1, 0.2, ..., 0.9, 1, 2, 3, ..., 6\}$

Table 3.2: **Operator Set** All possible operators for tax structures in the Sugarscape ABM

Operator	**Description**	**Arguments**	**Return**
add	simple addition	[float, float]	float
sub	simple subtraction	[float, float]	float
mul	simple multiplication	[float, float]	float
protectedDiv	division, return 1 if denominator is 0	[float, float]	float
gt	True if first argument greater than the second	[float, float]	bool
lt	True if first argument less than the second	[float, float]	bool
eq	True if first argument equals the second	[float, float]	bool
and	True if both arguments are True, otherwise False	[bool, bool]	bool
or	True if either argument is True, otherwise False	[bool, bool]	bool
not	negates input boolean	[bool]	bool
if-then-else	returns first argument if true, otherwise returns second argument	[bool, float, float]	float

An individual is applied to the Sugarscape ABM by applying the tax structure at each time step to all living ants, as described in Section 1.2. During taxation, an additional step is taken to ensure the tax rate generated by the individual is in the interval $[0, 1]$. It is possible that a randomly generated individual will produce a tax rate outside of $[0, 1]$, so we apply the following min-max operation to each tax rate when it is applied to an agent

$$\text{taxRate}_{i,j} = \min(1, \max(0, \text{individual}(\text{ant}_j, \text{tick}_i)))$$

This allows our genetic programing algorithm to work on an unconstrained space of all expression trees formed from our terminal and operator node sets, since all possible generated tax structures will be valid.

3.2.2 Initializing the Population

To initialize the population, we randomly generate a set of individuals from our operator set (Table 3.2) and terminal node set (Table 3.1). There are two well known methods used to generate an individual for the initial population [10]. These methods are referred to as "full" and "grow".

Using the full method, we randomly draw from the operator set until we reach a tree depth of one less than the maximum tree depth, then we draw from the terminal set to fill in the leaves. This results in trees with the property that all leaves (terminal nodes) are at the same depth. The grow method, on the other hand, creates trees by randomly drawing from either the terminal or operator set until the tree is complete or until the tree depth is one less than the depth limit, then after that point, further selections are made only from the terminal set. While the full method produces mostly symmetric trees of exactly the maximum depth limit, the grow method tends to produce much smaller, less symmetric trees.

Used individually, the full and grow methods produces a population with little variation in tree shapes and sizes. Therefore, we use a combination of the two methods called ramped-half-and-half. Ramped-half-and-half uses a variety of max tree depths and creates the population by using the grow method half the time and the full method the other half of the time. This combination approach has been shown to create a population with a variety of tree shapes and sizes, which is necessary to properly explore the search space [10]. In our implementation we used max tree depths ranging from 1 to 4.

3.2.3 Evaluation and Selection

During the evaluation step, we calculate the fitness of each individual in the population with respect to the optimization goal of the Sugarscape ABM. The fitness of an individual is calculated as

$$\text{Fitness} = \bar{J} \tag{3.1}$$

where J is defined in equation (2.1) and $\bar{J}$ is the mean taken over 10 simulations of the ABM.

Many implementations of genetic programming only reevaluate the fitness of the individual if the individual is changed in some way, such as through mutation. This can save significant computation time, since only a portion of the population has its fitness reevaluated at each iteration of the genetic programing algorithm. However, due to the fact that our fitness function is stochastic rather than deterministic, we reevaluated the fitness of every individual during each iteration of the genetic programming algorithm.

After evaluating the fitness of each individual in the population, a tournament selection is applied to determine which individuals in the population should move on to the new population. To implement this, $t = 4$ individuals from the current

population are randomly selected and the one with the best fitness value is copied to the new population. A copy of the selected individual remains in the old population. This procedure is repeated until the new population is the same size as the old population, having 250 individuals.

It should be noted that when a larger number individuals are chosen each round for the tournament, the likelihood of getting many copies of the current best individuals in the new population increases. To illustrate this, consider two extreme cases. If we only choose one individual each round, then the state of the population will not evolve towards containing individuals with better fitness. On the other hand, if we choose all of the individuals in the population each time, then the new population will contain only copies of the current best performing individual. In our implementations, we use 4 individuals selected for each tournament because this quantity balances the goals of evolving the population toward better performers and keeping genetic diversity present in each new population.

Blickle and Thiele [1] define the reproduction rate, $\bar{R}(f)$ as the expected ratio of the number of individuals with a certain fitness value, f, before and after the selection step. They calculate the reproduction rate for a tournament selection to be

$$\bar{R}(f) = t \left(\frac{\bar{S}(f)}{N} \right)^{t-1} \tag{3.2}$$

Where $\bar{S}(f)$ is the number of individuals with fitness value f or worse, t is the number of individuals selected during each tournament, and N is the population size.

Using our chosen tournament size of $t = 4$ and our population size of 250, the reproduction rate for our implementation is

$$\bar{R}(f) = 4 \left(\frac{\bar{S}(f)}{250} \right)^{3} \tag{3.3}$$

Figure 3.2 shows that our chosen tournament size will create more copies of better

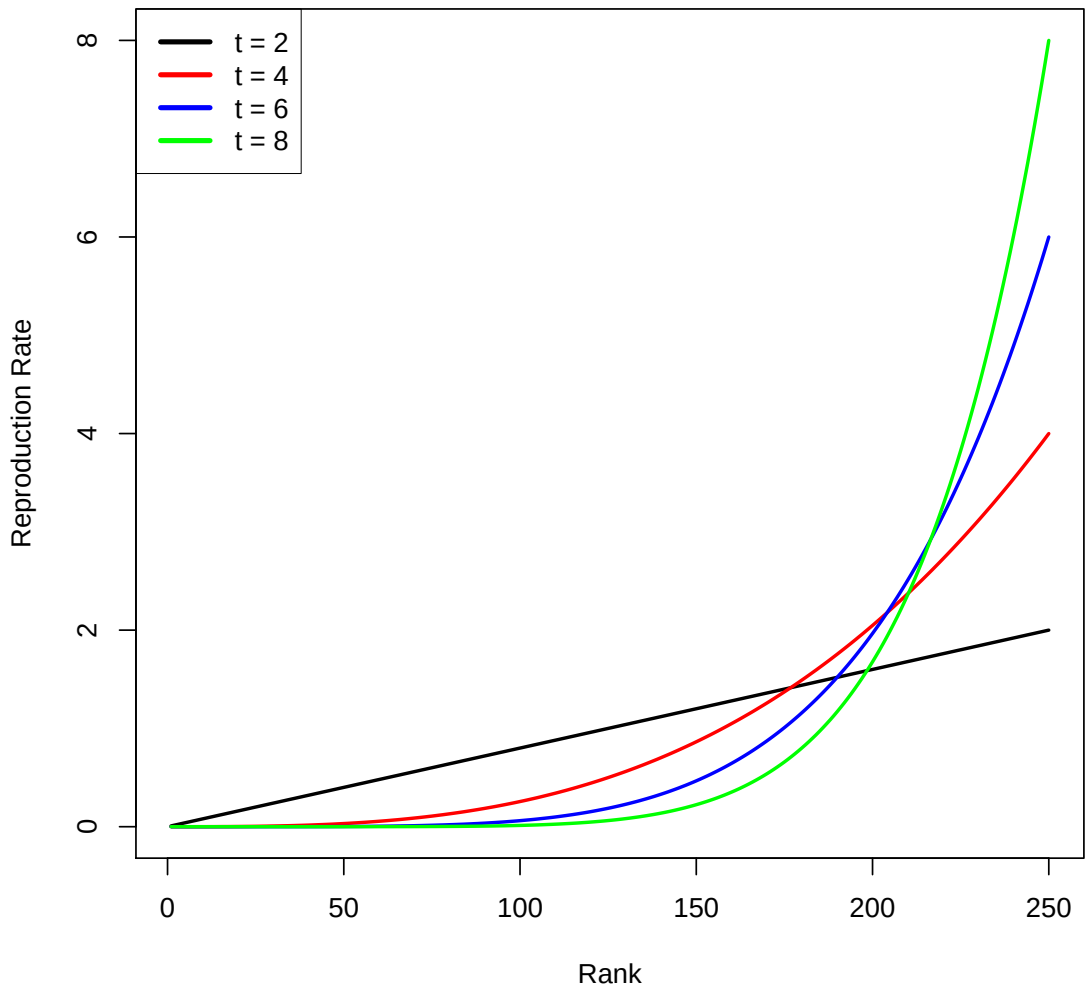

Figure 3.2: Expected reproduction rate for each rank in our population with rank 250 being the best and rank 1 being the worst. It is assumed no ties are present

performing individuals and less copies or worse performing individuals, moving the population as a whole to better solutions.

As mentioned above, while evolving toward better solutions, it is also important to not lose genetic diversity in the population too quickly or the algorithm will converge on local optima rather than exploring the search space for the global optima. To measure this effect Blickle and Thiele [1] define the loss of diversity, p_d as the expected

proportion of individuals that will not be selected into the new population after an application of the tournament selection procedure.

$$p_d = t^{-\frac{1}{t-1}} - t^{-\frac{t}{t-1}} \tag{3.4}$$

Where t is the number of individuals selected during each tournament. Using our tournament size value of $t = 4$, we find $p_d = 0.47$. So, after each tournament selection step, we expect 47% of the current population to not move on to the next iteration.

3.2.4 Genetic Operations

Crossover

To apply the crossover operation, we define P_{cross} as the probability that individual$_i$ and individual$_{i+1}$ will mate. Then we loop over our population and nominate each pair of individuals for crossover with probability P_{cross}. Once nominated, we choose a random node in each of the individual trees as our crossover point. Then we take the two subtrees with the crossover points as the roots and swap them. We then continue looping through the population until we nominate another pair or reach the end of the population. In our implementation, we chose $P_{cross} = 0.3$ as was chosen in several of the examples in the DEAP software documentation [5].

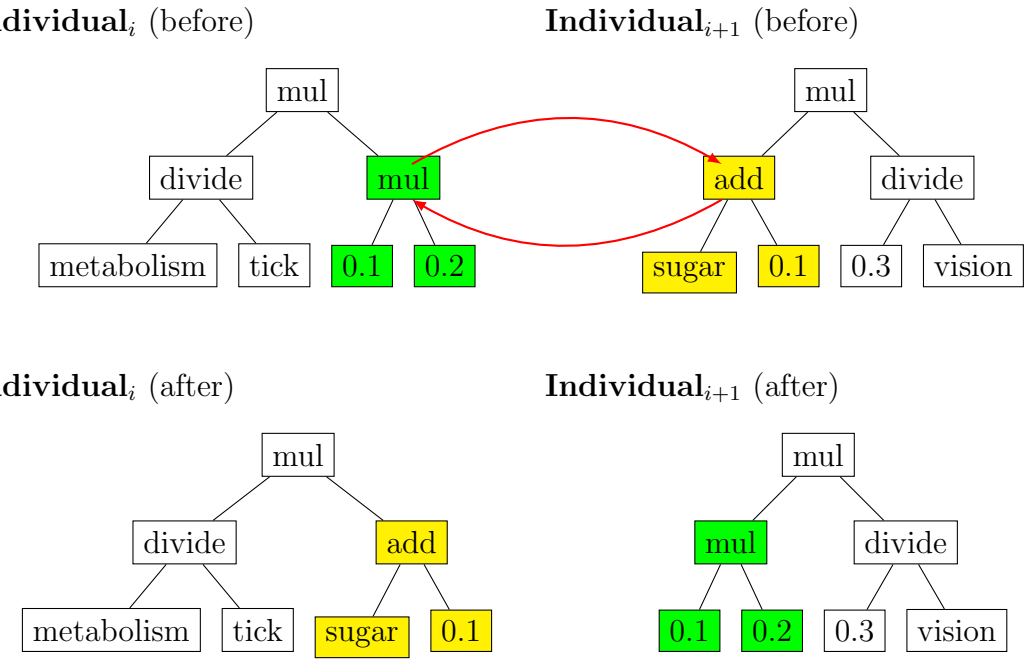

Figure 3.3: Individuals i and $i+1$ (top row) are nominated for crossover. A subtree from Individual i (green) and a subtree from Individual $i+1$ (yellow) are randomly selected. Two new individuals (bottom row) are created using the crossover operation by swapping the green and yellow subtrees.

Expression Mutation

To apply the expression mutation operation, we need to first define P_{mut} which will be the probability that a mutation will occur on any given individual. Once we have defined P_{mut} then we will loop through the population and nominate each individual for mutation with probability P_{mut} . Once nominated, a random node of the individual tree is chosen and the subtree with that node as the root is replaced by a new tree that was generated using the ramped-half-and-half method. We then continue looping through the population until we nominate another individual or reach the end of the population. For our implementation, we used $P_{mut} = 0.2$. This value was used in several of the examples in the DEAP software documentation [5].

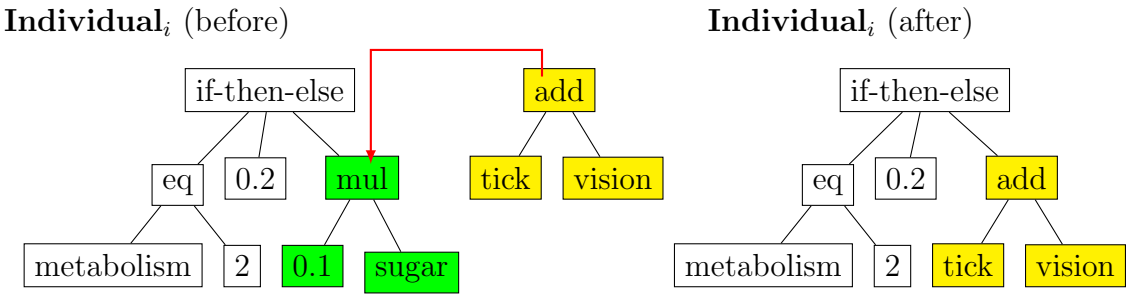

Figure 3.4: Individual i (left) is nominated for expression mutation. A subtree from Individual i (green) is randomly selected and a new subtree (yellow) is grown using ramped-half-and-half method. A new individual (right) is created using the mutation operation by replacing selected subtree (green) with newly grown subtree (yellow)

3.2.5 Stopping Criteria

The genetic programming algorithm can be made to stop by various criteria. For example we could stop after a fixed number of iterations or we could stop once our best fitness is within ϵ of some known best fitness value. We chose to have the genetic programming algorithm terminate when the same individual remains the most optimal (compared to the current population) for a sufficiently long number of evolutionary cycles. For our implementations, we assumed 10 cycles was sufficient for establishing convergence to the optimal result. Upon termination, the algorithm outputted the optimal individual and its fitness.

Chapter 4

Results

4.1 Parameters

We present the results of our approach to solving the Sugarscape optimization problem using the parameters in Table 4.1. The coefficients C_1 and C_2 in equation (2.1) were chosen so that nearly equal weight was placed on maximizing the number of ants alive at the final time and maximizing the total taxes collected. The value of C_3 in equation (2.1) depends on n, the parameter defining the relationship between the penalty of taxation and the rate of taxation in the optimization problem. The relationship $C_3 = -2500(2^{n-1})$ was chosen to ensure the penalty term $C_3 \left[\sum_{i=0}^{T} A_i\right]^{-1} \sum_{i=0}^{T} \sum_{j=1}^{A_i} [(\text{TaxRate}_{i,j})^n]$ in equation (2.1) is equivalent for all values of n when the Tax Rate is 0.5 for all ants at all times. Figure 4.1 illustrates the value of the penalty term for different values of n assuming all ants incur the same constant tax rate.

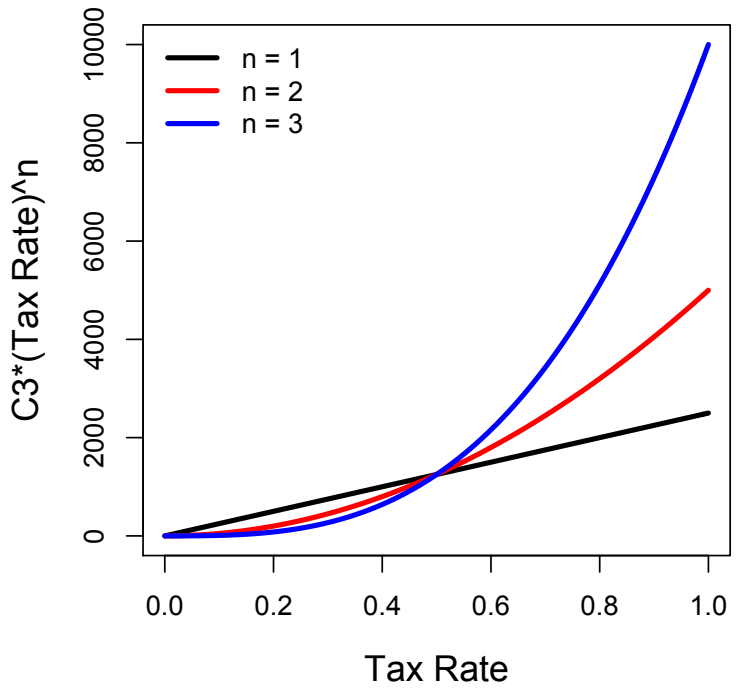

Figure 4.1: Curves show the relationship between the penalty term (third term in equation (2.1)) and tax rates for $n = 1, 2, 3$. Here it is assumed that a tax rate is applied to all ants at every tick.

Table 4.1: Parameters used equation (2.1) defining the Sugarscape optimization problem.

Parameter	**Description**	**Value**
C_1	Coefficient of the term describing the number of ants alive at the final tick	10
C_2	Coefficient of the term describing the total tax collected	1
C_3	Coefficient of the term penalizing high tax rates	$-2500(2^{n-1})$
n	Exponent on tax rates in penalty term	1,2, or 3

4.2 Baseline Simulations

To create a baseline against which we can compare the output of our genetic programming algorithm, we first simulate the Sugarscape ABM using constant tax rates ranging from 0% to 90% by increments of 10%. Figure 4.2 shows the distribution of fitness values for each of the flat taxes for $n = 1, 2$, and 3.

As expected, for each value of n, fitness values demonstrate a parabolic shape with high and low tax rates performing poorly. The parabolic shape is the result of the two main components of the fitness function. When the tax rate is low, the term measuring the number of ants alive at the end of the simulation is a strong positive contributor to the fitness function while the term measuring the total taxes collected is relatively small. Conversely, when the tax rate is high, many ants will die and therefore the term measuring the number of ants alive at the end of the simulation is very small, but the total taxes collected is a large positive value. The value of n affects the third and final term of the fitness function which adds a penalty to high tax rates. For higher values of n, a larger penalty is associated with tax rates approaching 100%. In the $n = 1$ case, the series of box plots are roughly symmetric, with both very low and very high rate performing equally poorly. For $n = 2$ and 3, a dramatic decrease in the performance of the higher tax rates is observed. This is a result of the heavy penalty that is associated with high values of the average tax rate.

In all three cases ($n = 1, 2, 3$), the best performing flat tax is 30%. In the next section, the 30% flat tax is used as a comparison for the best performing taxes generated from genetic programming algorithm. This comparison provides us with a sense of well the generated tax structures perform in comparison to basic solutions.

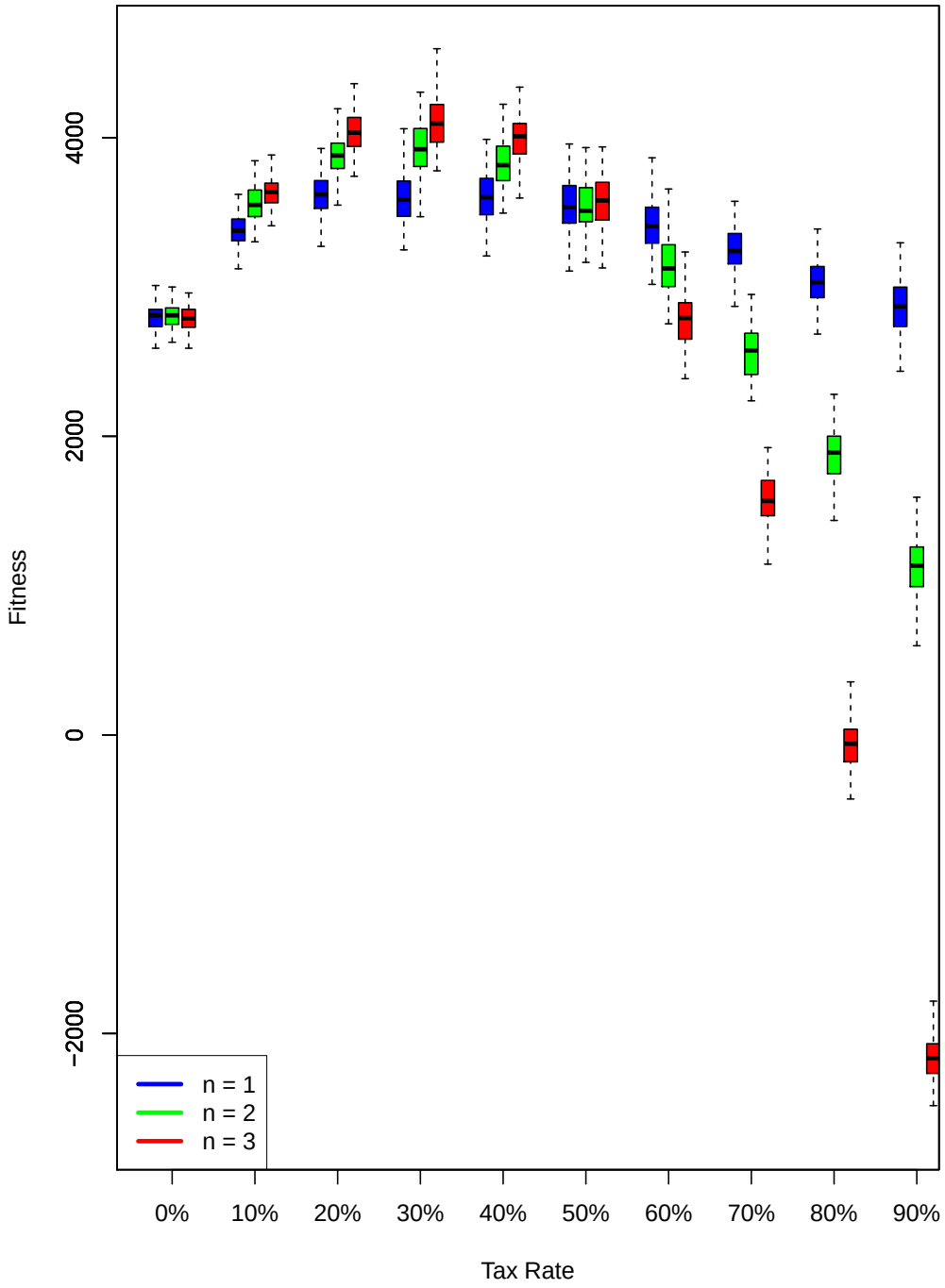

Figure 4.2: Box plots of 1,000 runs of the Sugarscape ABM under the different flat tax controls from 0% to 90% for $n = 1, 2, 3$.

4.3 Optimal GP results

After multiple applications of our genetic programming algorithm, Table 4.2 summarizes the three best performing tax structures discovered by the algorithm in each of the $n = 1, 2, 3$ cases. Included in the table is a string representation of the tax structure along with the average value of each component of our fitness function, J, computed over 1000 ABM simulations. In the following sections, we will examine these different control structures and the strategies they use to maximize the fitness function to find emergent behavior across the different controls and across the different values of n.

4.3.1 Tick-Based Control Strategy

For each value of n, one of the three best performing controls was based solely on the value of the tick variable. The general structure, or strategy, of the tick-based controls is to start with a very low tax rate at the beginning of the simulation and

Table 4.2: **GP Output** Tax structures generated by our genetic programming algorithm for $n = 1, 2, 3$ with baseline flat tax of 0.3 for comparison

n	**Function**	**Ants Alive (C_1 Term)**	**Total Tax (C_2 Term)**	**Penalty (C_3 Term)**	**Fitness**
1	0.3	2424.20	1946.29	-750.00	3620.49
1	$0.015 * (\text{tick}^3)$	2723.81	2350.05	-664.12	4409.74
1	$0.07 * \text{sugar}$	2416.74	1989.75	-649.70	3756.79
1	0.8/metabolism	2507.26	2037.97	-798.48	3746.75
2	0.3	2426.92	1949.80	-450.00	3926.72
2	$(0.4/3) * \text{tick}$	2648.94	1935.45	-474.52	4109.87
2	$0.04 * \text{sugar}$	2640.15	1632.66	-207.98	4064.83
2	$\text{patchSugar}/(6 * \text{metabolism})$	2744.01	1636.90	-234.60	4146.31
3	0.3	2428.63	1945.02	-270.00	4103.65
3	$0.1 * \text{tick}$	2722.70	1622.81	-176.73	4168.78
3	0.7/metabolism	2555.89	1909.82	-277.52	4188.19
3	$\text{patchSugar}/(5 * \text{metabolism})$	2678.30	1819.90	-219.27	4278.93

increase the rate as time progresses. The tick-based tax performs well because low initial tax rates allow ants to collect as much wealth as possible during the first few ticks of the simulation, causing very little additional death than what would have occurred naturally. As the time progresses, tax rates increase quickly to collect as much tax as possible. In this way, the strategyy maximizes both the number of ants alive as well as the total amount of tax collected. The tick-based control performs the best by a large margin in the $n = 1$ scenario. From Table 4.2 you can see that not only does this control have the best overall fitness, it also dominates all the other controls in the ants-alive term as well as the tax-term. See Table 4.3 for the values of the tax rate as the tick progresses throughout the simulation for the $n = 1$ case.

Table 4.3: Tax rate table for $(0.015 * \text{tick}^3)$

Tick	**Tax Rate**
0	0.000
1	0.015
2	0.120
3	0.405
4	0.960

When $n = 2$, the tick-based strategy is still better than the baseline flat tax. However, it no longer performs better than all the other controls generated by the genetic programming algorithm. This is due to the fact that when we increase n to 2, we remove a key component of the strategy that the previous tick-based control was exploiting. Recall from the baseline simulations that flat tax rates above 50% are hit with a severe penalty term in the $n = 2$ and $n = 3$ cases, so the strategy that the previous tick based control used where we tax very low in the beginning and then tax very high at the end no longer works due to the high tax rate penalty. See Table 4.4 for the values of the tax at each tick.

Finally, in the $n = 3$ case, the tick-based strategy becomes the least effective of the three strategies. Table 4.5 shows the tax rates for each tick. Notice that as n

Table 4.4: Tax rate table for $(0.4/3) * \text{tick}$

Tick	Tax Rate
0	0.000
1	0.133
2	0.267
3	0.400
4	0.533

increases, the average tax rate over all the ticks decreases as well as the highest tax rate imposed on tick 4. This is because as we increase the penalty of high tax rates (i.e. increase n) the benefit of high taxes collected at the end times is offset by the penalty term.

Table 4.5: Tax rate table for $0.1 * \text{tick}$

Tick	Tax Rate
0	0.000
1	0.100
2	0.200
3	0.300
4	0.400

4.3.2 Sugar-Based Control Strategy

For both the $n = 1$ and $n = 2$ cases, a control structure emerged in which individual tax rates are proportional to an ant's current sugar wealth. This structure makes sense because taxing the ants with the highest amount of sugar allows us to maximize the amount of tax collected while also minimizing deaths of ants with lowest sugar.

A positive linear relationship between taxes collected and sugar wealth is presented in the sugar-based strategy. As we would expect, the slope, or proportionality constant, decreases from the $n = 1$ to the $n = 2$ case. This is again due to increased penalty for high tax rates with larger values of n. The sugar-based control strategy does not appear in the $n = 3$ case. This is likely do to the fact that some ants with

high sugar wealth would be taxed close to 100% with the sugar-based control and that would result in a steep penalty and make the strategy less optimal for large n.

4.3.3 Metabolism-Based Control Strategy

The last control strategy that emerged in all three of our n cases was a metabolism-based control in which ants are taxed at a rate inversely proportional to metabolism. The general strategy of the metabolism-based controls is to tax the ants with lower metabolism more because they are most likely to gain the most sugar over time (and therefore least likely to die). Recall that ants in the Sugarscape ABM have two possible metabolism values, 2 or 4. Therefore, in the $n = 1$ case, the metabolism-based control (0.8/metabolism) taxes the low metabolism ants at 40% and the high metabolism ants at 20%. In the $n = 3$ case, the metabolism-based control (0.7/metabolism), taxes the low metabolism ants at 35% and the high metabolism ants at 17.5%.

Also included in this group of metabolism-based controls are the controls that utilizes the patchSugar variable in addition to metabolism. These controls account for both the income and consumption rates of the ants and apply the highest tax rates to those who collect the most and consume the least each tick. The best performing control in the $n = 2$ case, (patchSugar/(6 $*$ metabolism)), is an example. This control is proportional to patchSugar (i.e. the more sugar collected by an ant, the higher the tax rate) and inversely proportional to metabolism (i.e. the more sugar lost by an ant through metabolism, the lower the tax rate). Table 4.6 summarizes the tax rates for each possible combination of patch sugar and metabolism for this control. In the $n = 3$ case, the best performing control overall was (patchSugar/(5 $*$ metabolism)) and Table 4.7 summarizes the tax rates for each possible combination of patch sugar and metabolism for this control.

Table 4.6: Value table for patchSugar/(6 ∗ metabolism) when $n = 2$

	metabolism	
patchSugar	2	4
1	0.083	0.042
2	0.167	0.083
3	0.250	0.125
4	0.333	0.167

Table 4.7: Value table for patchSugar/(5 ∗ metabolism) when $n = 3$

	metabolism	
patchSugar	2	4
1	0.100	0.050
2	0.200	0.100
3	0.300	0.150
4	0.400	0.200

4.4 Improving GP Performance

There are several strategies that can be employed to enhance the performance of the genetic programming algorithm. One such technique is to seed the population with good performing individuals found in previous runs of the algorithm. The hope is that by seeding the population with good candidates, the algorithm will take the best performing aspects of the individuals and combine them to create an even better candidate. Alternatively, if a better solution is not created by cross-breeding the seeded individuals, then there is still a possibility that, through mutation, the algorithm will modify and improve an already well performing individual.

Another strategy for improving the performance of the genetic programming algorithm is to modify the tournament size, t. As described in detail in Section 3.2.3, the tournament size has a strong influence over how quickly our population converges upon the best individual. The previously used tournament size of $t = 4$ would result in the algorithm converging quickly to one of the best performing individuals in a

seeded population. To ensure appropriate time is provided for new genetic material to combine and improve upon the seeded population, a decrease in tournament size is needed.

We demonstrate how to modify the genetic programming algorithm for the $n = 2$ case, and evaluate the results compared to those in the previous section. To seed the population, ten copies of each of the three best individuals from the $n = 2$ case (Table 4.2) are placed in the initial population for the genetic programming algorithm. With the seeded initial population, the genetic programming algorithm runs as previously described in Chapter 3. All parameters specific to the Sugarscape optimization problem and genetic programming algorithm remain the same, with the exception of the tournament size, t. The tournament size is now set at $t = 3$.

4.4.1 Results for $n = 2$

By seeding the population and decreasing the tournament size, the genetic programming algorithm converged to a new solution that performed better than any of the previous controls obtained for $n = 2$ in Section 4.3. The new control is defined as $(\text{tick}/4) * (\text{patchSugar}/(3 * \text{metabolism}))$ and its performance is compared to the previous set of best solutions for $n = 2$ in Table 4.8 and Figure 4.3.

It is interesting that this new solution is a combination of the tick-based control strategy from Section 4.3.1 and the metabolism-based control strategy from Section 4.3.3. By combining both strategies into a single control is performs better than each strategy on its own. Table 4.9 shows the actual tax rates for every possible combinations of tick, metabolism, and patchSugar.

Table 4.8: **GP Output with Seeding** Tax structure generated by our genetic programming algorithm for $n = 2$ when seeded with the best performing results from Table 4.2. Included for comparison are the previous results and the baseline flat tax of 0.3.

Function	Ant Term	Tax Term	Rate Term	Fitness
$(\text{tick}/4) * (\text{patchSugar}/(3 * \text{metabolism}))$	2797.94	1827.04	-356.61	4268.37
0.3	2426.92	1949.80	-450.00	3926.72
$(0.4/3) * \text{tick}$	2648.94	1935.45	-474.52	4109.87
$0.04 * \text{sugar}$	2640.15	1632.66	-207.98	4064.83
$\text{patchSugar}/(6 * \text{metabolism})$	2744.01	1636.90	-234.60	4146.31

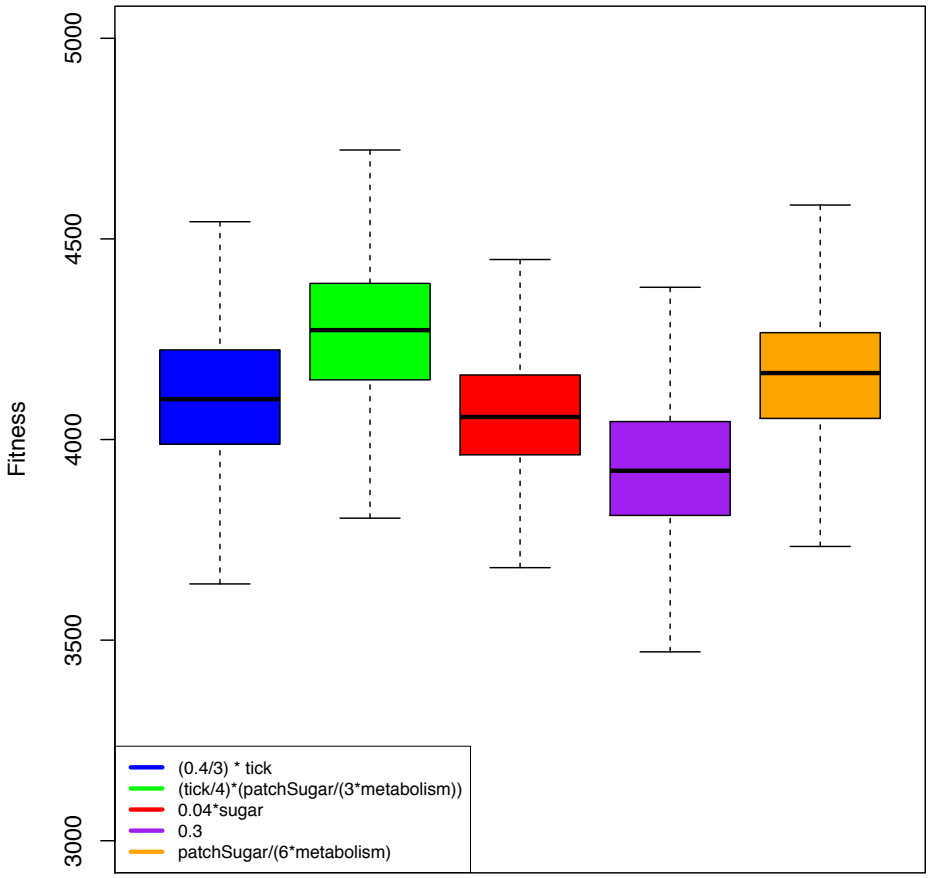

Figure 4.3: Box plots of 1000 runs of the the tax structure generated by our genetic programming algorithm for $n = 2$ when seeded with the best performing results from Table 4.2. Included for comparison are the previous results and the baseline flat tax of 0.3.

Table 4.9: Tax rate values for $(\text{tick}/4) * (\text{patchSugar}/(3 * \text{metabolism}))$ when $n = 2$

		metabolism: 2				**metabolism: 4**			
patchSugar:		1	2	3	4	1	2	3	4
tick	0	0.000	0.000	0.00	0.000	0.000	0.000	0.000	0.000
	1	0.042	0.083	0.125	0.167	0.021	0.042	0.063	0.083
	2	0.083	0.167	0.250	0.333	0.042	0.083	0.125	0.167
	3	0.125	0.250	0.375	0.500	0.063	0.125	0.188	0.250
	4	0.167	0.333	0.500	0.667	0.083	0.167	0.250	0.333

Chapter 5

Discussion and Conclusions

In this thesis, we examined a new approach to solving optimization problems for agent-based models (ABM). Our approach was demonstrated here using the Sugarscape ABM because this ABM is a prototype ABM that includes important ABM features such as spatial heterogeneity, accumulation of agent resources, agents with different attributes, and emergent behavior from interactions between agents and the environment. The Sugarscape ABM optimization problem seeks a strategy for taxation of agent resources which maximizes total taxes collected while minimizing impact on the agents over a finite time. Our solution approach utilizes the metaheuristic of genetic programming (GP) to evolve controls (i.e. taxation strategies) based directly on their fitness as solutions to the optimization problem.

5.1 Positive Outcomes

The fitness function in our optimization problem included three terms (1) maximize total ants alive, (2) maximize taxes collected and (3) minimize a penalty for high average tax rates. An exponent n was included in the penalty term to describe the convex relationship between taxes and penalty. For all values of $n = 1, 2, 3$, our GP approach produced controls that performed better than any constant tax rate

from our baseline simulations. The GP-generated controls were functions of tick, metabolism, patchSugar, and/or sugar. The non-constant controls provided insight into some of the strategies that could be used to develop optimal taxation policies on the Sugarscape ABM. By comparing results for the different n values, we gained insight into which strategies became more (or less) optimal when increased emphasis was placed on penalizing high tax rates.

Further simulations of the GP were performed to gain a better understanding of parameters specific to the GP algorithm itself, namely the tournament size t. We seeded the population in the GP from previous results in the $n = 2$ case in an effort to generate an improved control. However, seeding the population required us to decrease the tournament size to allow more evolutionary time. By seeding the population and decreasing the tournament size, the GP algorithm was able to produce a new solution that performed better than any of the previous ones in the $n = 2$ case.

5.2 Drawbacks

While our GP-based approach did produce interesting solutions that performed better than the baseline constant controls, we have no way of knowing if there exists solutions that with better fitness. This drawback is inherent in all simulation-based metaheuristics. The issue is that, on any finite time scale, there is no guarantee that the solution found by the GP algorithm is a globally optimal solution.

Another drawback of this method is that results can be inconstant and not easily reproduced due to the stochastic nature of the GP algorithm. The results shown in Table 4.2 were a collection of the best performing solutions over multiple simulations of the GP algorithm, but there is no way to guarantee which (if any) of the best performing individuals will be reported by the GP algorithm on a single run. This makes reproducing the exact same solutions from the GP algorithm, without the help

of seeding, impossible.

Finally, the controls produced by our GP algorithm tended to be simple expressions. One reason for this is inherent in the way we chose to represent solutions to the optimization problem as functions, or expressions trees. Another approach to solve this optimization problem with a similar technique would be to represent a solution as a vector of tax rates of length $M * V * P * \ldots$, where M is the number of possible metabolism values, V is the number of possible vision values, P is the number of possible patchSugar values, and so on for every parameter in the Sugarscape ABM. The optimal vector of rates can then be found using a genetic algorithm. Genetic algorithms are very similar to genetic programming except that individuals are vectors of values rather than computer programs. This approach has the ability to produce solutions that perform very well for our optimization problem. However, by using this method, we would lose the ability to gain insight into the strategies that the solutions are employing (as suggested by function forms).

5.3 Reasons to Use This Method

The major appeal of using the GP approach to solve an optimization problem for an ABM is that it does not require an equation-based approximation model. The GP algorithm builds solutions that are compatible with the ABM and then uses the ABM to test the fitness of each solution. By eliminating the approximation model, we eliminate issues created by transforming solutions from the ABM to the approximation and vice versa. The process of performing these conversions can create result in solutions that are not truly optimal for either model.

Another benefit to using the GP approach is that it imposes very little constraints on the types of possible solutions. The GP algorithm can generate a solution that is any combination of values in the terminal and operator sets. The operator set

we used contained basic mathematical functions, but it can be extended to include complex functions at the agent level that are challenging to define mathematically but are easy to represent in the ABM. The terminal set we used contained many attributes of the ant that were all either Float objects or Boolean objects, but it can contain any object defined in the simulation. For example, in a more complex ABM where agents have attributes that are themselves or other agents, the terminal set can contain those agents and the operator set can contain functions that act upon agents.

Finally, since the controls generated by the GP algorithm are functions, or programs, they can provide both a qualitative and quantitative understanding of optimal strategies for control. For example, in our prototype Sugarscape ABM, the tick-based controls that emerged from the GP indicated that tax rates should increase over time. The sugar-based controls indicated that wealthier individuals should be taxed at higher rates whereas the metabolism-based controls indicated that low metabolism ants should be taxed less than high metabolism ants. These controls could be combined to inform adhoc construction of other, well performing, controls.

Bibliography

[1] Blickle, T., Thiele, L. (1995, July). A Mathematical Analysis of Tournament Selection. *In Proceedings of the 6th International Conference on Genetic Algorithms* (pp. 9-16). San Francisco, CA: Morgan Kaufmann Publishers.

[2] Brown, D. G., Robinson, D. T. (2006). Effects of heterogeneity in residential preferences on an agent-based model of urban sprawl. *Ecology and society, 11*(1), 46.

[3] Castella, J. C., Trung, T. N., Boissau, S. (2005). Participatory simulation of land-use changes in the northern mountains of Vietnam: the combined use of an agent-based model, a role-playing game, and a geographic information system. *Ecology and Society, 10*(1), 27.

[4] Christley S, Miller Neilan R., Oremland M., Salinas R., Lenhart, S. Optimal control of the Sugarscape ABM via a PDE model. In review.

[5] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, Christian Gagné (2012). DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research*, 13, 2171-2175.

[6] Epstein JM, Axtell R. (1996). *Growing artificial societies: social science from the bottom up.* Washington, DC: Brookings Institute.

[7] Federico P, Gross LJ, Lenhart S, Ryan D. (2013). Optimal Control in Individual-Based Models: Implications from Aggregated Methods. *The American Naturalist, 181*(1), 64-77.

[8] Hinkelmann F, Murrugarra D, Jarrah A, Laubenbacher R. (2001). A mathematical framework for agent based models of complex biological networks. *Bulletin of Mathematical Biology, 73*(7), 1583-1602.

[9] Johansson, R. (2008, December 7). Genetic Programming: Evolution of Mona Lisa. Retrieved from http://rogeralsing.com/2008/12/07/genetic-programming-evolution-of-mona-lisa/

[10] Koza, JR. (1990, November). Genetically breeding populations of computer programs to solve problems in artificial intelligence. *In Proceedings of the Second International Conference on Tools for AI* (pp. 819-827). Los Alamitos, CA: IEEE Computer Society Press.

[11] Mi, Q., Rivire, B., Clermont, G., Steed, D. L., Vodovotz, Y. (2007). Agent based model of inflammation and wound healing: insights into diabetic foot ulcer pathology and the role of transforming growth factor-β1. *Wound Repair and Regeneration, 15*(5), 671-682.

[12] Wilensky, U. (1999). NetLogo [Computer software]. Retrieved from http://ccl.northwestern.edu/netlogo/.

[13] Oremland M, Laubenbacher R. (2015). Optimal harvesting of a predator-prey agent-based model using difference equations. *Bulletin of Math Biology, 77*(3), 439-459.

[14] Oremland M, Laubenbacher, R. (2014). Using difference equations to find optimal tax structures on the SugarScape. *Journal of Economic Interaction and Coordination, 9*(2), 233-53.

[15] Railsback SF, Grimm V. (2012). *Agent-Based and Individual-Based Modeling*. Princeton, NJ: Princeton University Press.

[16] Sakamoto, E., Iba. H. (2001) Inferring a system of differential equations for a gene regulatory network by using genetic programming. *In Proceedings of the 2001 Congress on Evolutionary Computation* (pp. 720-726). doi:10.1109/CEC.2001.934462

[17] Michael Stillger and Myra Spiliopoulou. (1996). Genetic programming in database query optimization. *In Proceedings of the 1st annual conference on genetic programming* (pp. 388-393). Cambridge, MA: MIT Press.

[18] Wilensky U., Rand W. (2015). *An Introduction to Agent-Based Modeling*. Cambridge, MA: MIT Press.