Populating Code Cities with NPCs Representing Bugs

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

James M. Young

Approved by:

Major Professor: Clinton Jeffery, Ph.D.

Committee Members: Jim Alves-Foss, Ph.D.; Terence Soule, Ph.D.

Department Administrator: Terence Soule, Ph.D.

August 2022

Abstract

A code city is a visual representation of a code repository using aspects of the repository to build and arrange the city; code files can become buildings, and a directory of files can become a city block. Until now, most code cities have consisted of a static depiction of code repositories. This thesis lays the groundwork to populate a code city with dynamic entities that represent bugs reported in a code repository. These dynamic entities are controlled via an evolutionary algorithm that allows the bug to adapt to its environment as well as the existence of other entities. By measuring fitness as a metric of bug lifespan, the evolutionary algorithm optimizes time spent alive given a bug's surroundings.

Acknowledgements

Many thanks to my friends, family, and those in my academic career that have helped me along the way.

Table of Contents

Chapter 1: Introduction

Virtual environments are most likely known through their use in Massively Multiplayer Online (MMO) games, in which a world is simulated for players to interact with. Within this world, players are able to interact with both one another, as well as the environment, performing various tasks to progress through the game. Code cities take this idea of a virtual world and apply it to representing aspects of code repositories. Given the name, often the representations take the form of a city; code files become buildings, and directories of code files become city blocks, alongside many similar metaphors. These representations are created with the goal of making large code bases more understandable and easier to approach.

However, while these representations translate code into a familiar environment, it only captures the static elements of a city. While there are towering buildings and streets connecting them, there is no dynamic element at play. MMOs can have large city structures, but also a population of Non-Playable Characters (NPCs) inhabiting them. Depending on the context, such an environment can employ the use of friendly or hostile NPCs to better convey the purpose of the location in the overall narrative that the game is trying to tell. By taking this idea of combining dynamic and static elements and applying it to modeling code repositories within a virtual space, it is possible to convey more information than just a static representation.

This leads into the core problem that this thesis attempts to address: code cities are static, but a large portion of information could be conveyed by implementing a dynamic aspect to these environments. Cities are intrinsically populated, so it would make sense that code cities have the capability of being populated as well. Specifically, this work seeks to take the bug ticket tracker that accompanies a code repository and translate this information into NPCs that populate the city. Since the information being depicted are bugs that need to be fixed, the representation for these reports are depicted as hostile populations of monsters, similar to how a player might interact with monsters in an MMO. These groupings of monsters within the code city environment are intended to draw attention to problems within the code base, ideally being more engaging and encouraging a user/developer to take notice of them and fix the bug.

With the introduction of NPCs, some form of artificial intelligence (AI) must be employed to control them. Since the generated environment varies depending on the code base being represented, the AI driving these NPCs should also be flexible enough to work in these varied environments. For this purpose, an evolutionary algorithm (EA) is employed, paired with rudimentary actions, to control these created NPCs. Being an evolutionary algorithm, it adds a great level of flexibility to what environments an NPC can function in, without having to worry about specific code city configurations.

In order to test the viability of this implementation, the program BugSim (Bug Simulation) was created to simulate and observe how these bug report representations might behave and evolve within a code city. Since the ultimate goal of a bug ticket is being fixed and subsequently closed, the goal of the dynamic representations is to garner as much attention as possible. By representing the bug ticket as a population of NPCs, the easiest and most straight forward approach to gain attention is to have the population grow in size. Giving these NPCs the ability to explore their environment, gather resources, and hunt other bugs, the EA must decide what actions to take based on the input from the environment in order to grow its corresponding population. BugSim is able to observe and track how well configurations made to both the environment and EA perform in order to achieve this goal of being able to generate robust populations from bug tickets.

Within this thesis, chapter 2 will provide an exploration of literature focused around code cities, AI, and bug tracking will be discussed. Chapter 3 focuses on a tool made for this thesis called "BugScraper" which describes how and what information is obtained from code repositories. Following this, chapter 4 gives an in-depth description of how BugSim is constructed and operates, with chapter 5 going into great detail about how the NPCs, referred to as bugs, can operate as well as detailing the evolutionary algorithm used. Finally chapter 6 provides an evaluation of the system, and chapter 7 provides a summary of the evaluation, future work stemming from this thesis, and conclusions that can be drawn from this work.

Chapter 2: Related Work

2.1 Introduction

For the work done in this thesis, Software Cities, NPC's and Game AI, and Bug Tracker Analysis were considered the most relevant fields of research. As the core idea of this work stems directly from the work done within the field of software cities, the first section discuses the main ideas behind a code city representation as well as some of the various implementations. Since this work involves controlling non-playable characters spawned into the code city, the following section focuses on areas of research done with either artificial intelligence or evolutionary computation. Finally, since the dynamic entities being added to a code city are based off of bug reports, a section is dedicated to work done on improving bug tracking and reporting.

2.2 Software Cities

Code cities are a way of representing a code repository in virtual 3D space. One of the early implementations of a code city, Knight and Munro [11] sectioned off Java classes into districts, within which buildings represented methods inside the class. For each of the buildings, other visual means were used to provide more information. Lighter walls on the building indicated that the method was public, while dark indicated private. Additionally, the number of doors on the building were leveraged to show the number of parameters each method took. The overarching idea with all these representation choices is to take the large amount of information needed to understand a code base and attempt to reduce it down to something more visual and palatable to the human eye.

Much of the work in this area continues to find more ways to leverage these metaphors of translating code base information into visual aspects. Wettel and Lanza [18] represent classes as buildings, with the building's size being dependent on its member variables, as seen in Figure 2.1. The layout of these buildings were controlled via treemap, and the height and topology where the building sits represented package nesting. Panas et al. [15] represents code much in the same way as Knight and Munro [11], depicting methods as buildings but with the containing class represented as a foundation for these buildings. Panas et al. [15] also features a closer look at dependencies between source and header files. Other values are represented, such as the number of lines of code being depicted as the texture of the building, and the landscape on which all this sits on representing directories. Vincur et al. [17] create a representation that utilizes VR for a more interactive look at the code city. While classes represent buildings and stories represent methods, this representation utilizes color coding to depict additional metrics. Further reading on this topic can be found in Jeffery [10], which is a survey of code city

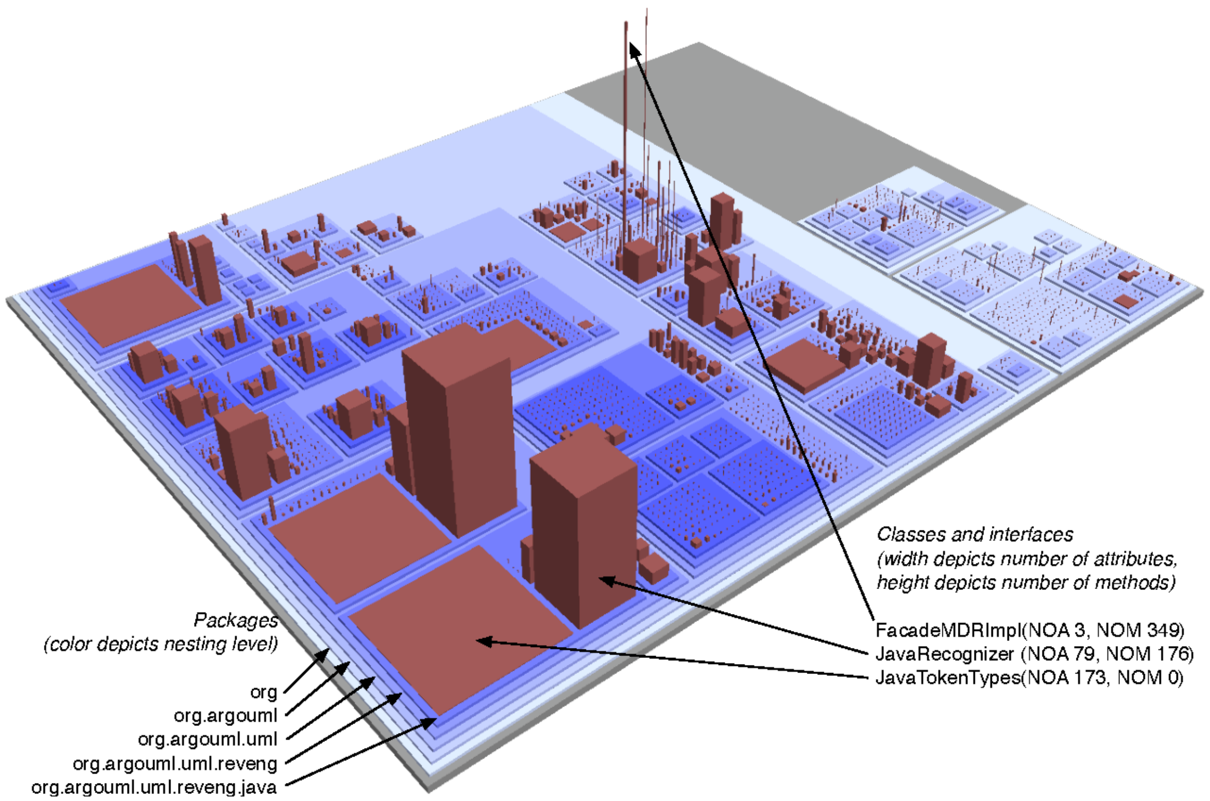

Classes and interfaces
(width depicts number of attributes,
height depicts number of methods)
FacadeMDRImpl(NOA 3, NOM 349)
JavaRecognizer (NOA 79, NOM 176)
JavaTokenTypes(NOA 173, NOM 0)
Packages
(color depicts nesting level)
org
org.argouml
org.argouml.uml
org.argouml.uml.reveng
org.argouml.uml.reveng.java

Figure 2.1: Depiction of software city from Wettel and Lanza [18].

representations.

While each of these works aim to improve the representation in their own ways through primarily static means, the work done in this thesis focuses entirely on the dynamic aspect. Using the static code city as the environment, entities are created using information found within the corresponding repository's bug tracker. These entities are then placed within a code city to act as a hostile agent to the user traversing the environment. By complementing the already existing code city and the metaphors that it utilizes, these dynamic entities offer an additional vector of conveying information.

2.3 NPC's and Game AI

In order to develop the control scheme behind the dynamic entities, various papers involving NPCs and game AI were looked at. As the initial foundation for the bug representation, Laramee [13] goes over creating a genetic algorithm to control a troll within a fantasy environment. By using the genetic algorithm, as well as keeping both the actions and goals of the troll simplistic, a satisfactory method of control was evolved to produce an optimal, hostile troll. In a similar vein, Che et al. [5] supports the idea that genetic algorithms are a better fit for a situation where over-fitting is a concern, in comparison to back-propagation learning algorithms even if the latter is faster. Cole et al. [6] takes existing first-person shooter games and creates bots controlled by a genetic algorithm. These bots, and by extension their corresponding genetic algorithm, aim to fine-tune their behavior rather than having preset values that a human would enter in while still achieving similar results. Jang and Cho [9] utilize an "influence map", which combine multiple layers of information found on a game map. This combined map of information is then fed into neural network in order to control an NPC optimally given the game state. Hong and Cho [7] also takes information from the environment and uses it as input for an NPCs genetic algorithm, but this time for use against another NPC.

In comparison to neural networks or hand-adjusting parameters, genetic algorithms provide a flexible solution to varied environments. Since a code city is generated differently based on the code repository being inspected, this flexibility provided an ideal approach for controlling the dynamic entities. Additionally, by utilizing information from the environment, the genetic algorithm could use this as input and evolve based on the environment. However, while a genetic algorithm worked for many of the papers mentioned, it proved to be too simple for the multi-population environment being produced. Instead, an evolutionary algorithm was employed, specifically an evolutionary decision tree, to transform environmental input into NPC actions.

Beyond the control mechanism behind the bug report representations, there is a motivation for the individual NPC's behavior to be somewhat believable— at least as believable insofar as a video game monster might behave. To achieve this, the NPC should be a nuisance within the code representation space with the capability to harass the user in some way. Various papers covering topics of NPC behavior AI involve ways of how to achieve or support this. Hussain and Vidaver [8] states that while evolutionary approaches have been used to to produce winning strategies and realistic group behavior patterns, this technique has seldom been used to develop rich NPC behaviors for player interaction. The paper goes onto state that an NPC must be able to plan, act, and react both in regards to the player as well as other NPCs. Mac Namee and Cunningham [14] discuss that with the large focus on graphics caused that field to improve, so can the state of the art in NPC's with that same level of focus. The paper further goes into the idea that NPCs should not be static, and have their own goals/directive that can allow for deeper immersion or be more believable to the player observing them. Bullen and Katchabaw [4] talk about the benefits of using GAs as a quicker and more elegant solution to creating character behaviors within modern video games, compared to traditional AI, in order to provide the player with a enjoyable experience. The paper also states that a GA can dramatically change based on the fitness function, as well as avoiding evolution into frustrating states for the player. Lankoski and Björk [12] cover qualities and traits of what makes NPCs believable, though it is focused on narrative aspects rather than behavior.

Again, since the required intelligence of the created NPC is not high, less focus was put into developing these aspects. However, interaction between same as well as different populations exist within this work. Created NPCs may decide to perform different actions depending on the state of other NPCs, either friendly or hostile. Additionally, these NPCs can engage in combat with other NPCs, in order to provide interaction opportunities outside of involving the players, with the goal of providing a more interesting experience.

2.4 Bug Tracker Analysis

Since this work involves taking information found in bug trackers, a survey of papers focused on the topic was also performed. The paper by Bissyandé et al. [3] performs a survey of projects found on GitHub while applying analysis with the goal of better understanding what an issue tracker provides to a software project. Understanding who submits issues, what is contained in these issue tickets, and the correlation between issue tickets and project success are all important factors since feedback is vital to the development of a project. Along the same line of thought, Bettenburg et al. [2] performs a survey across developers, which highlights the mismatch between what users supply and developers need in order to fix a bug. Pingclasai et al. [16] seeks to automatically classify bug reports through the use of topic modeling in order to save valuable time. Through the use of models employing ADTree, naive Bayes, and logistic regression, the models were able to able to successfully extract and classify bugs, with the naive Bayes approach being most efficient. In a similar vein, Banitaan and Alenezi [1] discuss the application of using metadata, rather than the actual content of the bug report, in order to guide bug triage. Through use of this metadata, there is a marked improvement of finding bug reports that require handling by an experienced developer, compared to the traditional machine learning approach. Ye et al. [19] offers an approach to bug reproduction through use of information found within the code base. By looking at methods within source code files, API descriptions, bug-fixing history, and code change history, a ranking is applied to each of the source code files as to how likely the issue might be found within it. This leveraging of domain knowledge approach allows for the offending source code file to be ranked within the top 10 recommendations for over 70% of the bug reports within the repository.

Due to the main focus of this work being the designing and testing of viability of a dynamic system within a code city, less focus is applied to the improvement of the bug fixing process. However, some level of support is still provided for future work on this project. BugScraper, described in chapter 3, grabs both the bug report description, as well as metadata from the report, with the intent of being later analyzed for use in the code city environment. In particular, a proposed area of future work would involve placing relevant bug reports nearby the corresponding source code building.

2.5 Summary

The work in this paper seeks to build primarily upon existing code city ideas by focusing on adding bug reports as a dynamic entity that exists within a code city. In order to achieve this, attention has been given to both how the entities will be controlled, as well as how the user might perceive them. Using an evolutionary approach, both of these topics are addressed, providing a way for the bug report entities to adapt to the generated environment. Finally, topics involving effective bug reporting, while not directly affecting development, have been considered for use in future work.

Chapter 3: BugScraper

3.1 Introduction

This chapter describes the utility program "bugscraper". Written in Python, the program is a relatively small tool (106 lines of code). When provided either a SourcgeForge or GitHub repository URL, the program will proceed to grab and analyze the contents of its bug/issue tracker. Once the relevant information has been analyzed, the program outputs a JSON file. The use of the JSON format provides a simple structured way to convey the desired information. Being widely used, the standardized data format also allows for widespread interoperability, such as making use of the JSON python module to easily read in the data.

Originally, this JSON file was intended to be used as a specification file for the creation of spawnpoints within a code city environment. Since the decision to use a simulation rather than an implementation to observe A.I. behavior was made partway through the development of the thesis, the exact information of a bug ticket is no longer used. However, test cases involving the relevant fields found in the JSON file are simulated, with more information being available in Chapter 6.

The following sections describe the behavior of BugScraper. The entire program can be found in Listing **??**.

3.2 Input

The program accepts a command line argument of either a SourceForge or GitHub repository URL. Special care must be taken to ensure the link is for the "homepage" of the repository, and not, for example, the address of the bug tracker (`https://sourceforge.net/projects/unicon/` vs. `https://sourceforge.net/p/unicon/bugs/`). A future refinement of this application may allow for any URL (relevant to the repository) to work.

3.3 Bug Scraping

Once a valid URL has been provided, the program will analyze the link and run either the SourceForge or GitHub bug scraping sub-functions.

The targeted information fields being scraped are as follows:

- The bug/ticket number
- The current status (open/closed)
- Who reported the bug

- A summary of the bug/issue
- A more detailed description of the bug/issue
- When the bug/ticket was created
- When the bug/ticket was last modified
- The priority (currently only available in SourceForge)

Fields that are shared between both platforms are targeted, with the single exception being priority, which is only found on SourceForge. Beyond this, out of the available data found in a bug ticket, these specific fields were chosen due to how they might represent the overall importance of the ticket. For example, an older ticket that has remained open could be considered of higher importance than a ticket that has been recently opened. The summary and description fields provide a more in-depth look at the state of the bug should a deeper inspection be warranted.

The overall goal for these fields of data being available within the bug report is to parse and subsequently convert the information that reflects the bug in a meaningful way. For example, the longer a bug ticket has existed as open, the represented bug may be more powerful within its environment. Another proposed, but not implemented, way of utilizing these fields would be to build a system that scans the description of the bug, checks it against the code buildings found within the city representation, and places it near relevant buildings (such as a bug within a particular source file).

3.3.1 SourceForge

SourceForge provides a convenient API to connect to and access a repository's bug tracker. For example, the URL `https://sourceforge.net/projects/unicon/` has a corresponding URL `https://sourceforge.net/rest/p/unicon/bugs` where all information relevant to each ticket are stored in JSON format.

After connecting to the API, the program will enter a loop that will grab all available bugs. The loop is required as the API only allows access up to 100 bug entries at a time, so each iteration of the loop grabs the information of up to 100 bugs. Additionally, this is a pre-processing step, as this will only get the total number and addresses of the bugs (as each ticket is located at its own URL), but not the actual contents that we are looking for. Once a Python list is filled with the bug tickets, then a second loop begins in which the specific addresses where the bug's information is stored are accessed. At this point, the program now has a list where each entry represents a bug/ticket, with relevant fields being accessed through a table. The program then returns the list to be written out to JSON.

3.3.2 GitHub

Access to GitHub is more tricky than with SourceForge. To prevent a potential denial of service, GitHub limits the number of requests from an un-authenticated source to 60 requests per hour.

One benefit of GitHub's API however is that it allows for more specific requests, as well as presenting the info of multiple bugs per page request. With the number-of-bugs-per-request being 100, this greatly cuts down the total number of requests needed. So unless the repository has over 6000 (100 bugs * 60 requests per hour) bug entries, the application is able to run to completion. If a greater number of bugs are needed to be read in, either Bugscraper should track what has been read and what needs to be read, or the tool should be refined to use an authenticated client. At this point, the program will iterate through the list of bug entries, grab and store the relevant fields of the JSON in a table, and store the table in a list. Once finished, the list will be returned for final output.

3.4 Output to JSON

At this point in the program, a Python list holds the trimmed down content of each bug. This information is then converted from a Python format into a JSON string, that is then written out to "bug_tickets.json" to be used in the creation of bugs in the CVE simulation.

Listing 3.1 is an entry found in a resultant bug_tickets.json. In particular, this is ticket 119 from the Unicon SourceForge repository.

Listing 3.1: A segment of the resultant JSON file.

```
...
{
   "summary": "wron g IXBIN path",
   "priority": "5",
   "created_date": "2012-01-22 06:30:30",
   "mod_date": "2012-01-22 06:30:30",
   "ticket_num": 119,
   "status": "open",
   "reported_by": "are_muc",
   "description": "when building UNICON it adds the wrong IXBIN path to
       the built executables; it is the path where UNICON is built
      \\\\\\\\(e.g. /home/are\\\\\\\\_muc/unicon/bin/iconx\\\\\\\\) but
       it should be the path where it will be intsalled \\\\\\\\(dest
      in the Makefile e.g. /opt/unicon/bin/iconx\\\\\\\\)"
},
...
```

3.5 Current Issues

Due to the way both SourceForge and GitHub store ticket information, a great deal of additional newline and carriage return escape characters run rampant within the description field of the bug ticket. While easily removable, a problem arises if some of those newline/carriage return characters were intended to help format the bug ticket. As of this point, the raw text is left in as is.

3.6 Summary

BugScraper provides the link between a code repository and the spawnpoints used to create the active entities within a code city. By reading code repository information and transforming it into a JSON file, a code city program should be able to easily read in this information to produce spawnpoints. From these spawnpoints, populations of monster-like entities will be produced to then interact with the environment. For the work done in this thesis, tests based on some of the fields aggregated by bugscraper will be conducted within the environment, in order to better test the control method of an evolutionary algorithm.

Chapter 4: BugSim

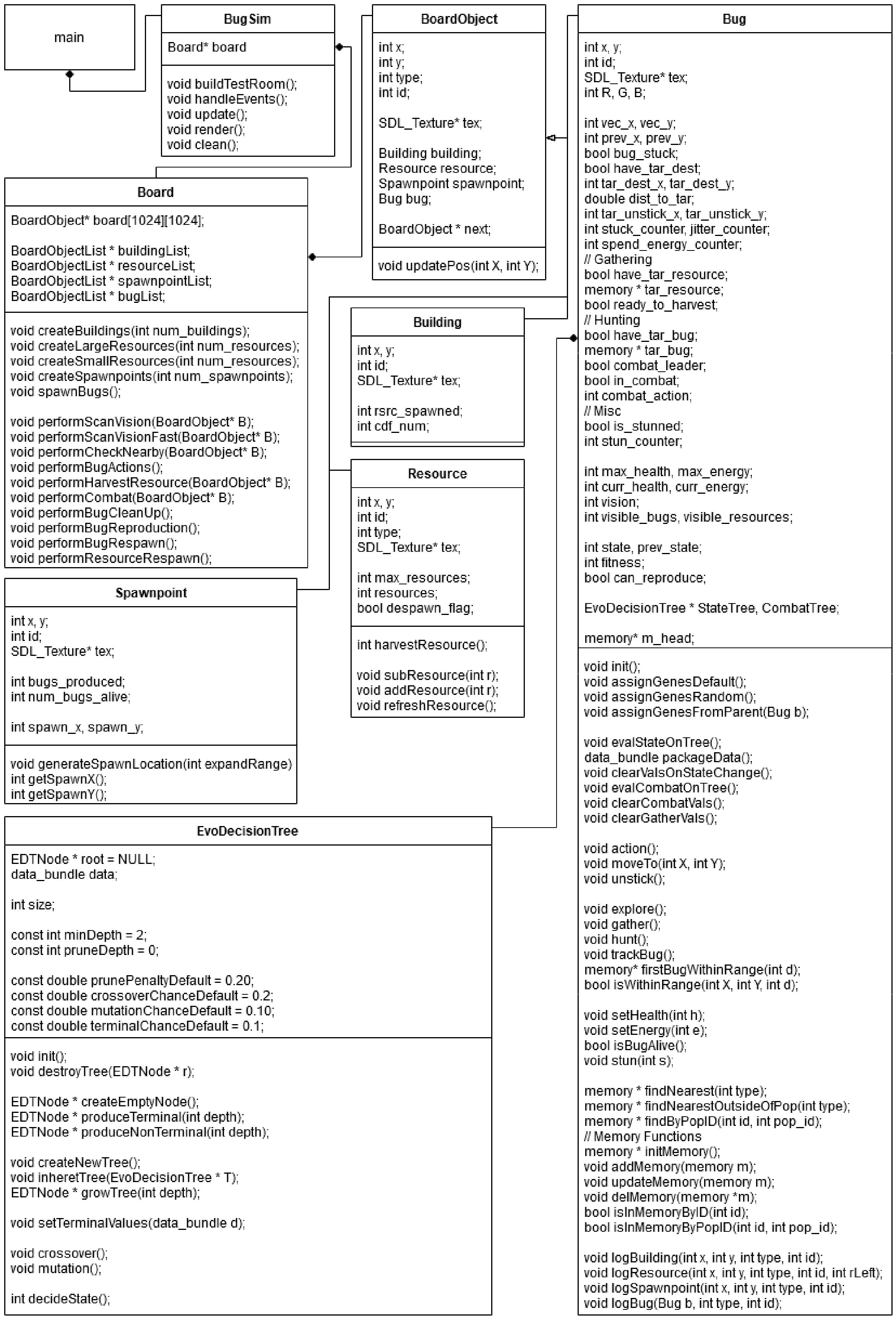
4.1 Introduction

BugSim, short for Bug Simulation, is the main focus of this thesis. Written in C++, this program was developed as a way of testing the effectiveness of bugs controlled by evolutionary code within a hypothetical code city. Effectiveness in this case refers to the ability of the different populations being able to grow and evolve to suit their environment; from being able to gather resources to hunting other populations in order to bolster their own. By increasing the size of a population, so too increases the chance of a user exploring the environment to come across said population. Since bug reports are converted into unique spawnpoints (and by extension populations) within this system, a larger population will draw more attention to a specific spawnpoint/bug report.

Within the simulation itself is a 2D representation of a code city. While code cities are represented in a 3D format, a 2D representation was chosen for this thesis as it provided a better format to observe the behavior of populations, removes computation complexity, and that information provided in a 3D environment, such as building height, had no practical use for bug interaction. Once the simulation has been built by placing buildings, resources, and spawnpoints, bugs are spawned near their corresponding spawnpoints, and then live their life within the environment. As fitness is based entirely on how long they are able to survive, bugs' primary goal is to properly maintain both their health and energy. To propagate genetics fit for their environment, bugs who are able to consistently amass energy reproduce, expending energy and creating a new generation of bugs that inherit the parents' genes. By gathering resources and hunting other bugs, a bug is able to recover health and energy lost through combat, exploration, or reproducing. The following section will go more into detail about each aspect of the BugSim. Additionally, refer to 4.1 for a UML diagram describing BugSim.

4.2 Environment

The bug simulation runs in an environment based on the output generated by a program called Gencity. Gencity was written by C. Jeffery and modified for this thesis. Within Gencity, buildings are generated based on the file structure of a code repository—the modified version of this program creates additional structures. After the buildings have been placed, the modified gencity goes down a list containing each building and decides at a 50% chance whether to place a resource or not. Each resource is placed in such a way that it should be within a set distance to the buildings—as a result an area with more buildings can be expected to have more resources. Once resources have been placed in the world, spawn points are then handled. Currently, the spawnpoints are placed into the world at



```
main
```

```
BugSim
Board* board
void buildTestRoom();
void handleEvents();
void update();
void render();
void clean();
```

```
BoardObject
int x;
int y;
int type;
int id;

SDL_Texture* tex;

Building building;
Resource resource;
Spawnpoint spawnpoint;
Bug bug;

BoardObject * next;
void updatePos(int X, int Y);
```

```
Bug
int x, y;
int id;
SDL_Texture* tex;
int R, G, B;

int vec_x, vec_y;
int prev_x, prev_y;
bool bug_stuck;
bool have_tar_dest;
int tar_dest_x, tar_dest_y;
double dist_to_tar;
int tar_unstick_x, tar_unstick_y;
int stuck_counter, jitter_counter;
int spend_energy_counter;
// Gathering
bool have_tar_resource;
memory * tar_resource;
bool ready_to_harvest;
// Hunting
bool have_tar_bug;
memory * tar_bug;
bool combat_leader;
bool in_combat;
int combat_action;
// Misc
bool is_stunned;
int stun_counter;

int max_health, max_energy;
int curr_health, curr_energy;
int vision;
int visible_bugs, visible_resources;

int state, prev_state;
int fitness;
bool can_reproduce;

EvoDecisionTree * StateTree, CombatTree;

memory* m_head;
void init();
void assignGenesDefault();
void assignGenesRandom();
void assignGenesFromParent(Bug b);

void evalStateOnTree();
data_bundle packageData();
void clearValsOnStateChange();
void evalCombatOnTree();
void clearCombatVals();
void clearGatherVals();

void action();
void moveTo(int X, int Y);
void unstick();

void explore();
void gather();
void hunt();
void trackBug();
memory* firstBugWithinRange(int d);
bool isWithinRange(int X, int Y, int d);

void setHealth(int h);
void setEnergy(int e);
bool isBugAlive();
void stun(int s);

memory * findNearest(int type);
memory * findNearestOutsideOfPop(int type);
memory * findByPopID(int id, int pop_id);
// Memory Functions
memory * initMemory();
void addMemory(memory m);
void updateMemory(memory m);
void delMemory(memory *m);
bool isInMemoryByID(int id);
bool isInMemoryByPopID(int id, int pop_id);

void logBuilding(int x, int y, int type, int id);
void logResource(int x, int y, int type, int id, int rLeft);
void logSpawnpoint(int x, int y, int type, int id);
void logBug(Bug b, int type, int id);
```

```
Board
BoardObject* board[1024][1024];

BoardObjectList * buildingList;
BoardObjectList * resourceList;
BoardObjectList * spawnpointList;
BoardObjectList * bugList;
void createBuildings(int num_buildings);
void createLargeResources(int num_resources);
void createSmallResources(int num_resources);
void createSpawnpoints(int num_spawnpoints);
void spawnBugs();

void performScanVision(BoardObject* B);
void performScanVisionFast(BoardObject* B);
void performCheckNearby(BoardObject* B);
void performBugActions();
void performHarvestResource(BoardObject* B);
void performCombat(BoardObject* B);
void performBugCleanUp();
void performBugReproduction();
void performBugRespawn();
void performResourceRespawn();
```

```
Building
int x, y;
int id;
SDL_Texture* tex;

int rsrc_spawned;
int cdf_num;
```

```
Resource
int x, y;
int id;
int type;
SDL_Texture* tex;

int max_resources;
int resources;
bool despawn_flag;
int harvestResource();

void subResource(int r);
void addResource(int r);
void refreshResource();
```

```
Spawnpoint
int x, y;
int id;
SDL_Texture* tex;

int bugs_produced;
int num_bugs_alive;

int spawn_x, spawn_y;
void generateSpawnLocation(int expandRange)
int getSpawnX();
int getSpawnY();
```

```
EvoDecisionTree
EDTNode * root = NULL;
data_bundle data;

int size;

const int minDepth = 2;
const int pruneDepth = 0;

const double prunePenaltyDefault = 0.20;
const double crossoverChanceDefault = 0.2;
const double mutationChanceDefault = 0.10;
const double terminalChanceDefault = 0.1;
void init();
void destroyTree(EDTNode * r);

EDTNode * createEmptyNode();
EDTNode * produceTerminal(int depth);
EDTNode * produceNonTerminal(int depth);

void createNewTree();
void inheretTree(EvoDecisionTree * T);
EDTNode * growTree(int depth);

void setTerminalValues(data_bundle d);

void crossover();
void mutation();

int decideState();
```

Figure 4.1: UML diagram describing BugSim.

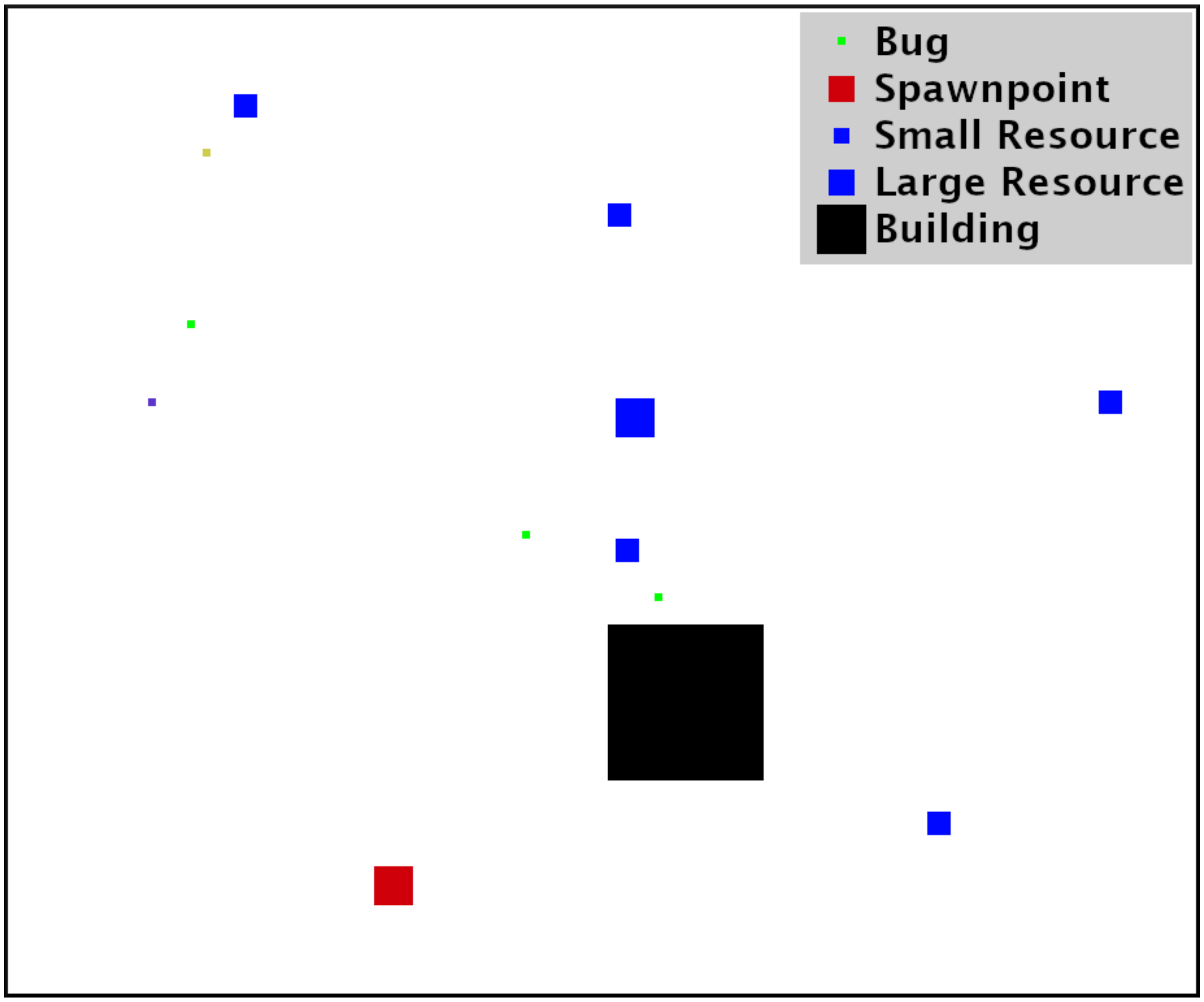


Figure 4.2: Screenshot of BugSim environment with legend.

random, but future work, such as parsing code files and placing related bugs next to their respective building is anticipated. During the placement of these resources and spawnpoints, measures are taken to ensure that the structures do not overlap.

For the simulation, a smaller environment is produced. Buildings are randomly placed within the environment, rather than being placed according to a file structure. There are a few other minor configuration differences, such as number of structures, number of resources to spawn, and number of bugs. Refer to Figure 4.2 to see visual representation of the environment.

4.3 Buildings

Buildings are a representation of code found within a code repository, and are intended to be organized based on their directory. Within BugSim, 10 buildings are randomly placed on the map. In the 3D representation, height is used to represent the size of the respective file. Due to the top-down perspective of the 2D simulation, the visual aspect of height is not conveyed within BugSim, but the information can still be stored within the object during runtime.

4.4 Spawnpoints

Spawnpoints control where bugs are spawned; each spawnpoint represents a unique population. After the initial population of the simulation has been created, a spawnpoint will produce a bug nearby if the population meets one of two conditions: the population goes below a certain amount, or a bug is capable of reproducing. When the population dips below the set threshold value, which is 5 in the current configuration of BugSim, a bug is spawned with randomly assigned genes—the purpose being to introduce new genetics that hopefully are able to perform well in their surroundings. When a bug is capable of reproducing, the bug will asexually spawn a new bug with their genetics being assigned based on the parent. In order to evolve, the genes are then subjected to crossover and mutation, with the goal of becoming more fit to their surroundings.

4.5 Resources

Resources are objects within the simulation which bugs can harvest to restore energy. Resources come in two sizes- small and large. Small resources dot the map and can only replenish a small amount of energy to a bug. The functional purpose of these objects are to provide enough energy that a bug may survive some level of exploration, but not necessarily create offspring. In the current configuration of BugSim, there are 100 small resource nodes that are randomly created across the environment, each containing 10 units of energy. When one small resource is exhausted, another is randomly spawned.

Large resources contain more energy, but are only spawned near code buildings. Their purpose is to act as an incentive for bugs to move towards the points of interest that are code buildings. As these larger resources provide a much higher amount of energy, bugs that seek these resources out are much more likely to reproduce. In the current configuration of BugSim, there are 10 large resource nodes that are randomly spawned nearby code buildings, each containing 100 units of energy. When a large resource is exhausted, another is spawned nearby a code building. As a way to model exhausting the resources in a specific area, each building has an internal counter that tracks how many resources have been spawned nearby. Increasing a specific building's counter (compared to other buildings) will in turn lower the weighted chance that a new resource will spawn there in the future.

4.6 Bugs

Bugs are the active agent within the simulation. Bugs interact with the simulation environment in three major ways: traversing the environment, gathering at a resource, or hunting other bugs. The longer the bug is able to survive within the environment, the higher fitness value they achieve. In addition to simply being around longer, a higher fitness bug is typically able to produce more offspring, and thus be more of a presence within the environment. More detail on bugs can be found in chapter 5.

4.7 Simulation Loop

Every iteration of the simulation loop consists of 6 major steps, each with numerous sub-steps, detailed as follows.

1. In parallel, iterate through all alive bugs and check:
    - (a) All points of interest in vision
        - i. Iterate through all buildings in simulation to ensure target destination is not inside building and unreachable
        - ii. Iterate through all resources in simulation, log each resource within vision range and increment visible resources counter
        - iii. Iterate through all spawnpoints in simulation to ensure target destination is not inside building and unreachable
        - iv. Iterate through all bugs in simulation, log each bug within vision range and increment visible bugs counter
    - (b) Iterate through memory to see if points of interest are still nearby where they were originally seen
        - i. If memory entry is bug, check if within nearby range
            - A. If location where bug was last seen is empty, remove entry from memory
            - B. If location where bug was last seen is occupied by other entity, remove entry from memory
            - C. If entry removed was hunting target, clear hunting target
        - ii. if memory entry is resource, check if within nearby range
            - A. If location where resource was last seen is empty, remove entry from memory
            - B. If location where bug was last seen is occupied by other entity, remove entry from memory
            - C. if entry removed was gathering target, clear gathering target
    - (c) Perform bug upkeep and decide on action(s) to take
        - i. Increase fitness by 1 point

ii. Calculate current energy percent, if higher than 75%, raise reproduce flag

iii. Evaluate evolutionary decision tree to determine state

iv. If state is Explore

  A. Set a target destination if not already set

v. If state is Gather

  A. If bug has no target resource, set a target resource

  B. If bug has a target resource, check distance to target resource

  C. If not in range, set target destination to target resource

  D. If in range, raise ready to harvest flag

vi. If state is Hunt

  A. If the bug has no target bug, set a target bug

  B. If the bug has a target bug, check distance to target bug

  C. If not in range, set target destination to target bug

  D. If in range, raise combat leader flag

vii. If state changed, clear all variables related to state (target resource, target bug, etc.)

viii. If stuck flag is raised, attempt to un-stick

ix. Determine direction to move, dependent on target destination

x. If this is the 20th action taken, reduce energy by 1, reset actions taken counter

xi. Increase actions taken counter

xii. Check to see if bug is in "jitter" state, and if so, raise stuck flag

2. Execute the bugs' decided action

  (a) Move bug to its new location, if possible

  (b) If unable to move, raise bug's stuck flag

  (c) If bug's harvest flag is raised, perform harvest resource

    i. Calculate and subtract units from target resource

    ii. If resources remaining are less than or equal to 0, raise despawn flag

    iii. Add units to bug's energy pool

  (d) If bug's combat leader flag is raised, perform combat

    i. Raise leader bug and target bug in-combat flag

    ii. Bug 1 (leader) evaluate evolutionary decision tree to determine combat action

        iii. Bug 2 (target) evaluate evolutionary decision tree to determine combat action
        iv. Determine outcome of combat based on both actions (refer to Table 5.2)
        v. Resolve combat by checking if both, either, or neither bug died or fled

3. Clean up any bugs that died this iteration
    - (a) Iterate through list of bugs
    - (b) If health is less than or equal to 0, bug is considered dead
    - (c) Decrement bug's population by 1
    - (d) Remove bug from the environment
    - (e) Remove bug from list of bugs
4. Check if any bugs can reproduce
    - (a) Iterate through list of bugs
    - (b) If reproduce flag is raise, begin reproduction
        - i. Iterate through list of spawnpoints based on corresponding population
        - ii. Find empty location next to spawnpoint to place new bug
        - iii. Create and initialize new bug
        - iv. Assign genes based on parent
        - v. Set parent and child bug energy to half of parent energy
        - vi. Increase bug's population by 1
        - vii. Add bug to list of bugs
        - viii. Lower parent bug reproduce flag
5. Handle bug respawning
    - (a) Iterate through all spawnpoints
    - (b) If spawnpoint's corresponding population is less than 5, begin respawn
        - i. Find empty location next to spawnpoint
        - ii. Create and initialize new bug
        - iii. Assign genes at random
        - iv. Increase bug's population by 1
        - v. Add bug to list of bugs
        - vi. Loop if corresponding population is still less than 5

6. Handle resource respawning
   (a) Iterate through all resources
   (b) If despawn flag is raised and resource is large, begin large resource respawn
      i. Remove resource from current location
      ii. Using a cumulative distribution function, randomly select a building
      iii. Find an empty location next to selected building
      iv. Set resource units to max and lower despawn flag
      v. Place resource in empty location
   (c) if despawn flag is raised and resource is small, begin small resource respawn
      i. Remove resource from current location
      ii. Randomly select an empty location in the environment
      iii. Set resource units to max and lower despawn flag
      iv. Place resource in empty location

After these steps have been performed, the result is then rendered to screen, and the loop continues.

4.8 Summary

The BugSim environment allows rapid testing of the proposed system to be used in implementing a dynamic element to a code city. Said system involves adding resources, spawnpoints, and bugs to facilitate the dynamic aspect that is populations corresponding to bug reports. Additionally, through use of this program, performance of populations can be observed given different configurations. This has allowed the development of a evolutionary algorithm used to control the dynamic entities spawned from spawnpoints which will be explored in the next chapter.

Chapter 5: Bugs

5.1 Introduction

The initial idea being inspired from monsters found in video games, bugs represent a potentially hostile agent to the user exploring the environment. Early on in development, a bug in the environment corresponded directly to a bug ticket found within the code repository. However, this configuration of the system resulted in dead environments, so the system was changed so that a single bug ticket now represents a population. This configuration allowed for much more interesting environments, promoting more interaction with the environment, as well as between other populations. Defined in this chapter are the individual agents that make up these populations, in particular to what extent they are able to act and how they are able to "choose" what actions they will take.

5.2 Bugs

The bugs spawned from bug reports are designed to act in a manner similar to monster-bugs found in video games so that they may be interacted with in a similar manner within a code city environment. Bugs will spawn from a set location, and begin interacting with the environment with decisions and actions based off of their genetic information. Bugs that are able to exist for extended periods of time are able to reproduce and create more of their population, thus increasing their population's presence within the environment.

5.3 Genetic Information

Using the Evolving a Perfect Troll Laramee [13] paper as a basis for the bugs, the genetic makeup consisted of three values in the range of 0.0 to 1.0, representing a bug's aggression, hunger, and fear. The idea behind this was that the higher these values, the more often the bug would perform actions in that vein. For example, a bug high in hunger and aggression would more often choose to hunt other bugs, while a bug with a high fear value would avoid hunting and focus on gathering from resources. On top of this, these values would control actions taken in combat following the same pattern; bugs high in aggression would attack more, while bugs high in fear would try to flee. With each new iteration of bug, these three values would mutate, in the hopes of forming a set of genes that would survive the longest within the environment.

While this approach worked in the environment proposed in Laramee [13], it did not lead to much success in BugSim. As implemented above, no evolution was taking place as a result of the environment being both large and dynamic. A typical result would be a bug lineage evolving to hunt, only for there to be no more bugs nearby and then starving to death. Similarly, bugs that evolved to gather would run out of immediate resources and meet a similar fate. All of this combined with the configuration of the system only allowing one bug per spawnpoint resulted in non-functioning bugs.

Two major changes were made in response to this problem: unlimited population size for each spawnpoint, and using an evolutionary decision tree to represent the genetic code. Having unlimited population size helped add more interaction within the simulation, such as allowing hunting bugs more targets to gain energy and reproduce, as well as vastly improving genetic transference to new generations. Using the evolutionary decision tree allowed for more dynamic choices depending on the bugs' immediate surroundings, such as deciding to gather resources if there are large amounts nearby, to hunting bugs if low health targets have been spotted.

Representing the bugs' predisposition to certain actions and behaviors are the bugs' genetic information—an evolutionary decision tree. Using this, a bug decides what action it will take given the state of itself and its surroundings. There are possible outcomes for every evaluation: explore, gather, or hunt, the specifics of each outcome are detailed later in this section. Additionally, bugs contain another evolutionary decision tree that is used to determine its actions within combat. In a similar manner, there are three outcomes for this as well: flee, defend, attack, also explained in further detail in Table 5.2 and the accompanying Figure 5.1.

Each bug also has a simplistic genetic algorithm that controls the color of how it is represented visually. As this is purely visual, it has no importance on the actual performance of the bug.

5.4 Evolutionary Decision Tree

Evaluation of the decision tree involves starting at the root of the tree, and navigating downwards. At each node, if it has reached a terminal, it will return the terminal, otherwise it will evaluate the current, non-terminal node and go either left or right down the tree. Each non-terminal consists of a: variable, non-terminal operator, and a random value between 0 and 100. A simple boolean check is performed of the variable against the random value, progressing down the left side on a "true" evaluation, and right on "false". The variable's values are based on the current state of the bug in the environment at that step.

5.5 General Behavior

Bugs produced from a spawnpoint are designed to behave in the likeness of a monster found within in a video game. They are capable of traversing about the environment and interacting with it, both in aggressive and non-aggressive ways. Within this environment, the primary goal of a bug is to survive, and when possible, produce offspring. By growing in numbers, the bug (and by extension its corresponding

Table 5.1: Overview of evolutionary decision tree configuration.

Terminal Set	Explore, Gather, Hunt OR Flee, Defend, Attack
Non-Terminal Set	VARIABLE COMPARISON CONSTANT
Variable Pool Set	Current Health, Current Energy, Visible Bugs, Visible Resources, Target Bug Health, Target Bug Energy, Target Resources Left, Distance to Target
Comparison Pool Set	Less Than, Greater Than
Constant Pool Set	Random value between 0-100
Terminal Chance	10%
Non-Terminal Chance	90%
Minimum Depth	2
Prune Depth	0
Prune Penalty	20% increased chance to become a terminal past prune depth
Crossover Chance	20%
Crossover Method	Select 2 random nodes
Mutation Chance	max(1%, (10 - generation)%)
Mutation Method	Perform chance on every node
Mutation Style	Randomly change node into terminal or non-terminal

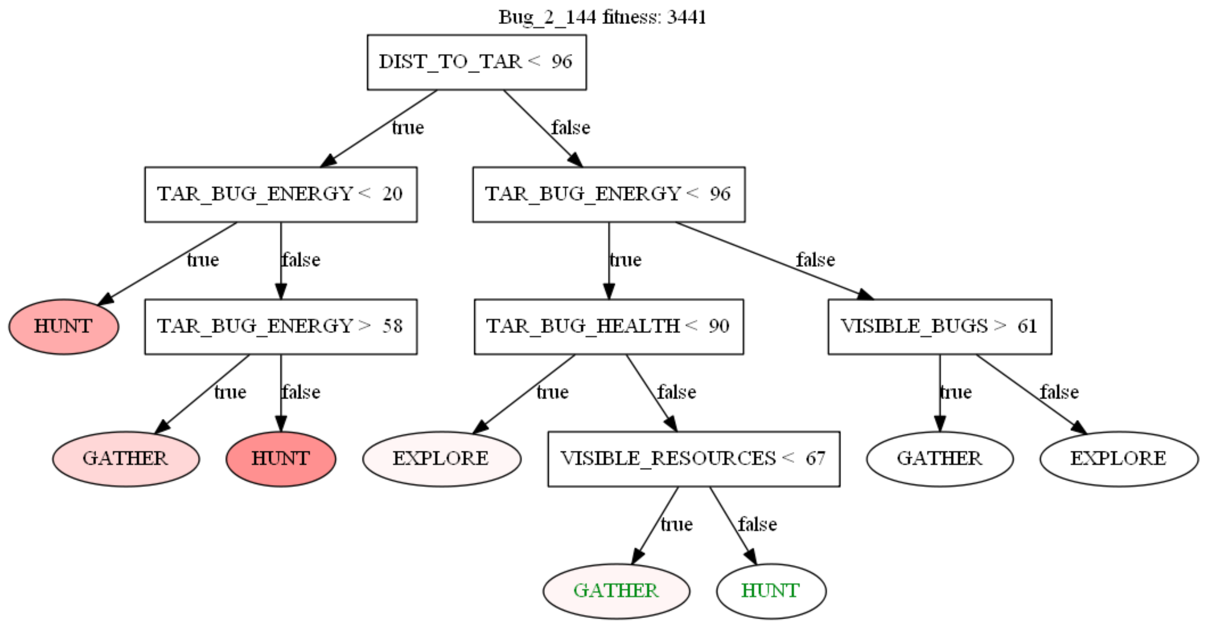


Figure 5.1: Example configuration of evolutionary decision tree.

spawnpoint produced by a bug ticket) become more noticeable compared to other bug populations who are unable to succeed. A bug's basic traits and behaviors are explained within this section.

5.5.1 Health

Health is the value that keeps the bug alive—so long as health is not 0, the bug continues to act within the environment. Health is depleted in two different ways: combat and running out of energy.

While in combat, depending on the actions of both bugs, each bug can lose an amount of health. If a bug runs out of energy, every unit of energy depleted will instead subtract from the health of the bug. The only way a bug is able to restore health is through successfully hunting and defeating another bug.

5.5.2 Energy

Energy is used as a means to "safely" do an action (i.e. not using health) as well as a metric to determine reproduction. Energy is depleted by 1 unit every 20 ticks a bug is alive. A larger amount of energy is also depleted upon combat actions, as defined in Table 5.2, as well as during reproduction where the bug will give half of its energy to produce offspring. Bugs can restore their energy by either finding and harvesting a resource, or by successfully hunting and defeating another bug.

5.5.3 Sight

Before every action, a bug will scan its surroundings and remember anything of interest. This includes: buildings, spawnpoints, resources, and other bugs. As the bug is limited in the range in which it can detect things, exploration is usually required to improve information gain. Objects of interest are stored as a memory of how the bug most recently saw the object in question, meaning that if said object leaves its range of vision it needs to be re-encountered to be properly updated.

5.5.4 Memory

Each bug has a memory bank in which it can store the locations of interest, specifically resources and other bugs. For resources, the location is stored, alongside how many units the resource has left. For bugs, the location, current health, and current energy of the bug are all stored.

5.5.5 Stuck

Due to the simplistic nature of the bugs' movement, a bug can occasionally get stuck on objects. When the bug has not moved from its location after 4 actions when it should be moving, the bug will attempt to get unstuck. Additionally, this unsticking will also occur if a bug "jitters," or moves back and forth across two locations. The bug will choose a random direction to move 10 units and then attempt to resume whatever task it was performing previously.

5.5.6 Eval State

After gathering information of its surroundings, the bug will perform an evaluation of what state it should be in. By feeding the information it gathered to its evolutionary decision tree, it will come to a decision and set its state.

5.5.7 Movement

Bugs are capable of movement in both cardinal and inter-cardinal directions. To perform a move, the bug first consults the location of where the bug desires to go. Then a calculation of its current location and its target location is performed to produce a degree value, which is then used to produce the 1 of 8 possible directions to move it closest to its goal.

5.6 States

States determine what a bug will do within the environment. There are 3 states a bug can be in: exploration, gathering, or hunting. Depending on which state that a bug is in will change their movement and objectives within the environment. States are dependent on the input the bug receives from its surroundings, so in an ideal case, the bug will change to the most advantageous state to ensure its survival as well as being able to produce more offspring.

5.6.1 Exploration

Exploration is the state a bug primarily uses to traverse through the environment. In this state, the bug will wander around the environment by means of choosing a random point at the edge of its vision, moving towards it, and upon reaching the point, repeat the process. By exploring the map, the bug can encounter resources and other bugs, both of which will be stored in its memory should they enter a state in which that information is relevant.

5.6.2 Gathering

Gathering is a more passive state of regaining energy. Bugs in this state will consult their memory to find the closest resource to their location and begin to move towards where they remember it being. Upon arrival at the resource, the bug will stop moving and begin to harvest from the resource, stopping until the resource is depleted or the bug changes state. If a bug is unable to come up with a location to move towards, due to not having any resources logged within their memory, the bug will remain stationary.

5.6.3 Hunting

Hunting is an aggressive state of the bug to replenish health and energy. A bug in the hunting state will consult its memory to the find the closest location of a bug outside of its population and proceed to move towards it. Upon reaching the targeted bug, the hunting bug will initiate combat. If a bug is unable to come up with a location of a bug to hunt as a result of not having any logged within their memory, they will not move in any direction.

Table 5.2: Potential combat outcomes.

Outcome	Bug 1 (Hunter) Action	Bug 2 (Target) Action	Combat Outcome
1	Attack	Attack	Bug 1 loses 10 health Bug 2 loses 30 health Each bug loses 10 energy
2	Attack	Defend	Bug 1 loses 10 energy Bug 2 loses 20 health
3	Attack	Flee	Bug 1 loses 10 energy Bug 2 loses 30 health Bug 2 attempts to flee
4	Defend	Attack	Bug 2 loses 10 energy
5	Defend	Defend	Nothing happens
6	Defend	Flee	Bug 2 attempts to flee with a bonus
7	Flee	Attack	Bug1 loses 10 health Bug 1 attempts to flee Bug 2 loses 10 energy
8	Flee	Defend	Bug1 attempts to flee with a bonus
9	Flee	Flee	Both bugs successfully flee

5.7 Combat

When in the hunting state, once a bug reaches its target bug, combat will begin. During each round of combat, both of the bugs will make a decision to either attack, defend, or attempt to flee—based on the evaluation of their combat evolutionary decision tree. Once the two bugs decide on an action to take, the outcome of the round is then resolved. There are 9 distinct outcomes, based on the combinations of actions taken by each bug, depicted below.

5.7.1 Ending Combat

The two ways combat can end is either with one or both of the bugs deciding to flee, or one bug defeats the other. Fleeing is dependent on the difference between each bug's energy. If the fleeing bug has more energy than their opponent, the fleeing bug has a higher chance to successfully escape. Additionally, if fleeing from the opponent, a bonus is applied to the chance of a successful escape.

On the other hand, whenever a bug is reduced below 0 health, they are considered to have died and combat ends. The winning bug gets 50 units of health and energy restored, having 'consumed' their opponent. Since multiple bugs can be in combat with the same target, only the bug who dealt the killing blow is granted the health and energy.

5.8 Summary

Bugs are the main actors within BugSim, being individuals that make up unique populations. Their purpose is to represent, similar to monsters found within video games, a potentially hostile agent that can oppose the user exploring the environment. Additionally, their attributes are similar to their video game counterpart, providing a similar experience to those familiar with the medium. While the states that they can be in has been kept simple, the use of an evolutionary decision tree allows for much more complex decision making when determining which state an individual bug will be in. Additionally, the use of evolutionary decision tree allows both the evolving and the propagation of good "decision-making" genes, such that bug populations have the chance to adapt to varying environments.

Chapter 6: Evaluation

6.1 Introduction

The primary goal of evaluating the system was to observe if evolved populations could appear within the environment. Once this was observed, seeing how well and in what ways the populations performed was observed. To do this, tests looking at how the evolutionary decision tree evolved as well as what state populations spent their time in were performed. By looking at this data, a baseline can be created to see how well this proposed system works, as well as what areas need attention. Additional tests were performed based off of the out from BugScraper in order to better simulation a potential application of this system within a code city.

6.2 Observing Fitness

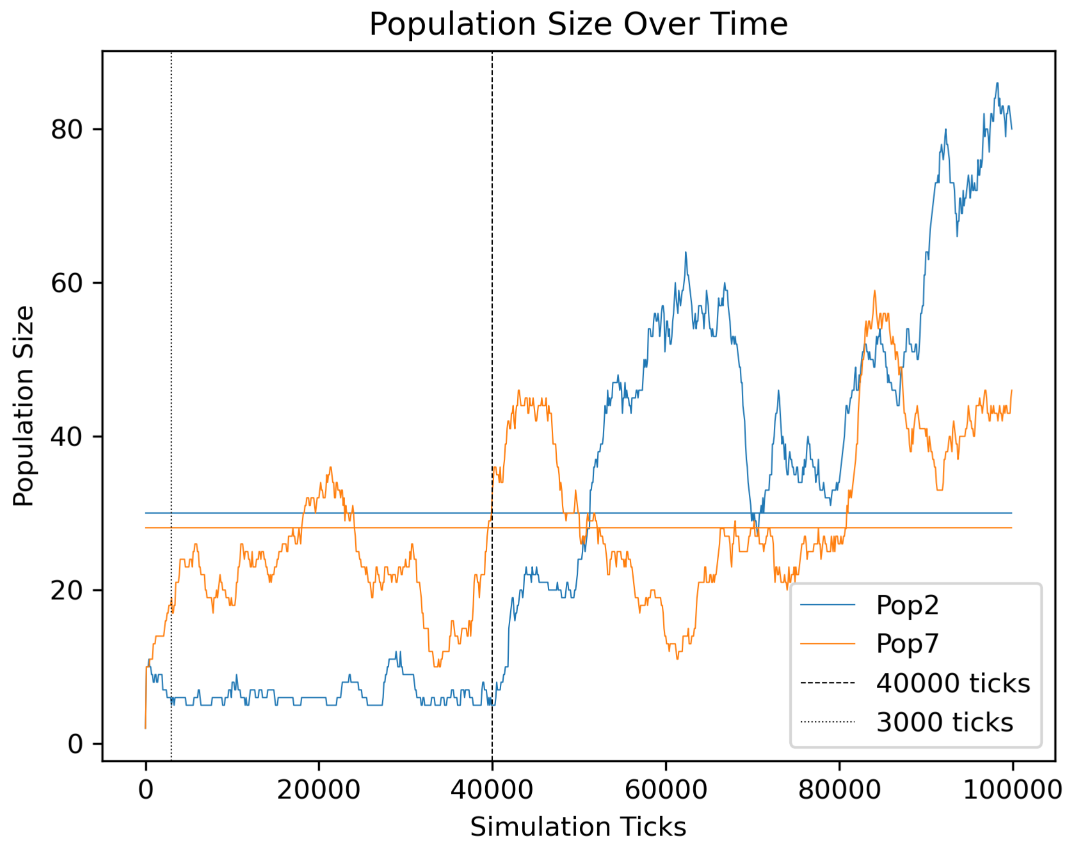


Figure 6.1: Graph of high performing populations.

Taken from a random run, Figures 6.1, 6.2, and 6.3 shows the size of each bug population over

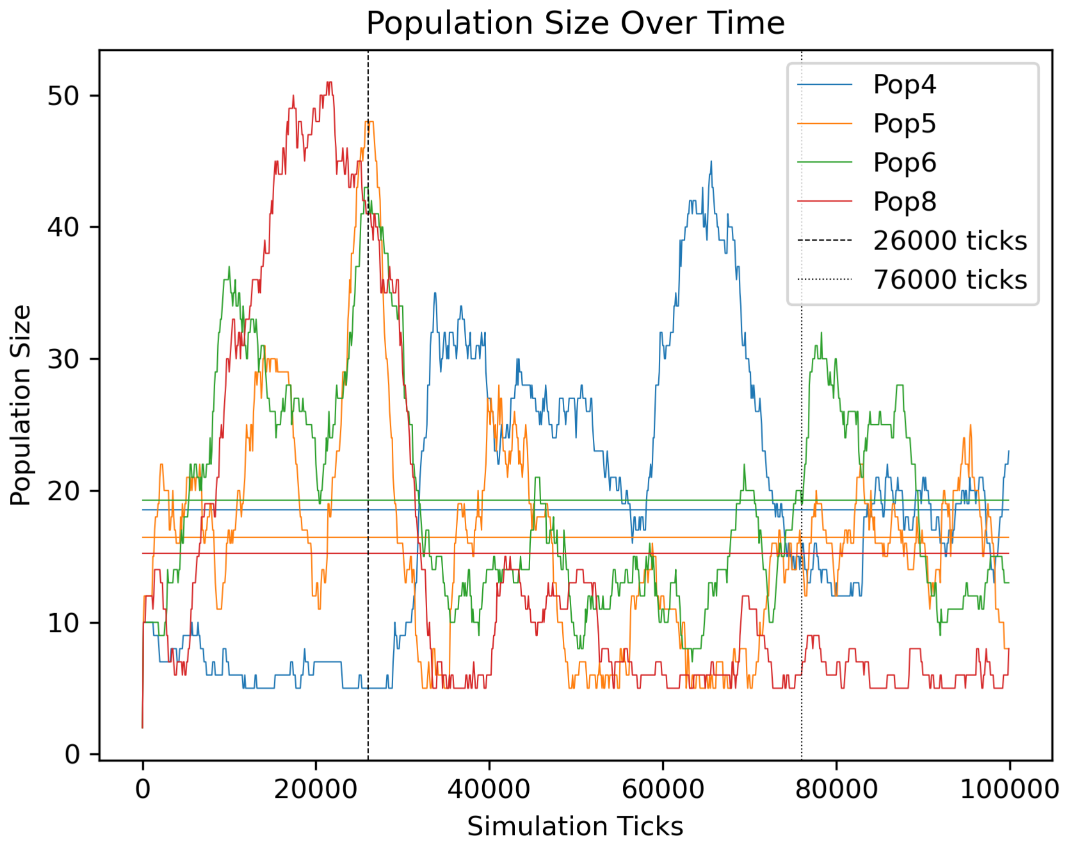


Figure 6.2: Graph of medium performing populations.

the course of 100,000 ticks. The additional horizontal line of matching color represents the average population over the course of the simulation. For the combined figure, refer to Figure B.1.

Table 6.1 shows the average size of a population over the course of the simulation. By using the values in Table 6.1, these populations can be divided into groupings of bug population averages: a high, medium, and low performance groups. These groups can be defined as high by having an average population of 20 or higher, medium between 10 and 20, and low being below 10. Figure 6.1 shows populations 2 (blue) and 7 (yellow) both fall into the high performance grouping. For the first half of the simulation, population 2 is unable to produce bugs that are able to both sustain and grow. However, near 40,000 ticks, an ideal set of genetics appear in the population and the population is able to rapidly

Table 6.1: Average population size of population over duration of simulation.

Population Number	0	1	2	3	4	5	6	7	8	9
Average Population Size	6.61	7.30	30.00	7.92	18.52	16.43	19.27	28.14	15.25	6.96

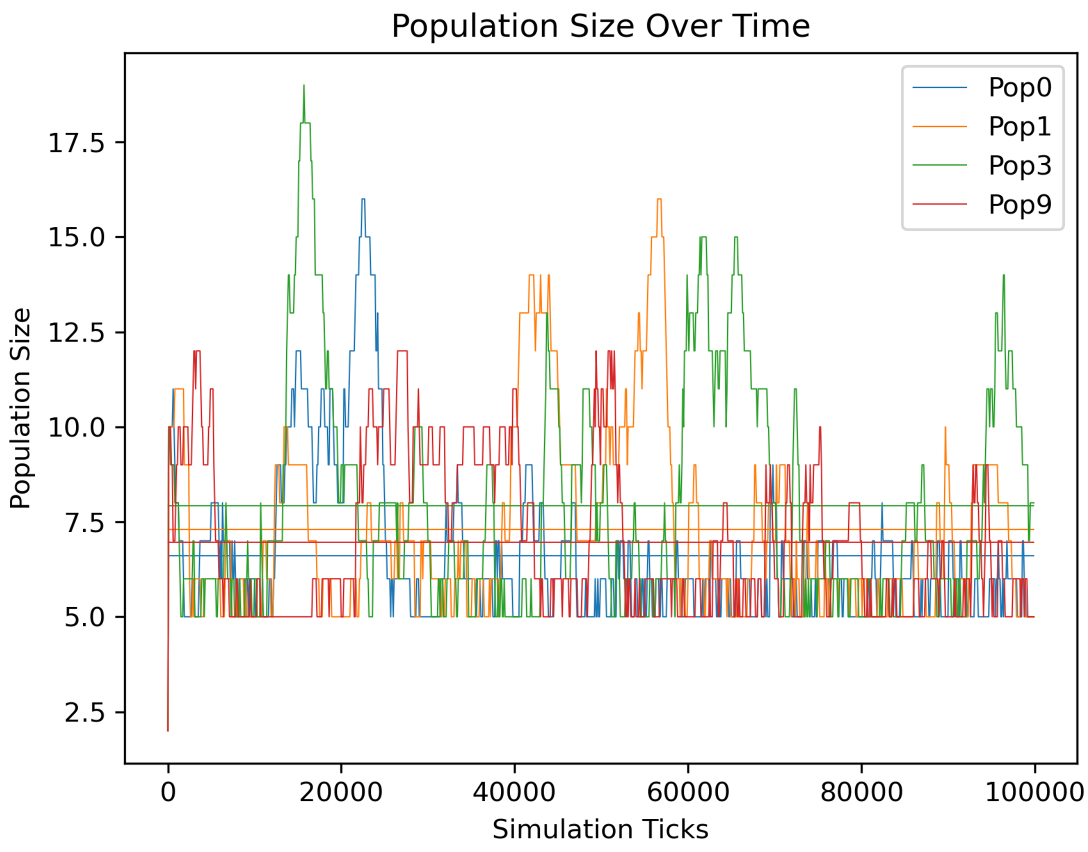


Figure 6.3: Graph of low performing populations.

grow, reaching the highest population both at a single point, and on average. Population 7 on the other hand is able to reach a population of over 20 simultaneous bugs just after 3000 ticks have passed. While there are peaks of population growth, population 7 does not reach the same explosive growth as population 2. Nevertheless, once established, the population was able to survive multiple large scale die-offs, and maintain a high population average throughout the simulation.

Figure 6.2 shows populations 4 (blue), 5 (yellow), 6 (green), and 8 (red) fall into a medium performance group. These populations either sustain themselves in smaller populations, or experience a large population boom, followed by a mass extinction with the inability to adapt to the changes in the environment. Populations 5, 6, and 8 are able to excel in the first 26,000 ticks of the simulation. However, an extinction event occurs and all populations experience a massive drop in numbers. While population 8 is completely wiped out, both population 5 and 6 are able to persist at reduced numbers, surviving and with population 6 even growing back at around 76,000 ticks. Population 4 experiences a population boom just after 26,000 ticks, where the other 3 populations in this group take a dive. Once established, the population is able to sustain itself for the rest of the simulation.

Table 6.2: Performance groups with average over 10 simulation runs.

Run	1	2	3	4	5	6	7	8	9	10	
High	3	2	2	3	2	4	3	2	2	4	2.7
Medium	3	5	3	0	4	3	5	5	1	0	2.9
Low	4	3	5	7	4	3	2	3	7	6	4.4

Finally, Figure 6.3 show populations 0 (blue), 1 (yellow), 3 (green), and 9 (red), all fall into the low performance group. These groups have, at best, a small population spike, but are unable to evolve into a sustained population.

Looking at Table 6.2, we can observe the general performance of bug populations. Defining high and medium performing populations as successful, and low as unsuccessful, over 10 runs 56% are able to sustain some notable level of population, while 44% are not. One noticeable aspect from this data is that there tends to be a higher amount of low performance populations the lower the medium group gets. This is due to the high performing group achieving such a large population that it severely limits other populations from being able to have a chance at evolving.

6.3 Evolution of Evotree

Looking at populations from the perspective of lineage, from the simulation graphed in Figure 6.1 and Figure B.1, due to the populations high population count it may not be a surprise that population 2 had the highest performing bugs, both in longest continuous generation as well as longest surviving bug. Figures 6.4 and 6.5 depict the first and last evolutionary trees of the bugs in the longest continuous population of 32 generations within the simulation. The two major observations that can be drawn from the two figures is that the tree grows to a much larger size over the course of 32 generations, and that at some point the bug changes from hunting to gathering. It should also be noted that these figures have been simplified for readability by removing non-reachable branches of the tree. For the full tree, refer to Figures B.2 and B.3.

Investigating further reveals that between generations 21 (Figure 6.6) and 22 (Figure 6.7), a major mutation occurs which causes the bugs behavior to shift. This change in behavior massively boosts the fitness of the bug, from 3320 in generation 21, to 13236 in generation 22. This massive increase in fitness sets the trend for the overall population, as the bug is able to continuously propagate its genetics over a long duration of time. These graphs have been simplified and cropped to view only the left side of the tree for improved readability, refer to Figures B.4 and B.5 for the full tree graphs.

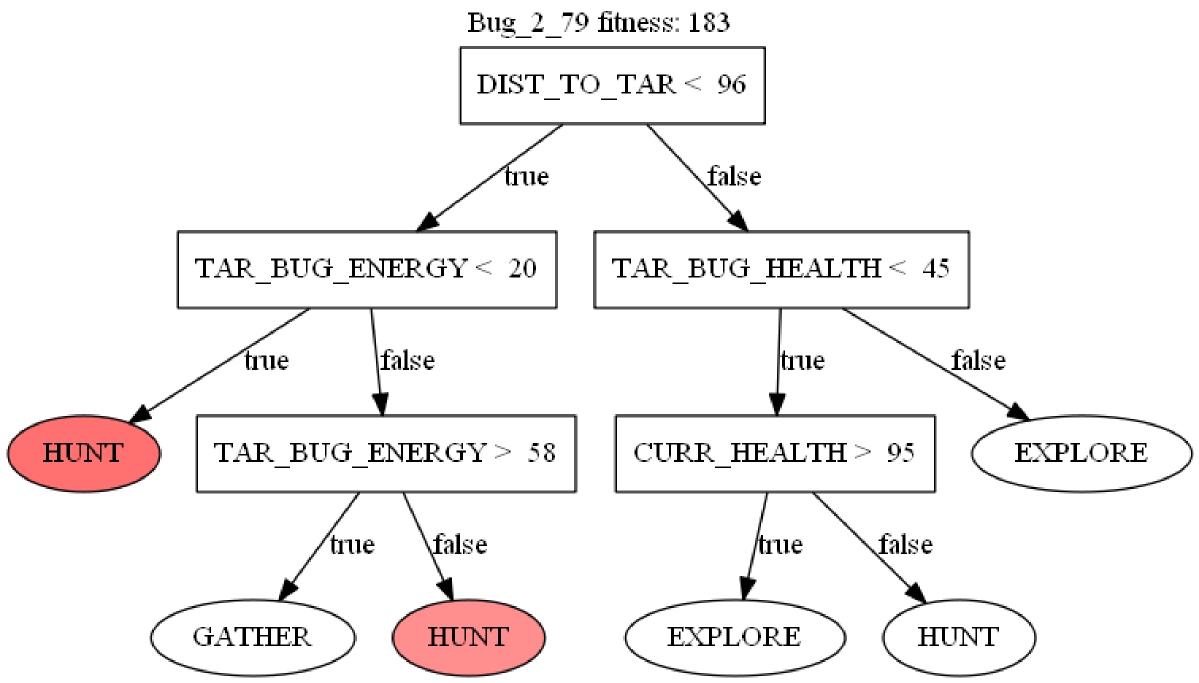


Figure 6.4: Simplified initial evolutionary decision tree configuration for high performing bug.

6.4 Time Spent in State

By looking at populations and their percentage of bugs in a certain state, it appears to reveal certain interactions between populations.

In Figure 6.8 at 47,000 ticks, population 1 spikes to a size of about 50 bugs. Shortly after that, it dips rapidly while population 6 surges in numbers. By looking at both Figure 6.9 and Figure 6.10, we can see how the two populations might have clashed. In Figure 6.9 population 1's behavior is mostly unchanging, likely being able to grow in numbers by gathering information by exploring, and then acting on that information to gather or hunt. However, what is important is that population 1 does not primarily hunt, and by extension, likely has not relied on evolving its combat tree in order to survive. To see the full population graph, refer to Figure B.6.

In Figure 6.10 a sharp and noticeable change occurs beginning around tick 50,000. Both exploration and hunting take up much more of the populations behavior. What this appears to imply is that a bug in population 6 had the genes that excelled in hunting and with the dense numbers of population 1, it had no shortage of prey to consume and as a result, produce many offspring— so much so that it altered the overall population. While population 6 was able to enjoy a level of prosperity, reaching a peak population of over 40 bugs, it likely over-hunted the surrounding bug population and was too adapted to easy hunting. Depicted in Figure 6.8, the population mostly died off at 70,000 ticks, leaving

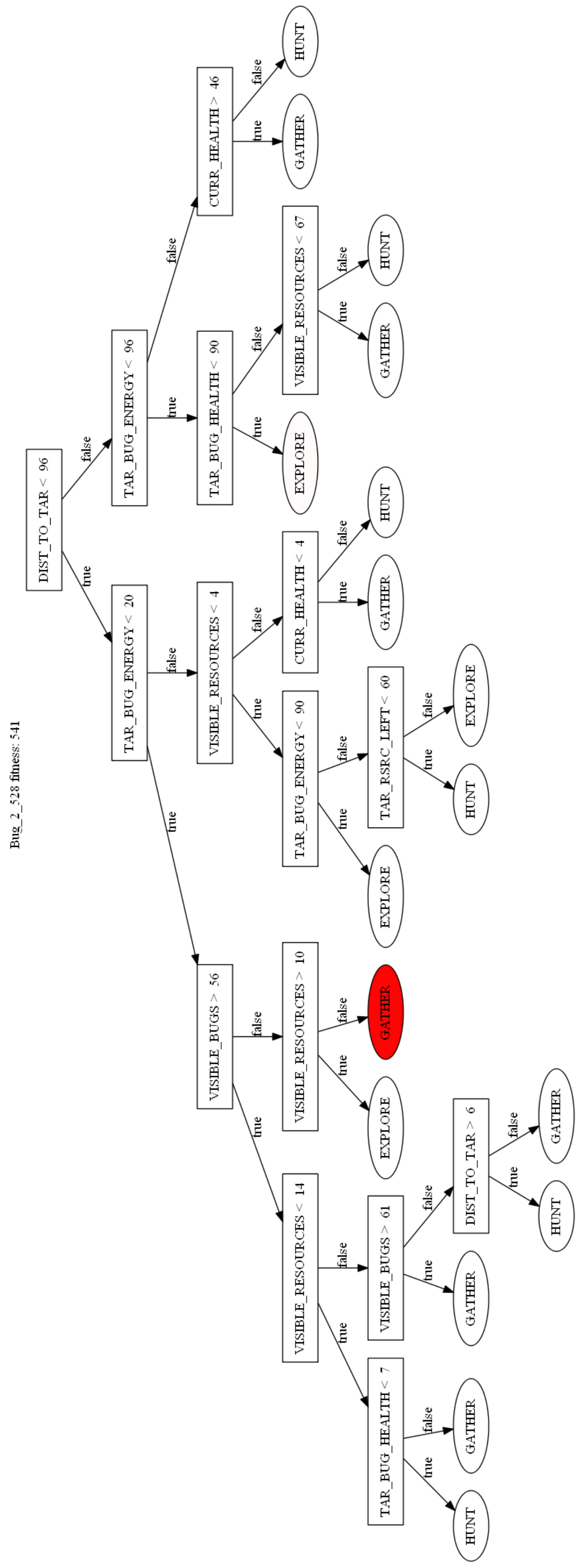


Figure 6.5: Simplified final evolutionary decision tree configuration for high performing bug.

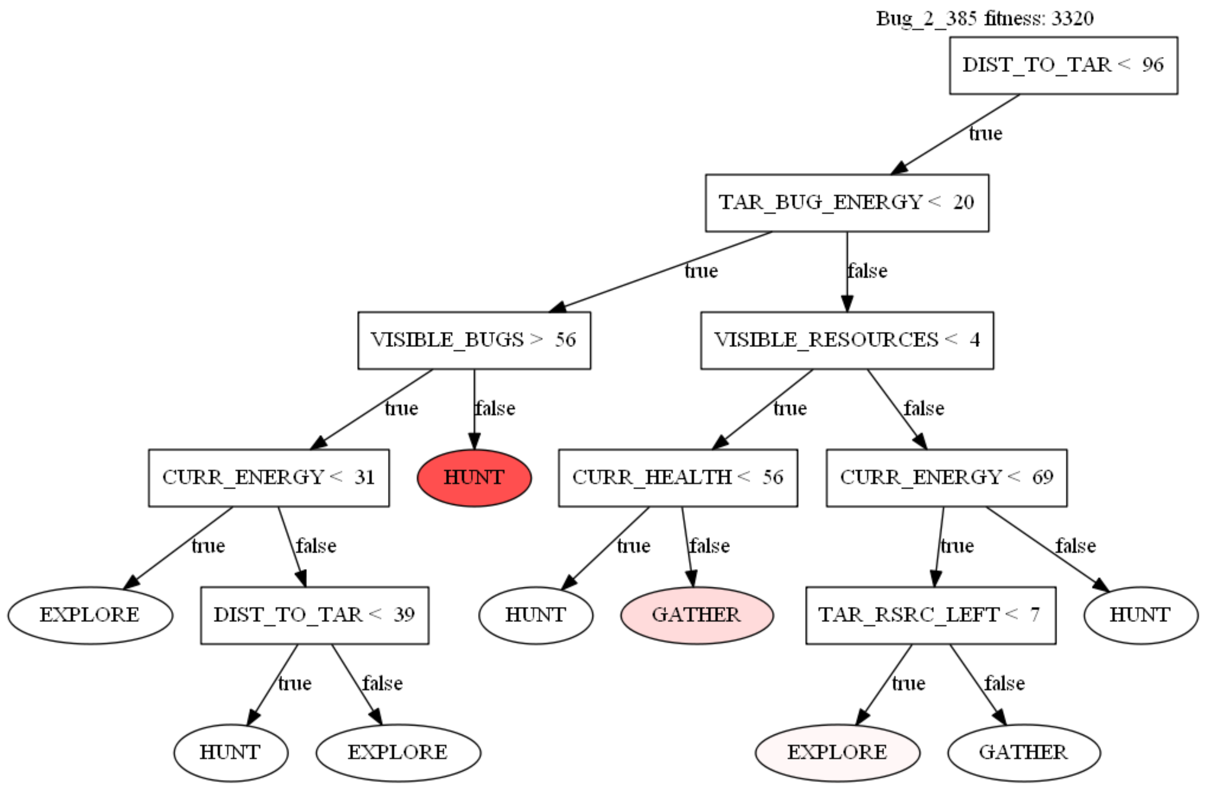


Figure 6.6: Left branch of simplified evolutionary decision tree before major mutation.

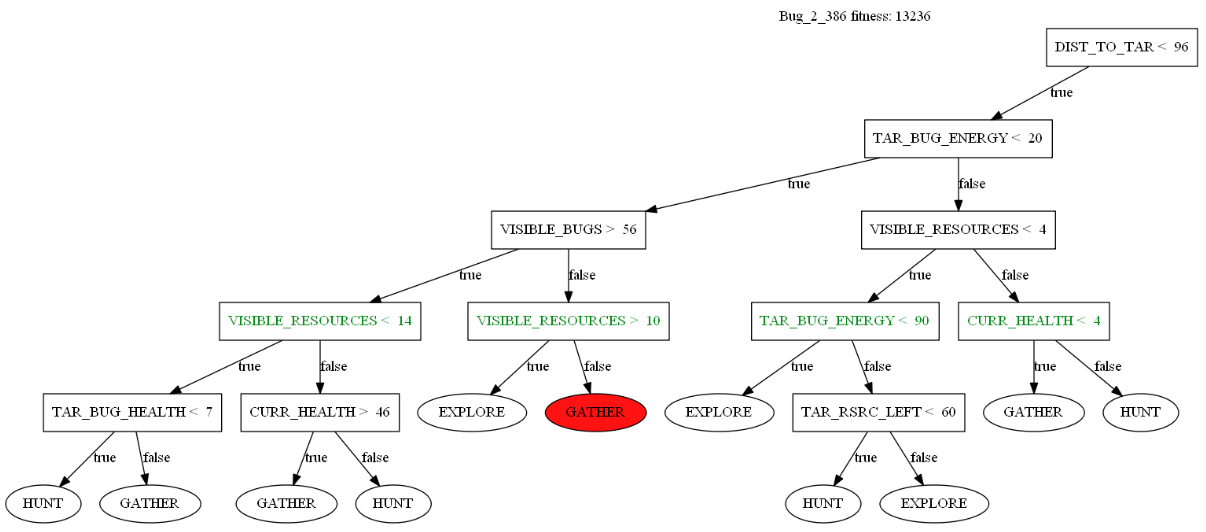


Figure 6.7: Left branch of simplified evolutionary decision tree after major mutation.

the population at only a fraction of the size.

It is also worth noting that after population 6 had died off, population 1 began to make a comeback; likely due to having a preference towards exploration and thus spreading out their numbers allowing for survivors to repopulate.

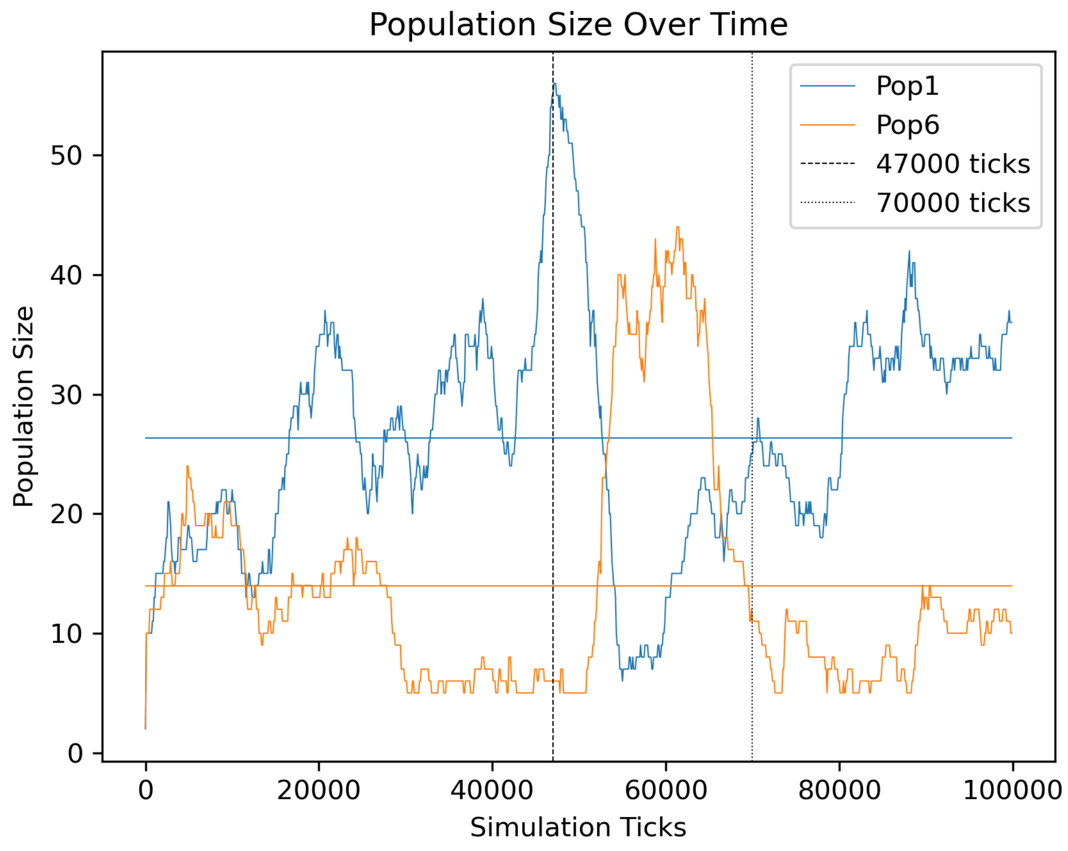


Figure 6.8: Graph of populations 1 and 6 for time spent in state comparison.

6.5 Simulating A Repository Environment

6.5.1 Adding New Populations over Time

By configuring BugSim to create spawnpoints at a staggered rate, we can simulate and observe the behavior of populations if the spawnpoints were created when actual bug tickets were submitted. For these tests, an initial population (population 0) is created at the beginning of the simulation, and every following 10,000 ticks, a new spawnpoint is created.

Looking at Figure 6.11, we can see a massive difference between population performance. Table 6.3 shows that populations 0, 1, and 4 fall into the high performance category, population 2 fall into

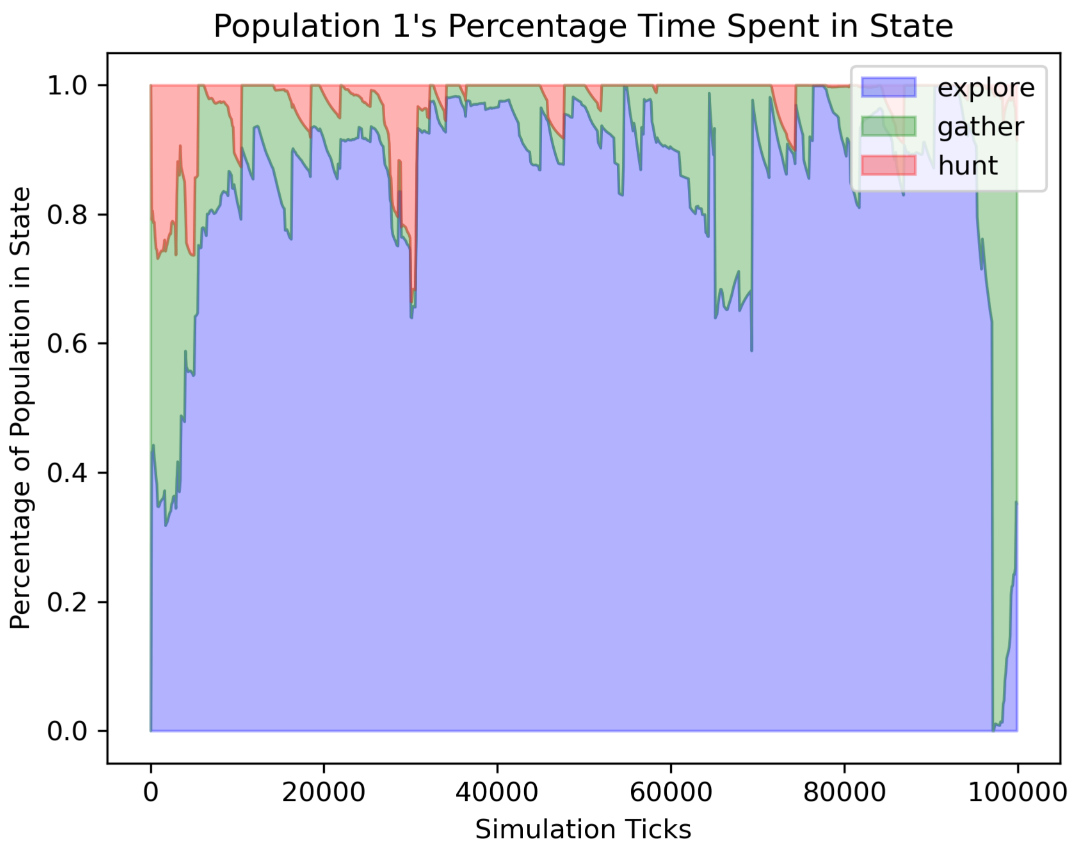


Figure 6.9: Percentage of population in state over time for population 1.

the medium performance category, and remaining populations 3, 5, 6, 7, 8, and 9 all fall into the low performance category. One aspect of note from these results is that population 1 vastly outperforms population 0. Possible explanations of this are either population 0 adapted to an environment with bugs, whereas population 1 adapted to a population with bugs or that this occurred simply due to the random nature of the simulation.

While these results showing the performance of populations over the course of the entire simulation, due to the delay in addition of spawnpoints, a temporal element must be observed when gauging the performance of populations. By averaging the population only over the time in which they were active, we can get a better understanding of how the newly created population was able to adapt to an already

Table 6.3: Average population size of staggered populations over duration of simulation.

Pop. Number	0	1	2	3	4	5	6	7	8	9
Avg. Pop. Size	46.514	111.688	14.749	4.884	21.9	3.402	3.454	1.996	1.411	2.706

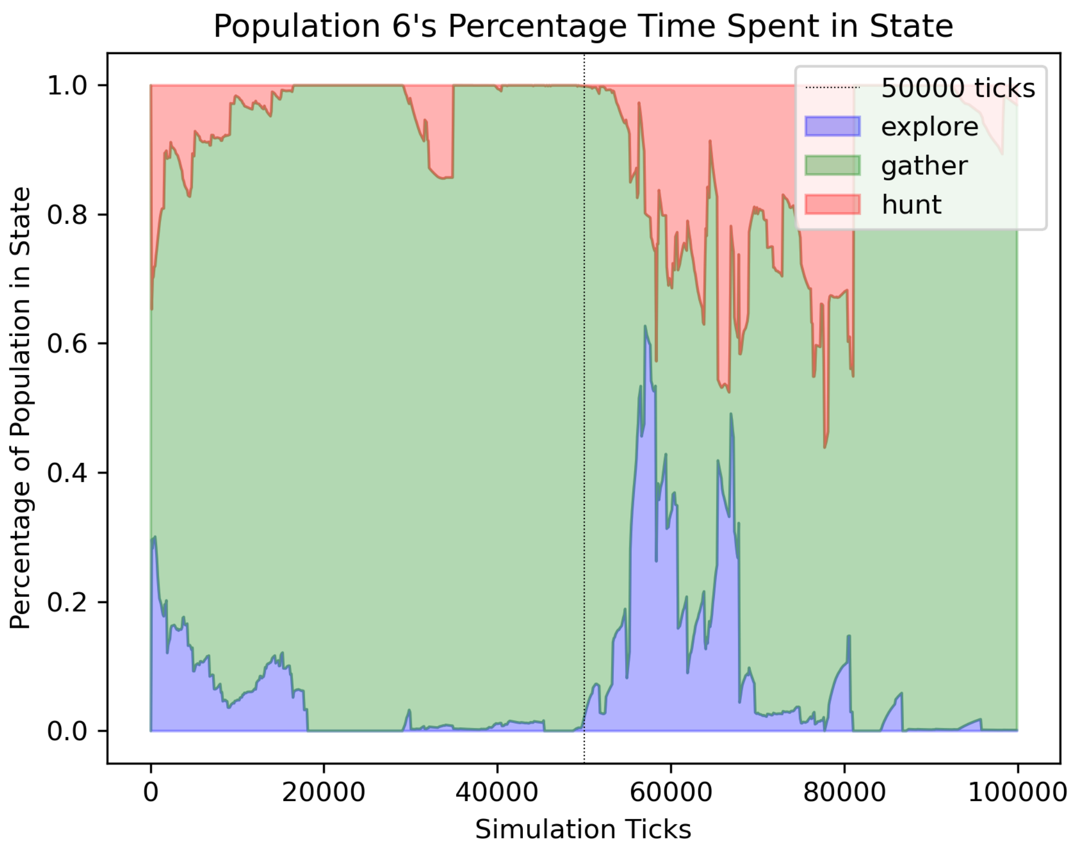


Figure 6.10: Percentage of population in state over time for population 6.

active environment. While most populations remain within their performance category when considering time, population 9's performance actually moves from low performance to high performance. Due to this population only existing for 10,000 ticks, it is hard to say if the population would maintain itself beyond this point. It does however show that even arriving late to the environment, it is still possible to adapt to it.

Expanding the analysis to a set of 10 runs with the same setup, we can see that the performance follows a straight forward trend. Table 6.5 shows that, on average, the populations average size declines the later the spawnpoint is introduced to the environment. While this result is not surprising, it does show that spawnpoints, and by extension the corresponding bug ticket, will likely be more noticeable

Table 6.4: Average population size of staggered populations over duration of spawnpoint lifetime.

Pop. Number	0	1	2	3	4	5	6	7	8	9
Avg. Pop. Size	46.514	124.098	18.436	6.977	36.5	6.804	8.635	6.653	7.055	27.06

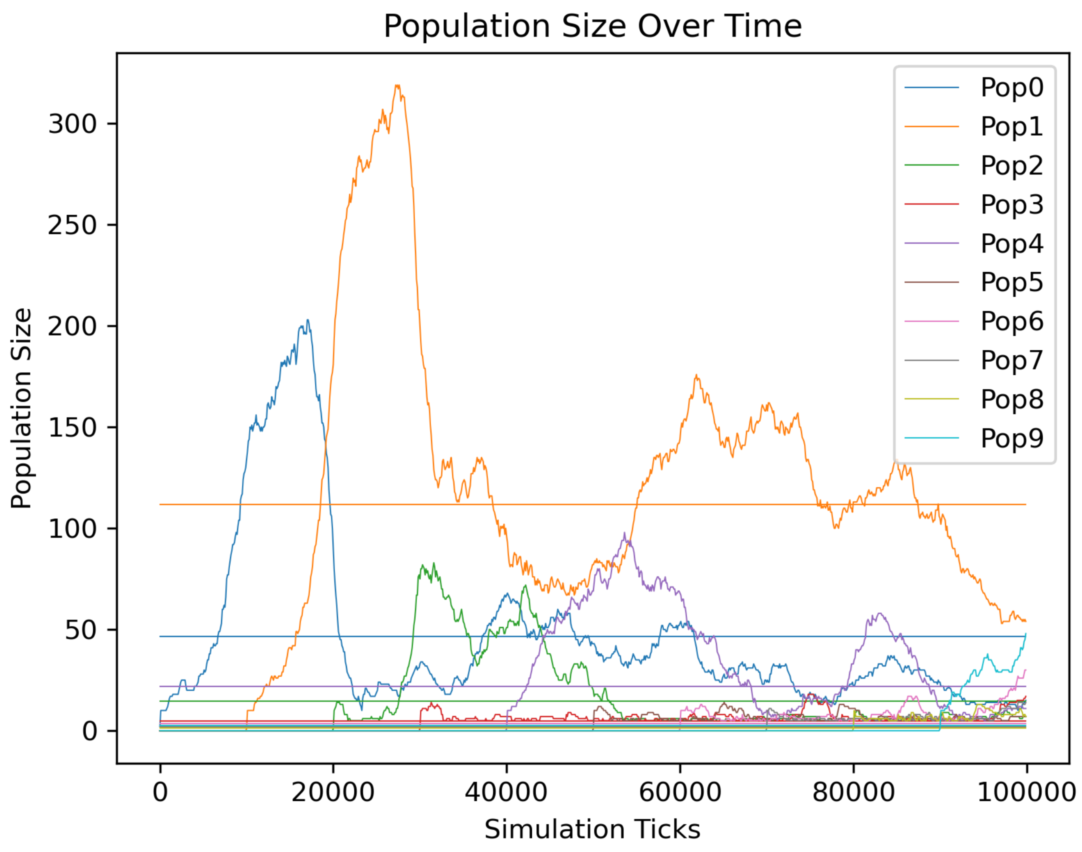


Figure 6.11: Graph of population size over time with staggered populations.

within the environment due to having more bugs populating the environment.

A more interesting result can be observed when looking at Table 6.6 which factors in population size with respect to the time the spawnpoint is introduced. We can see that it still follows a general trend of population average decreasing the later the spawnpoint is introduced. However, it also shows that populations 4, 5, and 6 have population averages which fall into the medium performance category. Compared to the first 4 populations having a high performance category, and the last 3 having a low performance category, this system configuration would promote the visibility of older bug tickets over more recent submissions.

6.5.2 Modifying Population Based on Bug Priority

Another field to incorporate from a bug ticket would be to use the priority of the bug. To simulate this, three populations are created at the beginning of the simulation, each with different minimum population values. Population 0 has a minimum population size of 5, population 1 has 10, and population 2 has 15. The idea behind this configuration is that a population with a larger size will more likely find a

Table 6.5: Average population size of staggered populations over duration of simulation for 10 runs.

Pop. Num.	0	1	2	3	4	5	6	7	8	9
Run 1	68.699	68.738	54.784	5.597	7.699	3.432	6.557	2.164	1.735	0.849
Run 2	38.722	86.728	5.848	45.399	5.5	5.736	3.857	3.3	4.212	1.056
Run 3	43.411	70.035	26.715	37.936	12.929	5.152	2.62	1.735	1.924	1.098
Run 4	59.467	38.433	5.89	36.741	10.452	13.711	2.836	4.226	1.377	0.751
Run 5	36.623	47.2	47.894	6.057	13.754	14.663	2.715	2.816	1.537	0.891
Run 6	23.009	51.864	14.233	55.931	10.86	3.1	8.139	2.11	1.683	0.677
Run 7	88.05	20.874	12.391	43.929	14.876	6.188	10.133	1.854	1.247	1.469
Run 8	71.639	51.023	7.957	48.658	5.831	3.752	11.285	2.989	2.24	0.756
Run 9	83.164	37.447	7.947	40.517	4.548	23.983	14.844	4.398	1.573	0.756
Run 10	41.611	8.924	90.702	8.504	9.831	20.084	11.716	2.012	1.248	0.971
Average	55.44	48.127	27.436	32.927	9.628	9.98	7.47	2.76	1.878	0.927

Table 6.6: Average population size of staggered populations over duration of spawnpoint lifetime for 10 runs.

Pop. Num.	0	1	2	3	4	5	6	7	8	9
Run 1	68.699	76.376	68.48	7.996	12.832	6.864	16.393	7.213	8.675	8.49
Run 2	38.722	96.364	7.31	64.856	9.167	11.472	9.643	11.0	21.06	10.56
Run 3	43.411	77.817	33.394	54.194	21.548	10.304	6.55	5.783	9.62	10.98
Run 4	59.467	42.703	7.363	52.487	17.42	27.422	7.09	14.087	6.885	7.51
Run 5	36.623	52.444	59.868	8.653	22.923	29.326	6.788	9.387	7.685	8.91
Run 6	23.009	57.627	17.791	79.901	18.1	6.2	20.348	7.033	8.415	6.77
Run 7	88.05	23.193	15.489	62.756	24.793	12.376	25.333	6.18	6.235	14.69
Run 8	71.639	56.692	9.946	69.511	9.718	7.504	28.213	9.963	11.2	7.56
Run 9	83.164	41.608	9.934	57.881	7.58	47.966	37.11	14.66	7.865	7.56
Run 10	41.611	9.916	113.378	12.149	16.385	40.168	29.29	6.707	6.24	9.71
Average	55.44	53.474	34.295	47.038	16.047	19.96	18.676	9.201	9.388	9.274

Table 6.7: Speed in simulation ticks at which population surpasses x4 minimum population size.

Run	1	2	3	4	5	6	7	8	9	10
Population 0	9500	68500	5600	1300	10200	10000	3100	5900	42800	3600
Population 1	17200	6200	46100	6400	9800	16300	35800	7600	5800	5200
Population 2	N/A	11600	5800	78600	6800	N/A	25600	6100	7100	9800

combination of genes that will allow success within the environment. Figure 6.12 shows a random run with this configuration.

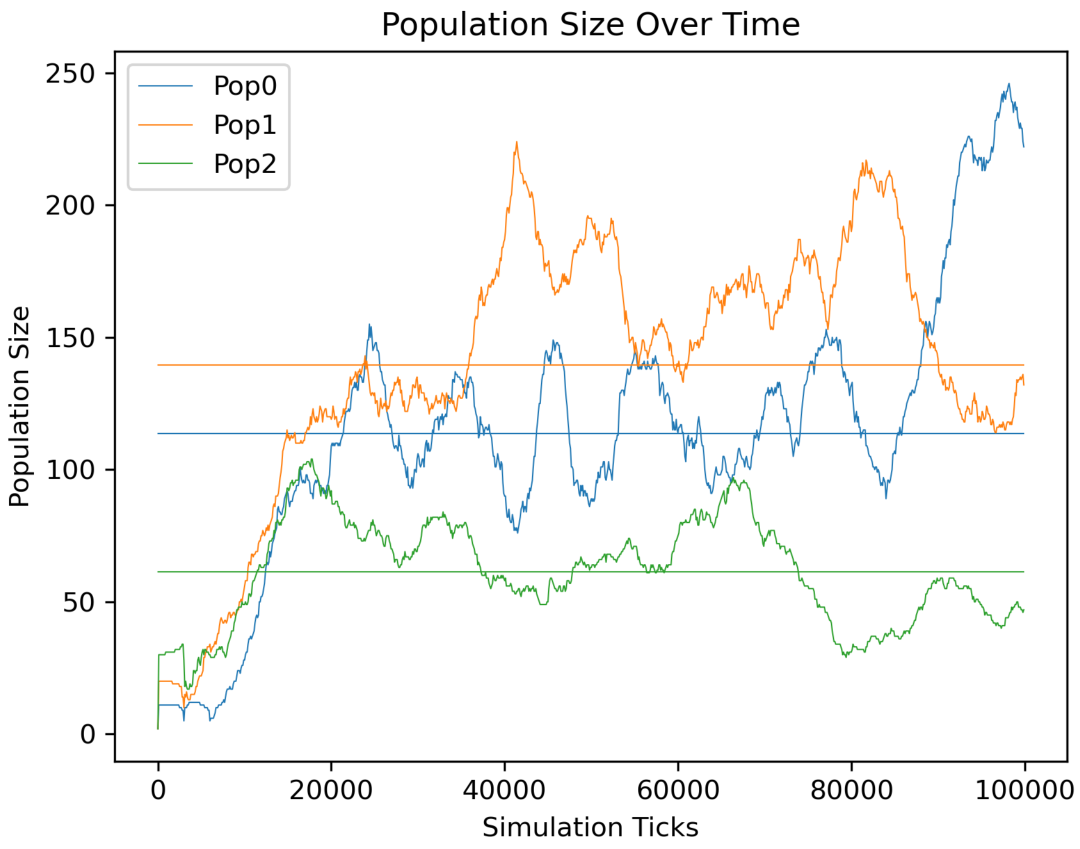


Figure 6.12: Graph of population size over time with modified populations.

To measure the performance of this configuration, the speed at which a population evolves will be the main metric used. Specifically, the speed at which a population reaches four times the minimum population will be observed. This means that population 0 must reach a size of 20, population 1 a size of 40, and population 2 a size of 60. By surpassing this value, the population shows that it has successfully achieved at least decent performing genetics for their environment.

Table 6.7 shows the results of 10 runs in this configuration. The most noticeable aspect of these results is that in two occasions, population 2 was unable to achieve a population size of 60 at any point in the simulation. Investigating further, we can look at the graphs of run 1 and run 6 in Figures 6.13 and 6.14. In both cases, population 2 is nearly unable to even sustain a small population, let alone reach the goal of four times its minimum population.

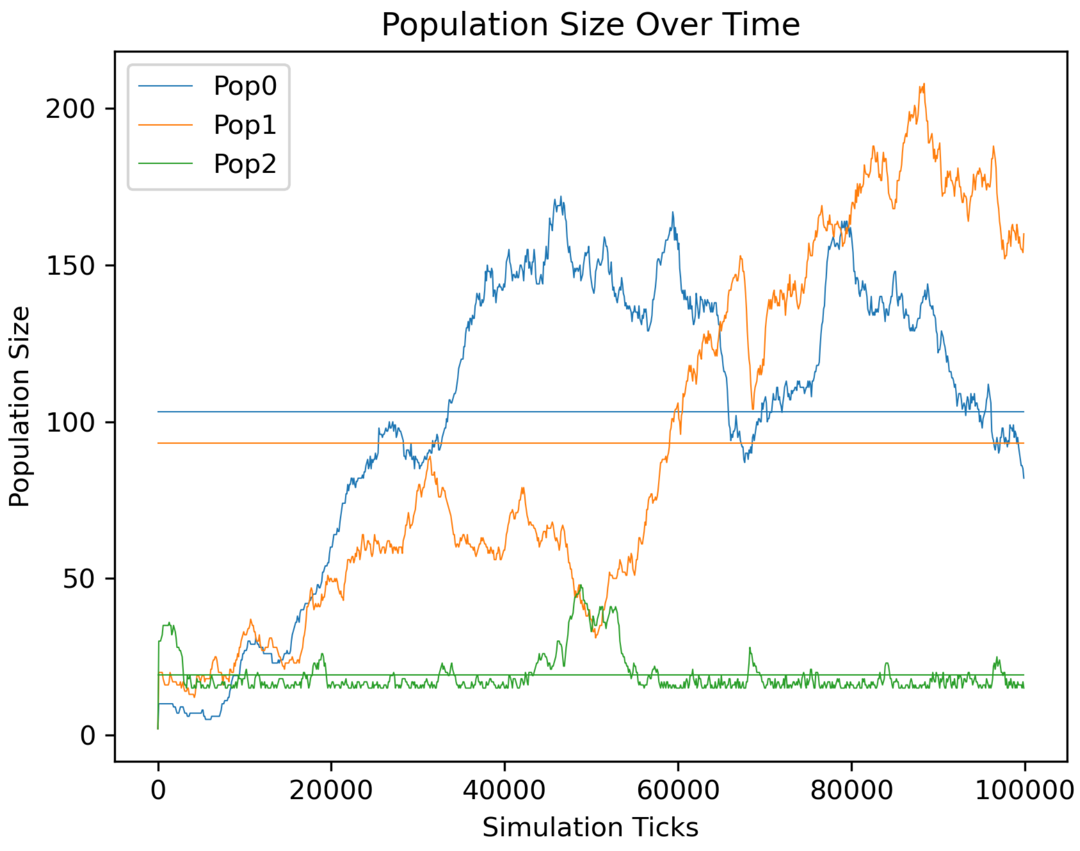


Figure 6.13: Run 1 graph of population size over time with modified population.

Ignoring the two failed populations, Table 6.8 shows the result of averaging out the time in which the populations reach four times their minimum population. While the larger minimum population size populations do reach an evolved state faster than the smaller populations, all three populations reach their mark within 1000 ticks of each other. This combined with the large variability in which the populations reach the stated goal, as well as population 2 failing to evolve at all, seems to indicate that this method of translating bug ticket information into the environment not ideal. Either a different way to translate the bug ticket priority should be used, or additional testing must be performed on this configuration in order to improve it.

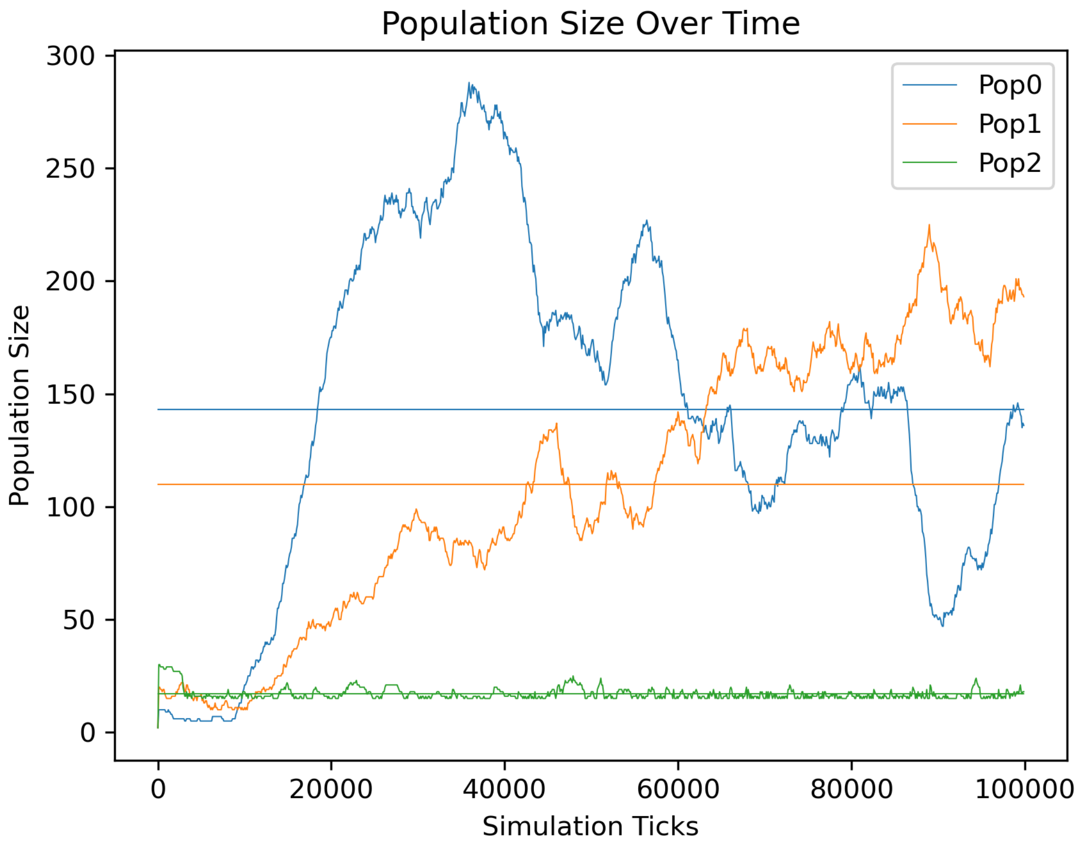


Figure 6.14: Run 6 graph of population size over time with modified population.

Table 6.8: Average time in ticks of population to reach x4 minimum population size.

Population 0	16050
Population 1	15640
Population 2	15140

Chapter 7: Summary and Conclusions

This system is able to, on average across 10 runs, produce 5 populations of successfully evolved bugs. By maintaining an average population of higher than 10 bugs over the course of the entire simulation, the population shows that it has been able to evolve a set of genes fit for the environment. Naturally, the extent of their evolution varies as some groups are unable to sustain themselves, but by surpassing this benchmark, at the very least the population was able to have a significant presence for an extended period of time within the environment. Furthermore, multiple populations are able to coexist— from the data observed, no single population was able to have total control over the environment. That being said, there are still cases where as few as 3 populations are able to control the majority of the environment, preventing other populations from having a chance to evolve.

Additionally, this system is able to produce robust groups of bugs capable of surviving mass extinction events. It should be noted that the inclusion of small resource nodes is what allows the survival of an otherwise successful population, should a large portion of it be suddenly wiped out. While this does rely on the modification of the environment, it can be considered what is required in breathing life into a code city.

While this system does not directly implement a player agent to interact with the environment, the player (or players) would be classified as a separate population. This means that bugs who hunt other bugs would also see the player(s) as a potential food source, and harass those within the vicinity. The hope is that this achieves a level of interaction to draw the player towards the bug, and when implemented in an actual code city, look at how to fix the bug.

From the set of tests integrating hypothetical output from BugScraper, it shows a promising application of use in an actual code city. Time can be used as a means to control how prominent a certain population is, with older populations (and by extension, older bugs), tending to grow larger populations compared to more recent bug reports. Utilizing the priority field would require a more involved solution than what was tested within this work. Thankfully, priority does not have to be tied to minimum population size, and can be applied to almost any other aspect of a bug or its population.

7.1 Future Work

As this work was not initially intended to be a simulation, but rather an implementation, there are many branches to explore for further work— some related to implementation, but others emerging from the final results of this thesis.

7.1.1 Implementation

First and most obvious would be an actual implementation of what BugSim simulates into a code city environment. This would involve connecting the control method of bugs, the evolutionary decision tree, to an NPC agent within the virtual environment. Said agents would need to be able to receive input from the environment in the form of sight, as well as interact with other agents, such as combat with other bugs as well as interaction with the player/user.

Further work in implementation would be to fully connect the output of BugScraper to bug behavior, beyond the currently supported 1 ticket = 1 bug population. First and foremost, scanning the description of the bug to more intelligently place a spawnpoint is a good place to start. Further work in this area could potentially take the form of strengthening more pressing bugs within the repository, while weakening less important ones. Additionally, other fields from BugScraper could change how the bug might look visually, such as older tickets resulting in older looking bugs. Other discussed but unimplemented ideas include bugs destroying their surrounding environment such as damaging buildings, harassing players/users, and "dropping" ticket information upon death, akin to monsters dropping treasure in video games.

Additionally, further improvements to BugScraper can be made, namely implementing a means to circumvent the restrictions enforced by the GitHub API. A proposed solution for this would be to have a way to track what needs to be updated, and only query what is needed in order to limit the number of accesses.

7.1.2 Optimization

While BugSim is able to make good use of multi-threading, there are still many avenues to improve performance. All data structures used in the program are fairly simplistic, being no more than linked lists and binary trees. In particular, one of the larger bottlenecks during runtime is when bugs scan their surroundings. Both the checking of the area around the bug, as well as storing relevant data can likely be improved upon.

Optimization is also doubly important on a larger scale. Stress tests of higher populations of bugs, in a larger environment, would also be worthwhile. Since it is hardly uncommon to see larger repositories with hundreds of open tickets, this system would likely need improvement to see more practical use—and stress tests could point out the exact areas that need attention. As a stress test of a different sort, running the simulation for an extended period of time could also reveal some areas that require more work. In testing, the simulation appears to not reach a stagnant state, but other, smaller issues may come up over extended periods of runtime.

7.1.3 Bug Improvement

As mentioned in the previous section, improvement upon the simulations would likely benefit the overall program, however improvements specific to the evolutionary decision tree is definitely an area of interest. At the moment, the evotree uses its non-terminal nodes as a way of converting the information a bug observes into output. The effectiveness of the types of current nodes as well as the possibility of adding new, (potentially) more information dense nodes are definitely areas of study.

On the other end, more bug behavior could be explored. Within this thesis, bug actions were intentionally kept as simplistic as possible, but that doesn't necessarily mean better. Perhaps new states outside of "explore", "gather", and "hunt" could be implemented, to build a more robust bug, capable of adapting to a wider range of environments. Improvement, or at least refinement of existing states could be beneficial as well— such as how movement (and to an extent pathfinding) is handled.

7.2 Conclusions

The work done in this thesis supports the idea of using an evolutionary algorithm as a control scheme for representing dynamic entities representing bug reports within a code city. Populations are able to evolve robust genes in a multi-population environment to produce large populations, as well as having varied evolutionary paths. Tests simulating the integration of bug ticket creation date also support the potential of incorporating, and leveraging, real-world data found within a bug report to put more attention on a certain grouping of bug tickets (in this case, a larger focus on older bugs).

References

[1] Shadi Banitaan and Mamdouh Alenezi. Tram: An approach for assigning bug reports using their metadata. In *2013 Third International Conference on Communications and Information Technology (ICCIT)*, pages 215–219. IEEE, 2013.

[2] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 308–318, 2008.

[3] Tegawendé F Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillere, Jacques Klein, and Yves Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*, pages 188–197. IEEE, 2013.

[4] T Bullen and M Katchabaw. Using genetic algorithms to evolve character behaviours in modern video games. *Proceedings of the GAMEON-NA*, 2008.

[5] Zhen-Guo Che, Tzu-An Chiang, Zhen-Hua Che, et al. Feed-forward neural networks training: a comparison between genetic algorithm and back-propagation learning algorithm. *International journal of innovative computing, information and control*, 7(10):5839–5850, 2011.

[6] Nicholas Cole, Sushil J Louis, and Chris Miles. Using a genetic algorithm to tune first-person shooter bots. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, volume 1, pages 139–145. IEEE, 2004.

[7] Jin-Hyuk Hong and Sung-Bae Cho. Evolving reactive npcs for the real-time simulation game. In *CIG*. Citeseer, 2005.

[8] Talib S Hussain and Gordon Vidaver. Flexible and purposeful npc behaviors using real-time genetic control. In *2006 IEEE international conference on evolutionary computation*, pages 785–792. IEEE, 2006.

[9] Su-Hyung Jang and Sung-Bae Cho. Evolving neural npcs with layered influence map in the real-time simulation game 'conqueror'. In *2008 IEEE symposium on computational intelligence and games*, pages 385–388. IEEE, 2008.

[10] Clinton L Jeffery. The city metaphor in software visualization. In *27. International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, pages 153–161. WSCG, Václav Skala-UNION Agency, 2019.

[11] Claire Knight and Malcolm Munro. Virtual but visible software. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 198–205. IEEE, 2000.

[12] Petri Lankoski and Staffan Björk. Gameplay design patterns for believable non-player characters. In *DiGRA Conference*, pages 416–423, 2007.

[13] Francois Dominic Laramee. Genetic algorithms: Evolving the perfect troll. *AI game programming wisdom. Charles River Media*, pages 629–639, 2002.

[14] Brian Mac Namee and Padraig Cunningham. A proposal for an agent architecture for proactive persistent non player characters. Technical report, Trinity College Dublin, Department of Computer Science, 2001.

[15] Thomas Panas, Thomas Epperly, Daniel Quinlan, Andreas Saebjornsen, and Richard Vuduc. Communicating software architecture using a unified single-view visualization. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 217–228. IEEE, 2007.

[16] Natthakul Pingclasai, Hideaki Hata, and Ken-ichi Matsumoto. Classifying bug reports to bugs and other requests using topic modeling. In *2013 20Th asia-pacific software engineering conference (APSEC)*, volume 2, pages 13–18. IEEE, 2013.

[17] Juraj Vincur, Pavol Navrat, and Ivan Polasek. Vr city: Software analysis in virtual reality environment. In *2017 IEEE international conference on software quality, reliability and security companion (QRS-C)*, pages 509–516. IEEE, 2017.

[18] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, pages 921–922, 2008.

[19] Xin Ye, Razvan Bunescu, and Chang Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699, 2014.

Appendix A: BugScraper Source Code

Listing A.1: Bugscraper source code.

```
import sys
import requests
import json


def main(argv):

    url = argv[1]

    if (url.find("sourceforge") != -1):
        print("Sourgeforge␣url␣found")
        bug_list = handle_sourceforge(url)
    elif (url.find("github") != -1):
        print("Github␣url␣found")
        bug_list = handle_github(url)
    else:
        print("Invalid␣url,␣exiting...")
        exit()

    final_fp = open("bug_tickets.json", "w")
    json.dump(bug_list, final_fp)



# Good testing repositories
# https://sourceforge.net/p/bibdesk/bugs/
# https://sourceforge.net/p/freedos/bugs/
# https://sourceforge.net/p/mingw/bugs/

def handle_sourceforge(url):
    #below is an example of an api call that will retrieve all issues from
        a repository
        #https://sourceforge.net/rest/p/unicon/bugs/
        #details extracted by appending a number corresponding to a bug
            entry
        #for example, if you wanted the 28th bug entry, the call would be:
        #https://sourceforge.net/rest/p/unicon/bugs/27

    # passed in url
    # https://sourceforge.net/projects/mingw/

    repo_name = url.split("/")

    # bug url
    # https://sourceforge.net/rest/p/mingw/bugs/

    bug_url = "https://sourceforge.net/rest/p/" + repo_name[4] + "/bugs/"

    print(bug_url)
```

```
response = requests.get(bug_url)
print(response.status_code)
if response.status_code == 404:
    print("404 response, resource not found, exiting...")
    exit()

ticket_numbers = []

page_count = 0
loop_flag = True

while loop_flag is True:
    print("Querying tickets... " + str(len(ticket_numbers)+1))

    bug_page_url = bug_url + "?limit=100&page=" + str(page_count)

    bug_page = requests.get(bug_page_url)
    bug_json = bug_page.json()


    if len(bug_json["tickets"]) == 0:
        print("finished counting tickets")
        loop_flag = False
    else:
        for t in bug_json["tickets"]:
            ticket_numbers.append(t["ticket_num"])

    bug_page.close()
    page_count += 1


bug_list = []

max_tickets = len(ticket_numbers)+1

while len(ticket_numbers) > 0:
    i = ticket_numbers.pop()

    print("Logging ticket: " + str(i) + " out of " + str(max_tickets))

    bug_page_url = bug_url + str(i)
    bug_page = requests.get(bug_page_url)
    bug_json = bug_page.json()

    bug_entry = {}

    bug_entry["ticket_num"] = bug_json["ticket"]["ticket_num"]
    bug_entry["status"] = bug_json["ticket"]["status"]
    bug_entry["reported_by"] = bug_json["ticket"]["reported_by"]
    bug_entry["summary"] = bug_json["ticket"]["summary"]
    bug_entry["description"] = bug_json["ticket"]["description"]
    bug_entry["created_date"] = bug_json["ticket"]["created_date"]
    bug_entry["mod_date"] = bug_json["ticket"]["mod_date"]
```

```
        bug_entry["priority"] = bug_json["ticket"]["custom_fields"]["
            _priority"]

        bug_list.append(bug_entry)

        bug_page.close()

    return bug_list



# Good testing repositories
# https://github.com/TheAlgorithms/Python
# https://github.com/ossu/computer-science

# With no authentication, you can only make 60 requests per hour

def handle_github(url):
    # below is an example of an api call that will retrieve 50 items from
        page 0 from all issues from a repository
        # https://api.github.com/repos/TheAlgorithms/Python/issues?state=
            all&per_page=50&page=0

    # passed in url
    # https://github.com/TheAlgorithms/Python

    repo_name = url.split("/")


    # bug url
    # https://api.github.com/repos/TheAlgorithms/Python/issues
    bug_url = "https://api.github.com/repos/" + repo_name[3] + "/" +
        repo_name[4] + "/issues"
    print(bug_url)

    tickets = []

    # github starts pages at 1 (page 0 and 1 will give identical results)
    page_count = 1
    loop_flag = True

    while loop_flag is True:
        print("Querying␣tickets...␣" + str(len(tickets)+1))
        bug_page_url = bug_url + "?state=all&per_page=100&page=" + str(
            page_count)

        bug_page = requests.get(bug_page_url)
        bug_json = bug_page.json()

        # Perform 404 check here to save 1 request
        if bug_page.status_code == 403:
            print("403␣response,␣exceeded␣60␣requests␣per␣hour,␣exiting..."
                )
            exit()
        elif bug_page.status_code == 404:
            print("404 response, resource not found, exiting...")
            exit()


        if len(bug_json) == 0:
            print("finished obtaining tickets")
            loop_flag = False
        else:
            tickets = tickets + bug_json

        bug_page.close()
        page_count += 1

    bug_list = []
    max_tickets = len(tickets)
    counter = 1

    while len(tickets) > 0:
        print("Logging ticket: " + str(counter) + " out of " + str(
           max_tickets))
        entry = tickets.pop()

        bug_entry = {}

        bug_entry["ticket_num"] = entry["number"]
        bug_entry["status"] = entry["state"]
        bug_entry["reported_by"] = entry["user"]["login"]
        bug_entry["summary"] = entry["title"]
        bug_entry["description"] = entry["body"]
        bug_entry["created_date"] = entry["created_at"]
        bug_entry["mod_date"] = entry["updated_at"]
        bug_entry["priority"] = -1 # set to -1 sentinal value as github
           does not have priority

        bug_list.append(bug_entry)

        counter += 1
    return bug_list



if __name__ == "__main__":
    main(sys.argv)
```

Appendix B: Figures

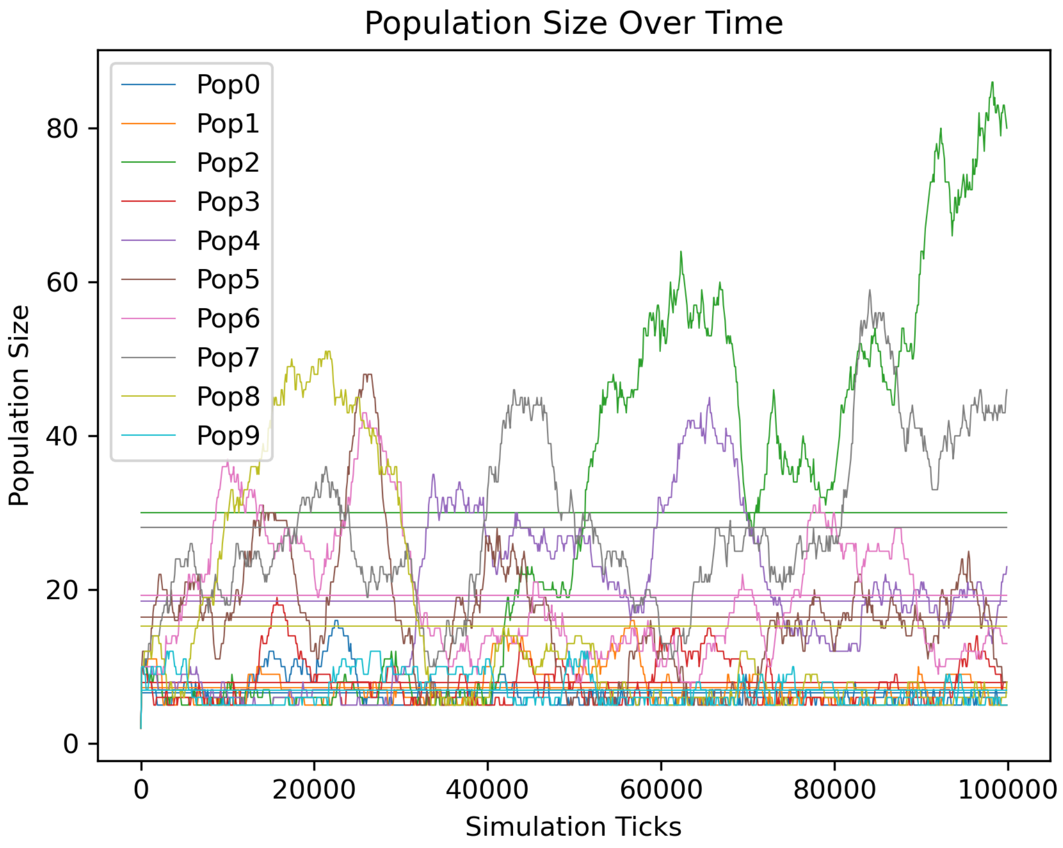


Figure B.1: Graph of population size over time for random simulation run.

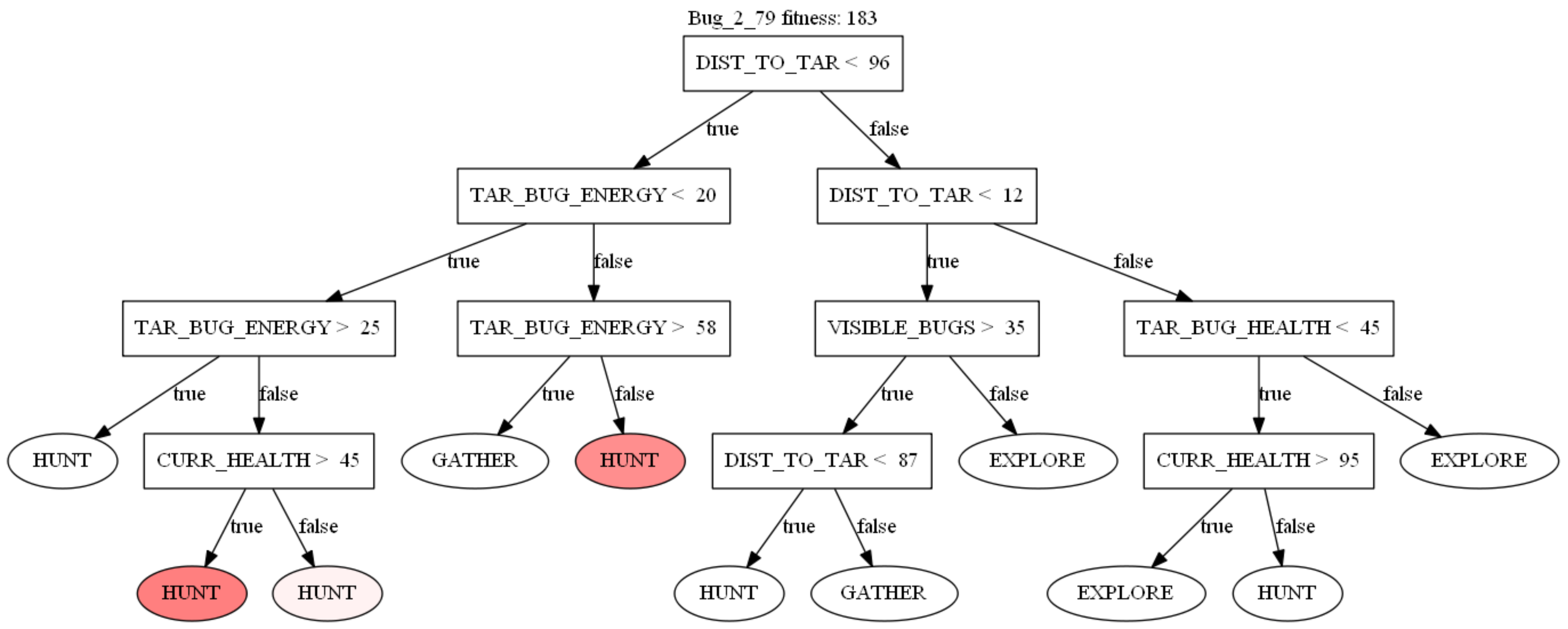


Figure B.2: Initial evolutionary decision tree configuration for high performing bug.

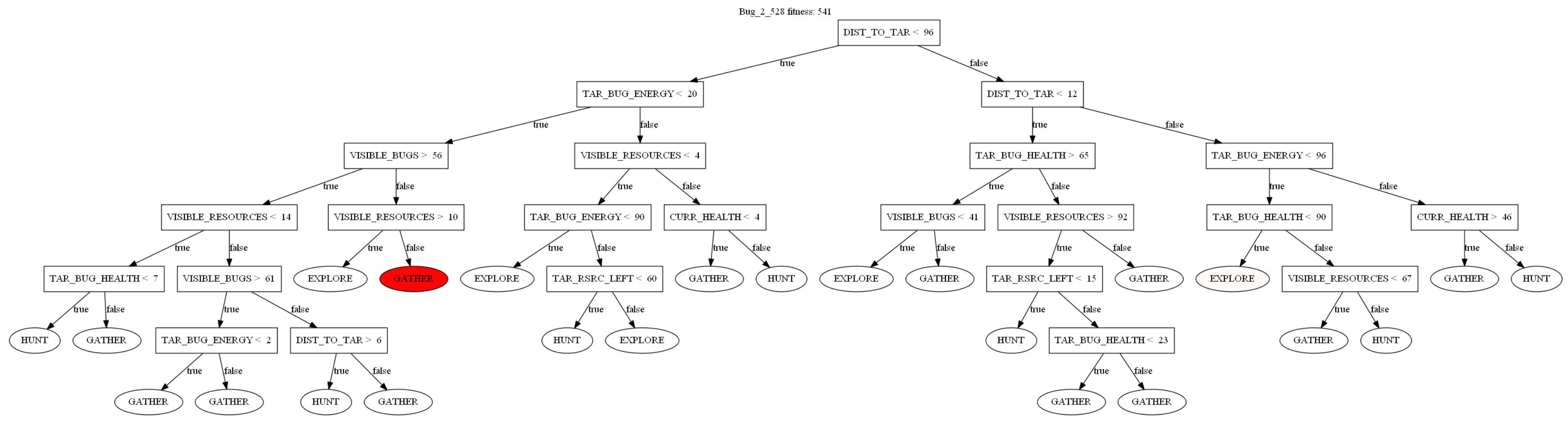


Figure B.3: Final evolutionary decision tree configuration for high performing bug.

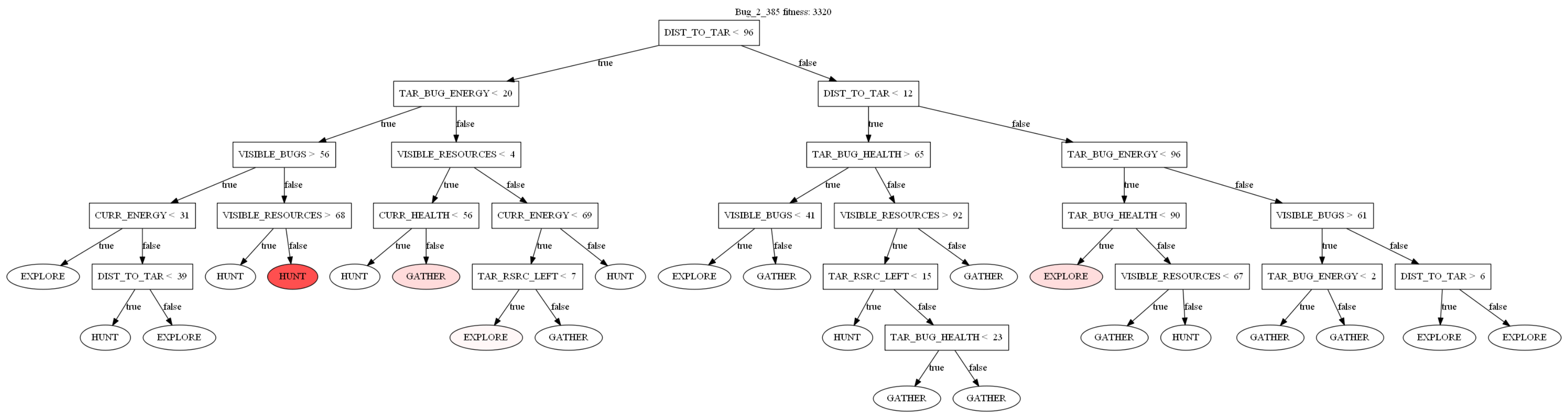


Figure B.4: Evolutionary decision tree before major mutation.

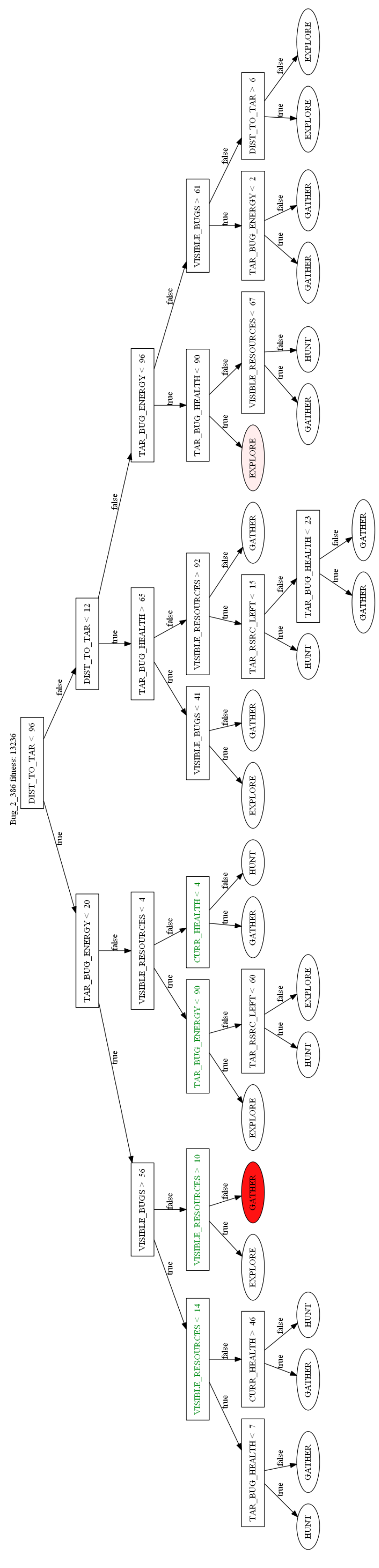


Figure B.5: Evolutionary decision tree after major mutation.

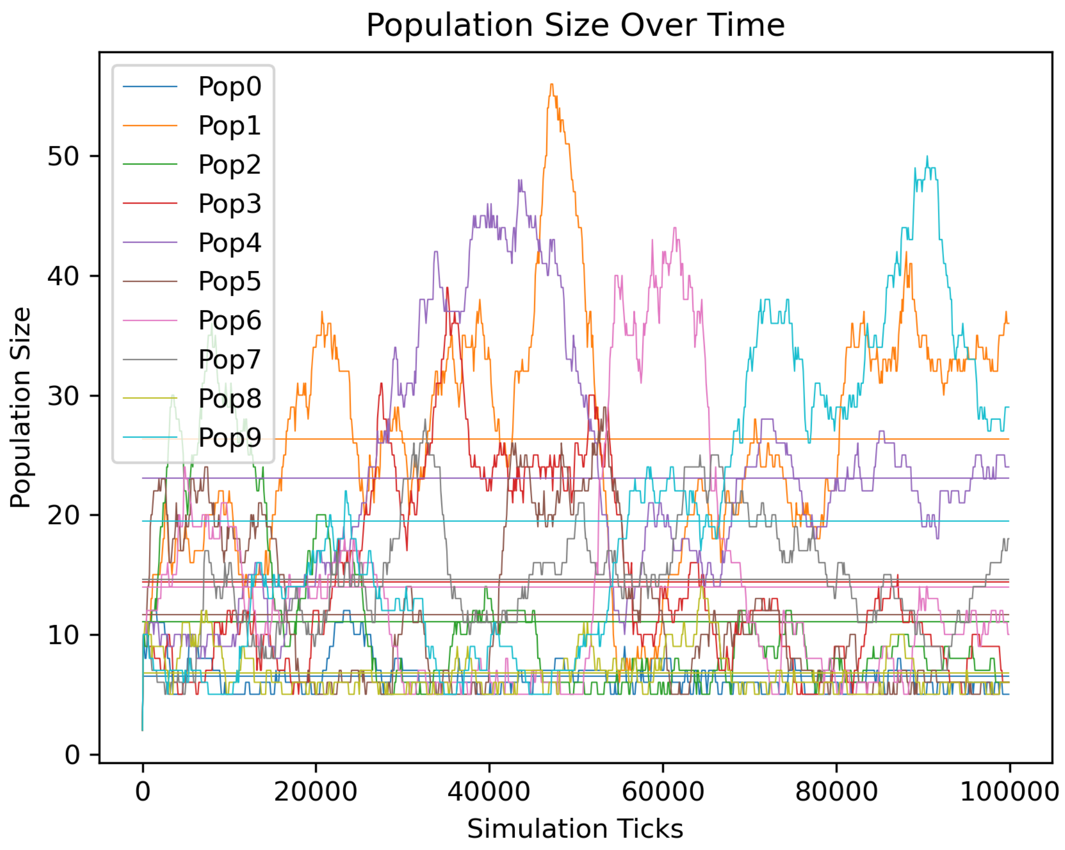


Figure B.6: Graph of population size over time for time spent in state comparison.