Genetic Programming for the RoboCup Rescue Simulation System

Andrew Runka, B.Sc. Computer Science (Hons.)

Computer Science

Submitted in partial fulfillment
of the requirements for the degree of

Masters of Science

Faculty of Computer Science, Brock University
St. Catharines, Ontario

Abstract

The Robocup Rescue Simulation System (RCRSS) is a dynamic system of multi-agent interaction, simulating a large-scale urban disaster scenario. Teams of rescue agents are charged with the tasks of minimizing civilian casualties and infrastructure damage while competing against limitations on time, communication, and awareness. This thesis provides the first known attempt of applying Genetic Programming (GP) to the development of behaviours necessary to perform well in the RCRSS. Specifically, this thesis studies the suitability of GP to evolve the operational behaviours required of each type of rescue agent in the RCRSS. The system developed is evaluated in terms of the consistency with which expected solutions are the target of convergence as well as by comparison to previous competition results. The results indicate that GP is capable of converging to some forms of expected behaviour, but that additional evolution in strategizing behaviours must be performed in order to become competitive. An enhancement to the standard GP algorithm is proposed which is shown to simplify the initial search space allowing evolution to occur much quicker. In addition, two forms of population are employed and compared in terms of their apparent effects on the evolution of control structures for intelligent rescue agents. The first is a single population in which each individual is comprised of three distinct trees for the respective control of three types of agents, the second is a set of three co-evolving subpopulations one for each type of agent. Multiple populations of cooperating individuals appear to achieve higher proficiencies in training, but testing on unseen instances raises the issue of overfitting.

Acknowledgements

A thank you to...

Dr. Beatrice Ombuki-Berman for setting me on the path and opening the doors.

To my supervisory committee for sharing their insights which drove this thesis to completion.

To my external examiner Dr. Marcus V. dos Santos for his helpful suggestions.

To the staff of the Computer Science department at Brock University for the years of support.

A special thanks to Cale Fairchild for addressing my concerns on an increasingly frequent basis.

Thank you to those closest to me for your patience, understanding, and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis utilizes Genetic Programming (GP) to evolve the decision logic for a set of rescue agents in the RoboCup Rescue (RCR) agent simulation domain.

The RoboCup Rescue competition [1] is an branch of the well-known RoboCup Soccer (RCS) league [2]. Similar to RCS, RCR is a multi-agent simulation competition in both robotics and artificial intelligence. The RCR, however, involves a more complex environment than its predecessor with multiple teams of cooperating agents and dynamic objectives. The RCR system models a large-scale urban disaster scenario similar to the aftermath of an earthquake in a densely populated city.

International competitions in RCR take place annually. Three RCR leagues exist: a robot league, a virtual robot league, and a simulated agent league. This thesis focusses on the simulated agent league. Here, instead of controlling actual robots, a simulation environment is provided to replicate a post-earthquake disaster scenario. Competitors in the RCR simulated agent league are charged with the task of developing an intelligent mechanism for the control of emergency response teams. Namely, there exist Ambulance, Fire Brigade, and Police Force teams, which must rescue civilians, extinguish fires, and clear roadblocks respectively. Each agent plays an important role in the mitigation of this simulated crisis, both individually and in conjunction with each other.

As an established competition of 10 years, RCR offers not only a standardized simulation environment but also a wealth of benchmark results from all of the past competing teams. Therefore it is an appropriate system for the development and evaluation of novel approaches. An additional inspiration for the use of this environment in this thesis is an ideological motivation to further the knowledge in a collaborative field directed towards helping those in need.

GP is a popular form of automatic programming which uses the evolutionary mechanism of natural selection to evolve potentially complex programs from simple building blocks. The approach of this thesis is to utilize Genetic Programming to evolve the controlling behaviours of the emergency teams in the RCR Simulation System (RCRSS). To the best of the author's knowledge this is the first

attempted application of GP to the development of agent control behaviours for the RCRSS. Much research has been done in the field of evolving intelligent behaviours, such as the foraging behaviour of ants [3, 4] and predator-prey behaviours [5, 6]. However, considerably less literature exists on the evolution of more complex behaviours such as those required in this multi-agent, multi-objective, dynamic environment.

A common motivation for the utilization of GP in novel situations is its potential to discover unexpected solutions that a manually-coded solution may overlook. Whether intentionally or not a human programmer will imbue their solution with the biases of their training and outlook. GP's admittedly chaotic approach to searching the solution space may yield solutions beyond the limits of a programmer's expectations.

Since the RCR domain is a complex one (having eluded a definitively optimal solution amidst a decade of competition), it may seem unreasonable to expect GP to perform well at all. While it may be a daunting task, a successful application of an automation technique to the development of such complex intelligence would be a great stride in Machine Learning and speaks to the fundamental motivation of Artificial Intelligence.

It should be made clear from the outset that the goal of this work is not to surpass the previously benchmarked results of past RCR competitors, although such a result would be considered a success. The main goal of this thesis is to study the suitability of GP for evolving the necessary 'intelligence' required within this complex environment. This study is not one-dimensional. The interest here is as much about individual emergent behaviours as it is about the coalescence of those behaviours into an effective set of heterogeneous intelligent rescue agent teams. This thesis examines the ability of variations of GP to evolve several specific behaviour elements that are required in the RCRSS. The main contributions of this thesis are as follows:

- *First application of GP to RCRSS*

  To the best of this author's knowledge, this thesis presents the first known application of Genetic Programming to the RoboCup Rescue Simulation System. This entails the development of a suitable GP language to describe the RCR environment with sufficient depth and modularity to allow for a wide range of possible solutions.

- *Step-wise learning paradigm for GP*

  This thesis presents a definition and analysis of the 'step-wise' learning paradigm for GP. This is a strategy to simplify the computational effort required by GP to solve a problem by subdividing the solution space using subsets of the GP language. This is shown to reduce the size of the initial search space without limiting the sufficiency of the GP language as a whole.

- *Exploration of GP's suitability for developing certain intelligent behaviours*

  A Genetic programming system is applied to the development of the operational behaviours required to perform the expected tasks in the RCRSS. This thesis primarily focuses upon GP's

ability to develop the operational behaviours, which are those behaviours specified by the problem itself. This is evaluated in terms of the consistency with which the evolved populations converge to the expected behaviours. It was found that given a descriptive enough fitness function to guide the search, GP is capable of determining simple go-to-and-act behaviours, but not more complex tasks such as knowing how and when to refuel.

- *Comparison of two forms of GP population*

  Two types of GP population are developed for the control of this multi-agent system. A comparison performed between a single population of heterogenous individuals and co-evolved cooperating subpopulations of homogenous individuals. It was found that the use of multiple subpopulations inspired higher proficiencies in training, however, this lead to an overfitting issue resulting in lower performance when tested on unseen problem instances.

Finally, the best performing behaviours during training are tested against the preliminary competition results from RoboCup Rescue 2006. The results indicate promise for the ability of GP to create solutions, but the overall performance does not suggest that automatic programming is a sound replacement for manual programming techniques in this domain.

The remainder of this dissertation is organized as follows. Chapter 2 explains the background information of the elements of this thesis. A description of the RCRSS is presented in terms of the problem to be solved as well as a brief overview of its implementation. Following this is a brief tutorial of Genetic Programming. Finally, a review is presented of work related to Genetic Programming, Cooperative Multi-Agent Systems, the RoboCup Domain, and RoboCup Rescue. Chapter 3 describes the system implemented for this thesis. This includes the application of GP to the RCRSS, the GP language developed, and a break-down of the objectives this system is applied to. Chapter 4 presents the setup details, results, and discussion for the experiments performed in this thesis. Finally, chapter 5 summarizes the conclusions of this thesis including suggestions and intended directions of future work.

Chapter 2

Background Information

This chapter presents background information necessary for the rest of this thesis. Specifically, the sections of this chapter describe RoboCup Rescue, Genetic Programming, and previous works related to these topics and this thesis as a whole.

2.1 Robocup Rescue Simulation System

On January 17th 1995, the Great Hanshin Earthquake hit the Japanese port city Kobe. It measured 6.8 on the Moment magnitude scale, causing approximately 6500 deaths, and upwards of $100 billion in damage [7, 1]. The scale of the damage was largely due to the high density of the population in Kobe. In response to this disaster it was decided that the current disaster relief policies needed to be rethought and restructured.

On April 30th 1999, at the location of Japan's local RoboCup Soccer (RCS) competition, it was decided that a simulator would be created to model large scale urban disasters, and the Robocup Rescue Simulation System (RCRSS) and Robocup Rescue (RCR) league were founded [1]. Building on the success of RCS, it is hoped that the RCR simulator and competitions will serve to promote international research collaboration in the fields of AI and robotics, toward the development of improved rescue capabilities.

RCR is a post-earthquake urban disaster mitigation simulation. The RCR league itself consists of three competitions: the robotics competition, the virtual robot competition and the simulated agent competition. This thesis focusses on the latter.

2.1.1 Problem Objectives

The RCRSS attempts to model a real-world large scale urban disaster. Accordingly, the objectives of the simulation are similar to those you would expect in a real-world disaster. Version 0.49.9[1] of the simulator, which is used in this thesis, uses Formula (2.1) as the scoring function for a single run.

$$Score = (P + \frac{H}{H_{init}}) * \sqrt{\frac{B}{B_{max}}} \tag{2.1}$$

where P is the number of persons alive, H is the amount of health points of all agents (civilians included), H_{init} is the initial health points of all agents, B is the area of buildings that remain unburnt, and B_{max} is the area of all buildings.

From this it can be seen that this problem can be expressed in terms of multi-objective optimization. The two main objectives are the minimization of civilian casualties (and injuries) and the minimization of damage done to buildings by fire. A third sub-objective not expressed in Formula (2.1) is the clearing of debris from the roads. Three types of agents exist in this environment and are individually tasked with accomplishing these objectives. These agent types are the Ambulance Teams (AT), Fire Brigades (FB), and Police Forces (PF).

The *saving civilians* objective is generally performed by the AT agents. This objective, however, can be broken down into three smaller tasks. Firstly, the civilian must be located. This task can be accomplished by any one of the three types of agents (AT, FB, or PF). Following this, the civilian must then be rescued from the collapsed building. This task can only be performed by AT agents, but can be expedited by the use of multiple ATs. Finally, the civilian must be transported back to a refuge, this requires exactly one AT agent to accomplish.

The second objective, *minimizing building damage*, is pursued by the FB agents. Several fires are ablaze throughout the simulated city area. Each fire, if left unattended, will damage the building on which it burns as well as spread to adjacent buildings. As with the saving civilians objective, all types of agents can locate a fire, however only FB agents are capable of extinguishing one. Multiple FBs may work together to extinguish a single fire or spread out to contain many fires. Each FB agent must be mindful of their current water capacity, which is used up while extinguishing and requires refilling at a refuge when low.

A third objective, not expressed in Formula (2.1), is the *clearing of debris* from the roads. This is the task given to the PF agents. Without the removal of roadblocks many civilians and fires will be unreachable. Therefore, although this task does not directly influence the score, a PF team unsuited for this objective will be a major detriment to both other main objectives.

[1] Available online: http://sourceforge.net/projects/roborescue/

2.1.2 Problem Constraints

Hampering the agents' ability to achieve their objectives are a number of constraints, all of which model a real life scenario. The foremost constraint the agents must work against is time. The fires spread and damage buildings over time and civilians lose health while trapped. Thus agents are competing against the clock to complete their objectives as efficiently as possible. Compounding this constraint, the simulator lasts 300 cycles, at a rate of 1 cycle per second, where each cycle simulates one minute of real time. Commands such as taking actions or communicating are issued by the agents once per cycle, however all commands that are not issued in the 1 second allotted are ignored. Thus, in effect, the simulation proceeds in an approximation of real-time, where an uncertain agent can waste valuable time pausing to make up its mind.

The second most significant constraint facing the agents is their own limited awareness. Every cycle, the simulator's kernel issues limited sensory information to each agent at a small radius around their location. This information contains what they see and whether or not they hear someone. Such a limited awareness of a rapidly changing environment means that agents must be highly adaptive in order to proceed effectively with their objectives. Combined, the limited awareness and various time constraints illustrate the fully dynamic nature of this problem.

Additional constraints on the problem are defined in the parameter file on a per-instance basis. Some parameters are given standard values, these include limits on the size and number of communications each agent can send as well as the number of communications they can hear each step, a limited number health points (HP) that are consumed when an agent takes damage from fire, and limited extinguishing range and water capacity for FB agents. Other parameters, such as the map used, and the positions of all objects on the given map, are defined at the time of competition.

2.1.3 Simulator Design

The Robocup Rescue Simulation System (RCRSS) is structured in multiple modules as seen in Figure 2.1. The central module, known as the kernel, connects to all other modules via UDP sockets[2], and performs a managerial role for all of its connected components.

The GIS (Geographical Information System) initializes and provides the kernel with information on the location of every object in the simulation world, including roads, buildings and agents. A number of component simulators connect to the kernel, each controlling individual aspects of the environment such as fires or traffic.

The final set of modules are the agents. The list of agent modules connected to the kernel may contain 0 or more of the following: Civilians, Fire Brigades (FB), Ambulance Teams (AT), Police Forces (PF), Fire Station, Ambulance Centre and/or Police Office. Figure 2.2 presents a screen shot of the viewer from a sample simulation execution. Agents here are represented as circles of a

[2]For detailed information regarding the specific protocol used during simulation see [1]

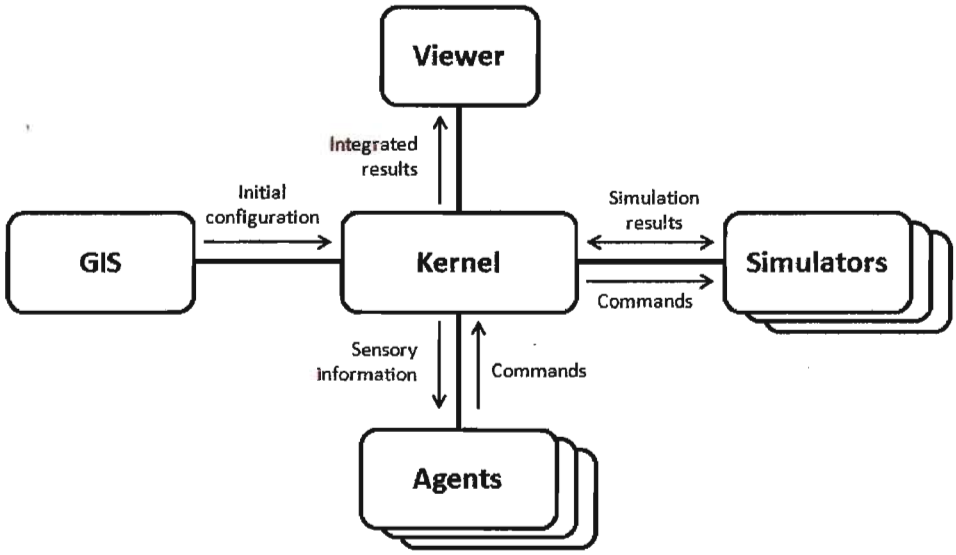

Figure 2.1: Structure and behaviour of RCRSS modules

Figure 2.2: Example viewer display from execution of RCRSS

particular colour. Civilians (shown in green in Figure 2.2) are automated agents that will move to a refuge if not trapped or injured. Civilians trapped in buildings require assistance and will slowly lose HP until they die (black in Figure 2.2). Fire Brigade agents (red in Figure 2.2) extinguish fires.

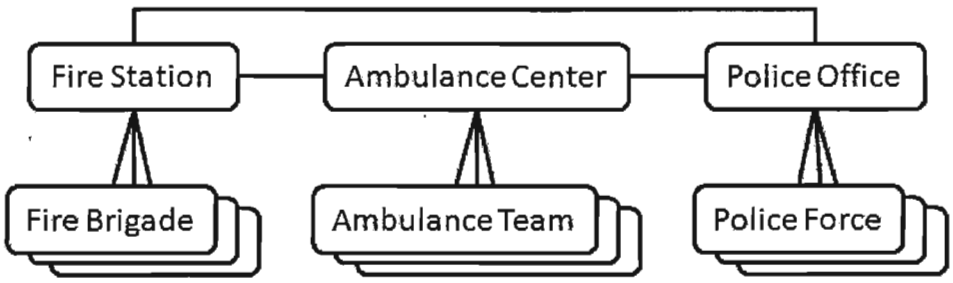

Figure 2.3: Communication links between rescue agents and centres in the RCRSS

The amount of water they can carry to do so however is limited. A Fire Brigade with no water must return to a refuge and wait a number of steps to refill. Ambulance Team agents (white in Figure 2.2) are responsible for rescuing civilians from collapsed buildings as well as transporting them to refuges. Police Force agents (blue in Figure 2.2) are responsible for clearing roads by removing blockades. Lastly, the Fire Station, Ambulance Centre and Police Office agents are buildings that act as communication hubs for their respective agent teams as illustrated in Figure 2.3.

Command	Direction	Description
Action Commands		
Move	agent → kernel	Move agent
Clear	agent → kernel	Police Force agent clears a road
Extinguish	agent → kernel	Fire Brigade agent extinguishes a building
Rescue	agent → kernel	Ambulance Team agent removes a trapped civilian from a building
Load	agent → kernel	Ambulance Team agent loads an injured civilian for transport
Unload	agent → kernel	Ambulance Team agent unloads a carried civilian
Say	agent → kernel	Agent communicates to other nearby agents
Tell	agent → kernel	Agent communicates via transmission
Sensory Information		
See	kernel → agent	Visual information near agent
Hear	kernel → agent	Auditory information near agent
Listen	kernel → agent	Transmission from another agent

Table 2.1: List of agent commands in the RCRSS

All communication between simulator and/or agent modules are presented as commands that must travel through the kernel to reach their destination. An agent wishing to perform some action sends a command to the kernel. The kernel then broadcasts each action to all of the simulators, and those with use for the command act on it and respond. The list of agent action and sensory commands is presented in Table 2.1.

As an example, consider an AT agent wishing to rescue a civilian from a nearby building. At time t, the agent receives a see command from the kernel containing information regarding the civilian,

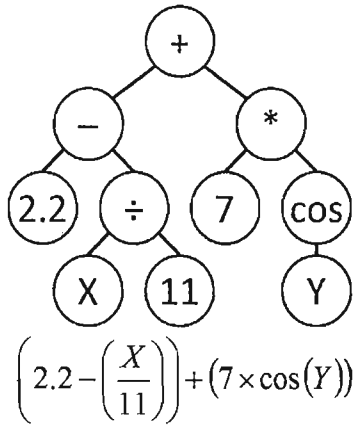

Figure 2.4: GP chromosome representation: The *genotype* refers to the tree structured solution while the *phenotype* refers to the actual formula when decoded from the tree.

including its buriedness. The AT decides to rescue this civilian and thus sends a rescue command to the kernel. The kernel then broadcasts this command. The Misc simulator, which is responsible for such things as civilian buriedness and health would then respond to the kernel stating that the civilian becomes less buried. Then at time $t+1$ the AT receives another see command containing this civilian with a buriedness less than at time t.

This description of the RCRSS has been kept brief to spare the recreation of the RCRSS manual. For a more detailed explanation of this simulation environment, see [1] and [8].

2.2 Genetic Programming

"GP is a systematic, domain-independent method for getting computers to solve problems automatically" [9]. Genetic Programming (GP) was first popularized in 1992 [3] and has since been successful in achieving human-competitive results in various fields including electronic design, game playing, searching, sorting and more [10, 9]. In GP, solutions are traditionally represented as program trees (or s-expressions), where each internal node in the tree is a problem-dependent operator and its subtrees are its operands. The leaf nodes of the tree are terminals in the expression. Together the set of all terminals and operators in the GP language are known as the primitive set of the given GP system. Figure 2.4 illustrates the *chromosome* representation of a simple formula. Here the tree, known as the genotype (or genetically represented solution), represents the formula below it, known as the phenotype (or actual solution).

The task of genetic programming is to evolve a *population* of programs that solve a given problem. This is done using the concepts of natural evolution found in the Evolutionary Algorithm (EA)

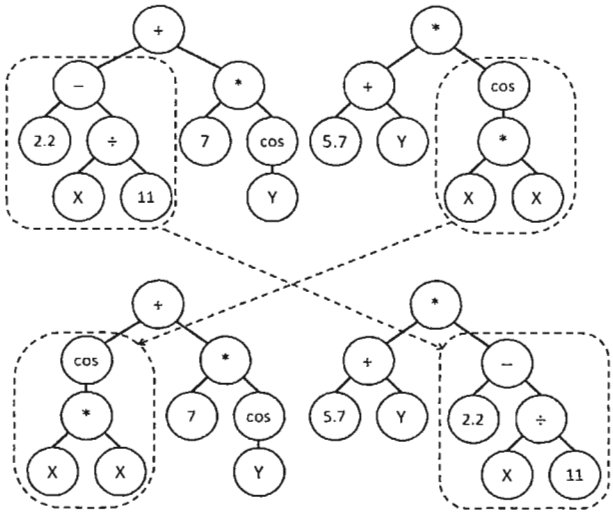

Figure 2.5: Example subtree crossover: Subtrees in each of the parents (top) are swapped to create new offspring (bottom)

framework. In EAs and in GP an initially random population of solutions to a problem is evolved by iteratively applying reproduction and selection operators.

The standard GP reproduction operator is a crossover operator known as the subtree crossover. As depicted in Figure 2.5, offspring are created by exchanging the subtree of one parent for the subtree of another, while ensuring syntactic and program correctness. The standard GP mutation, known as subtree mutation, works in the same fashion. One parent is selected for mutation and subtree crossover is performed with a randomly generated second parent. A third operator, known as cloning (sometimes called reproduction), can be used in place of crossover or mutation. In cloning an individual is simply copied over from one generation to the next.

With a new population of programs instantiated by the reproduction operators, the individual programs are evaluated on their performance at solving some problem based on a problem-dependent fitness function. The resulting fitness scores are the basis for the subsequent selection operation.

A common selection operator used in GP is tournament selection. In tournament selection, a small number of individuals (typically between 2 and 7) are picked at random from the population. From these individuals the best is then selected according to its fitness score. Two such selected individuals are used for crossover, and one such selected individual is used for mutation or cloning. The iterative application of the process of selection and reproduction is intended to search the high dimensional space of possible solutions (solution space) to ultimately refine the quality of solutions in the population from a random collection of individuals to a converged population of near-optimal solutions. The crossover and mutation operators serve to create new solutions in the population thus *exploring* the solution space, while the selection operator refines the population to samples of

its superior individuals thus *exploiting* known good areas of search. There is a necessary tradeoff to be addressed between exploration and exploitation in a GP population, the equilibrium of which is problem-dependent.

Variations to each of these operators exist in the literature, but are not used in this thesis. For information and references on extensions to the simple GP system refer to [9].

2.3 Related Work

2.3.1 Improvements to Genetic Programming

A large number of useful techniques and paradigms exist which have enhanced the performance of certain algorithms. This section presents several of such techniques which are of interest to this thesis.

A common technique used in evolutionary algorithms known as *co-evolution*, describes a situation in which the fitness of two or more individuals are dependent on each other in some way. In nature it is an understood fact that the evolution of organisms occurs in tandem with that of the organisms in its environment. For example, bees and flowers have co-evolved to dependency. In the simulated evolution of GP (and similar heuristics), it is sometimes beneficial or even natural to express such dependencies. Problem environments involving multiple agents can be simplified by the use of multiple individuals from the GP population acting and thus being evaluated together. Two forms of co-evolution exist in literature: *competitive* and *cooperative*. The competitive form has been widely applied in the environments of artificial life and game playing as it provides a natural difficulty ramp of opponents [11, 12, 13, 6]. As Koza describes in [14], "one almost never has *a priori* access to [an ideal] strategy for the opposing player or the ability to perform an exhaustive test." The use of competitive co-evolution allows the environment of one individual to contain a similarly competent adversarial individual.

Cooperative co-evolution is considerably less prolific in literature and tends to arise primarily in situations where the environment dictates the collaboration of multiple individuals, and often the performance of all individuals is given a single evaluation [5, 11, 12, 4]. The RoboCup Rescue problem is one such environment where teams of heterogenous and homogenous agents exist simultaneously in the environment striving towards a common goal with intertwining tasks. While competitive co-evolution offers the challenges and benefits of a matched opponent, cooperative co-evolution imposes the strengths and weaknesses of one individual on the fitness of another. It is often difficult to determine the individual contributions to a global fitness value, leading to what is known as the *credit assignment problem* [15]. In this way one individual's prowess at a given objective may be overridden by the ineptitude of a 'cooperating' peer. In order to combat these negative effects, clones of a single individual are typically used in place of multiple individuals. Thus the fitness of a single individual is dependent on its own proficiency and its ability to cooperate with itself. An

alternative scheme proposed in [16] is to use heterogenous individuals consisting of multiple agents per individual. This way, the agents making up a team will evolve together, removing the need for any specific credit assignment beyond the team's overall evaluation.

A second improvement technique is Automatically Defined Functions (ADFs) [17, 18, 19, 9]. ADFs are a technique used to modularize the automatically generated code of a GP solution to more closely resemble the human concept of program organization. These are implemented as separate trees from the solution tree, given their own constraints and GP language subset, but are evaluated and evolved along side the main solution tree. In terms of the GP individual, an ADF can be used in the same way as any other terminal primitive. This allows GP to reuse blocks of code without the need to spontaneously evolve them multiple times. The ability of ADFs to call other ADFs as well as recursive ADF calls leads to a potential for hierarchical structuring of solutions, which has been suggested as an effective means to automatically solve complex problems. The drawback, however, is that realm of each ADF must be specifically defined in terms of parameters, language, and interconnection. The ideal set of this information is often unknown *a priori* and can be difficult or expensive to search for. Architecture altering operations are an attempt to address this issue by dynamically evolving these attributes, however additional evolution is required to do so [20, 18].

Similar to the notion of ADFs is the Adaptive Representation (AR) paradigm [21]. While with ADFs functions are evolved as part of an individual and evaluated along-side the individual, here functions are actively sought out and constructed. By using a combination of domain-specific knowledge and statistical data gained from the GP population, small blocks of code are identified as fit candidates and generalized into new functions. The recursive application of this approach gradually raises the abstraction level of the function set thus narrowing the search space over time.

Stone *et al.*'s Layered Learning [22] is a technique in which a complex problem (such as creating soccer-playing intelligence) is hierarchically decomposed into simpler sub-problems or layers (such as intercepting a pass, evaluating a pass's success, selecting an appropriate target for a pass, etc). Each layer solves one problem which lends itself to the solution or simplification of the succeeding layers. The use of layered learning is appropriate for situations in which a direct solution from inputs to outputs is intractable. As a general framework, Layered Learning does not dictate how a problem is to be decomposed or the specific machine learning algorithm to be used at each layer. The examples provided in [22] use a different algorithm for each layer (Neural Networks, C4.5, and Reinforcement Learning respectively for the examples mentioned above).

A Layered Learning variant of Genetic Programming was applied to a simplified soccer sub-problem by Gustafson [23]. In that approach a standard single-layer GP that aimed at minimizing passing turnovers was compared to a two-layered GP that first sought to maximize passing accuracy before minimizing turnovers. In the layered learning GP, the population from the final generation of the first layer was the initial population of the second layer. The only difference to the actual GP system between layers was the fitness function used. The results found that both GP variants were

capable of finding good solutions while the layered learning GP converged much quicker.

Zhang *et al.* [24] describe a similar technique known as Fitness Switching. Here, a complex problem is broken down into n sub-behaviours, each of which is associated with a fitness that motivates that behaviour. The GP individuals are structured with n subtrees, one for each behaviour/fitness. In the sequential variant of Fitness Switching, each subtree is evaluated in turn by separate GP executions resulting in a situation similar to Layered Learning. Zhang *et al.* argue, however, that a co-evolutionary variant of Fitness Switching is superior to both sequential Fitness Switching and standard GP. In this variant a single GP execution is used to evolve all sub-behaviours simultaneously.

This thesis deals with the use of Genetic Programming to solve a difficult problem within limiting constraints. This necessitates the use of improvement techniques such as those described in this section. This thesis uses the term *hierarchical evolution* as an umbrella term to describe those techniques that construct complex solutions out of simpler ones. The hierarchical evolution techniques described above in this section were considered as extensions to the GP system developed for this thesis. A number of techniques which would require additional evolutionary time were discarded as infeasible in this instance due to the strict time constraints and already high execution time. Similarly techniques which employ problem decomposition were omitted from consideration since alterations to the RCRSS were neither simple nor desirable alternatives. Fitness switching was attempted, the description of which is presented in Chapter 4.2. Ultimately a new form of hierarchical evolution was developed that draws on features of many of these techniques. This technique, called Step-wise learning, employs the stages of goals found in fitness switching and layered learning, it uses the search-space simplification found in problem decomposition without altering the problem, and provides the search space direction found in seeded populations without programmer bias. The specifics of the Step-wise learning technique are described in Section 3.4.

2.3.2 Evolving Cooperation in Multi-Agent Systems

Panait *et al.* [25] classify learning in cooperative multi-agent systems into two broad categories: *team learning* and *concurrent learning*. In team learning a single learner is trained to control a team of agents. This may lead to a larger search space as the single learner must account for interactions within the team. The team of agents may be: 1) *homogenous*, where a single behaviour is learned and cloned to multiple agents, 2) *heterogeneous*, where multiple behaviours are learned by a single learner one for each agent, or 3) *hybrid* where the team is separated into heterogenous "squads" of homogenous agents. Typically this is a choice determined by the amount of specialization required by the environment. In concurrent learning multiple learners are co-evolved to control multiple agents, usually one learner per agent. This has the effect of reducing the individual search space by dividing it amongst the learners, but contrarily complicates the search space by adding additional learners to the environment. A similar tradeoff identified in concurrent learning is that it enhances

the ability to create specialized agents, but it requires consideration be paid to the credit assignment problem.

In [5] Luke *et al.* investigated variations of breeding rules for populations of cooperative predators in a predator-prey simulation. Three types of populations were developed, a population of homogeneous individuals where multiple predators are made from a clone of a single GP individual, and two populations of heterogeneous individuals where each GP individual is made up of four ADFs each representing a single distinct predator. Of the two homogeneous populations, individuals in one are allowed to interbreed freely between ADFs, and in the other ADF_1 was bred specifically with ADF_1, ADF_2 with ADF_2, etc. It was concluded that the latter - the restricted interbreeding heterogenous population - evolved more rapidly. In addition it was determined that agents utilizing a more direct form of sensing between each other outperformed lower-grade or more general sensing techniques, suggesting the use of direct communication (as seen in RCR) as a distinct advantage for cooperating agents.

Raik *et al.* [26] applied genetic programming to the evolution of volleyball playing agents, where agents were tasked with selecting the correct type of shot and direction for a given situation. Several stages of development were noted ranging from random to perfection. Initial applications of cooperation (passing the ball) were seen as detrimental to fitness as it led for more opportunities for error, though ultimately the perfect strategies were developed which used the necessary amount of cooperation.

Panait *et al.* [4] utilize genetic programming in the discovery of ant foraging behaviour rules. In that work cooperation is central to the task of finding a path between a food source and a nest, and arises as a necessity of the pheromone-based decision/communication model used by ants.

Yong *et al.* [27] studied the co-evolution of neural networks in a predator-prey domain. A comparison was performed between a centralized approach where all predators are centrally controlled versus an autonomous approach where each predator is controlled by a network evolved from a separate subpopulation. Additionally a comparison between communicating versus non-communicating agents was performed, where communication here is defined as knowledge of the relative locations of each peer. Several subtasks of increasing difficulty are presented to the evolving populations. It was found that non-communicating heterogenous teams evolved quicker and more effectively over each task the than the other techniques considered. The cooperation developed is attributed to stigmergy; that is, indirect coordination based on "role-based responses to the environment" (e.g. the pheromone communication of ants). Thus each subpopulation converged to specific roles due to cooperative co-evolution.

Iwata *et al.* [28] define cooperation in multi-agent systems in five ways relating to teamwork theories and coordination theory. From this they extract ten specific definitions of cooperation in the RCR simulation. Their analysis compares and contrasts the various definitions of cooperation amidst the competitors of RoboCup Rescue 2006, but does not relate them to any other means of

fitness evaluation to determine the utility of such definitions.

2.3.3 RoboCup Domain

Kummeneje [29] examines the applicability of the RoboCup domain to research, education, and the dissemination of technological advances. While primarily focussed on the RoboCup Soccer (RCS) domain, the paper concludes that RoboCup provides a wide array of challenges suitable for a diverse field of researchers. Moreover, the uncertainty and thus the complexity of the RoboCup Rescue system is far greater than that of RoboCup Soccer, so the challenges provided by RCR make it an even more suitable scenario for research.

The feasibility study of this thesis, at least from a coarse-grained perspective, is largely based upon the research done using GP in the RCS league. Although the soccer domain is simpler than the rescue domain, requiring a homogenous set of agents to achieve a single goal, a number of similarities exist. For instance, both domains consider teams of cooperating agents ideally coordinating in a dynamic environment with limited awareness. The fact that GP has been applied successfully to this domain illustrates promise for the RCR domain. Luke *et al.* [11, 30] considered RCS to be a very difficult problem for GP, but agree that there is an attractiveness to having solutions to such a difficult problem created automatically. Amongst the many challenges faced in their project, they report time being the most difficult. To reduce the amount of time spent evolving solutions they use a number of techniques including co-evolution, more expressive (higher-level) functions in the GP language, and highly-parallelized executions. An interesting note was that a multi-facetted fitness function was found to be less effective than a simple difference of goals function. In 1997 Luke *et al.* presented their evolved soccer players at the robocup soccer competition. These evolved agents defeated two human-coded soccer teams, suggesting the utility of GP in this domain.

Wilson [31] attempted several experiments utilizing low-level GP primitives to evolve soccer-playing strategies. An initial experiment which mapped soccer-simulator given perceptions to actions with very few additional functions was unable to find any meaningful combinations of logic. A second experiment, which attempted to seed the population with hand-coded solutions, found that the population quickly converged to those solutions. A final experiment increased the abstraction level slightly and managed to evolve simple chasing behaviours.

Aronsson's application of GP to the Robocup Soccer domain was met with reasonable success [32]. By limiting the search space via a small function set and hand-coded portions of the agent's logic, Aronsson evolved both coordinated defensive and offensive behaviours. Time and computational constraints as well as a necessarily small population size are cited as major issues in this domain.

2.3.4 RoboCup Rescue Team Description Papers

As an open source competition, all participants in the RCR league submit not only the source code of their work, but also each must provide a Team Description Paper (TDP) explaining their approaches

and contributions. Thus, a wealth of information is available from past competitors. Kleiner *et al.* [33] form the winning team of RCR 2004 known as *ResQ Freiburg*. This team contributed approaches to many of the subproblems found in RCR. Perhaps most notable is their solution to the problem of efficient path planning known as *longroads*. Here, multiple small segments of road in the simulators road graph are connected into one single longroad which has no inner crossings. This considerably shrinks the graph on which agents must plan their movements. In addition to this the ResQ Freiburg team studied the lifetime expectancy of civilian agents trapped in buildings. This information was then used in a genetic algorithm to select the best sequence of trapped civilians to rescue.

Paquet *et al.* [34] formed the DAMAS team, placing second in the 2004 RCR competition. Their main contribution was the development of a 'selective perception' learning method. This offline learning technique forms a decision tree that serves to reduce the size of the search space when a FB agent needs to decide which building to extinguish. The technique uses a reward system similar to the Q-values in reinforcement learning.

Habibi *et al.* [35, 36, 37] forming the *Impossibles* team have competed several times in RCR competitions, most notably achieving first place in 2005 and second place in 2007. They have used a number of techniques including fuzzy logic, auction heuristics, emotional decision making, Breadth First Search, Dijkstra's algorithm and reinforcement learning for path finding, as well as the manual modelling of trends to aid the prediction of civilian life expectancy.

Martinez *et al.* [38] used an evolutionary reinforcement learning strategy called XCS to evolve decision rules for determining the number of ambulances required to rescue a victim based on a attributes such as the victim's health points, buriedness and the current world time. In addition this paper used the longroads mechanism from the ResQ Freiburg team [33], a next victim selection algorithm similar to the fire selection presented by the DAMAS team [34], as well as several hard coded decision rules for things such as rubble clearing and fire extinguishing. The agents designed in [38] were compared to the results from the 2004 RCR competition and it was concluded to have outperformed most.

While a sizeable volume of TDP's exist covering a wide array of techniques applied to RCS, none appear to have utilized GP for the development of rescue behaviours.

Chapter 3

Genetic Programming System for RoboCup Rescue

This chapter describes the design and implementation details of the genetic programming system developed and applied to the RoboCup Rescue Simulation System in this thesis. An overview of the developed system is illustrated in Figure 3.1. There are three main components to the system: ECJ, RCRSS, and app.Rescue. ECJ and RCRSS are predefined systems adapted for use in this thesis and are described in more detail in Chapter 2. The implementation contributions of this work are in the app.Rescue component and are described throughout the remainder of this section following a brief description of the modifications made to the ECJ and RCRSS components.

Evolutionary Computation in Java (ECJ)[1] is the GP system used in this thesis. RCRSS[2], as described in Section 2.1 is the problem being solved by the GP system. Both of these systems are provided freely by their authors, but have been modified for use in this thesis. ECJ was modified to allow for threaded evaluation of cooperative individuals from three separate subpopulations and to read entire populations from file. The RCRSS system was altered to allow for multiple simultaneous simulations to run on a single machine. In order to hasten execution times the standard 10-minute wait time to shut down simulators was replaced with a broadcasted exit message. Further, the 1-second wait for agent commands at each step of simulation was changed to be a *maximum* 1-second wait, meaning the simulator would proceed sooner *iff* all agent commands (and communications) were presented in less than 1-second. These alterations sped up the RCR simulation (and thus the GP evaluation) dramatically without altering the problem definition.

[1]ECJ available online: http://cs.gmu.edu/ eclab/projects/ecj/
[2]RCRSS available online: http://sourceforge.net/projects/roborescue/

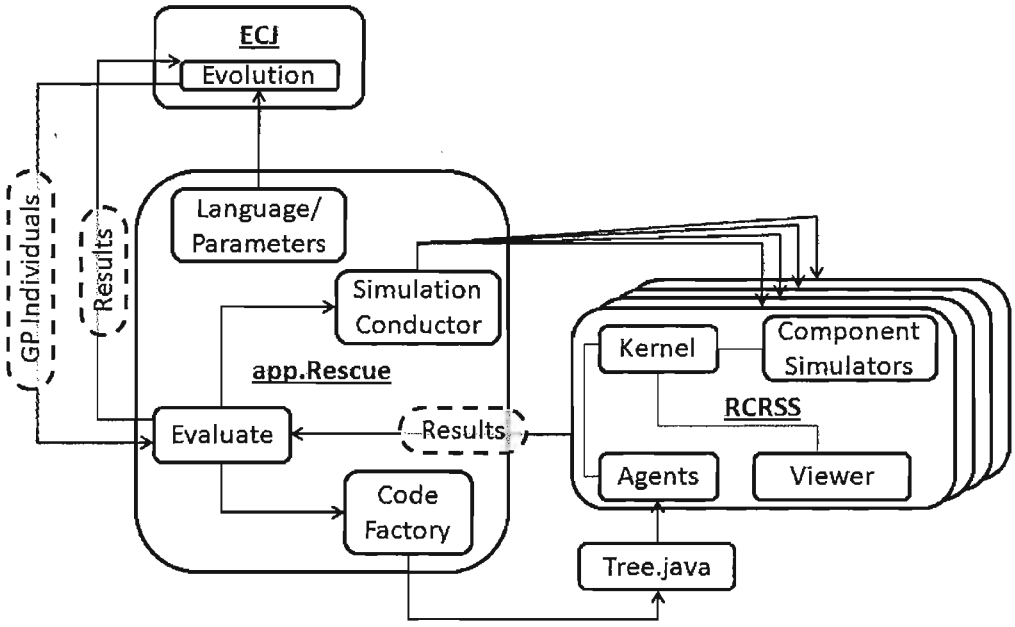

Figure 3.1: Overview of the developed RCR-GP system. Three main elements are the ECJ system, the app.Rescue directory, and the RCRSS

3.1 Design and Implementation

The implementation within this thesis is primarily concerned with combining the ECJ GP system with the RCRSS problem domain. To this end, app.Rescue is a custom directory within the ECJ system that takes care of all of the intermediary tasks required to combine these systems. This includes defining the parameters and language, managing multiple RCRSS executions in parallel, and automatically converting GP trees into compiled java class files on the fly for subsequent use as agent control structures in the RCRSS. This section describes the design of the population structure (a set of parameters to the ECJ system), and the evaluation of a single GP individual (in terms of its conversion into RCR agents). Discussion of the GP language is deferred to Section 3.2, and of the experimental parameters is deferred to Chapter 4.2.

The first attempted variant of the GP population structure consists of a single population as illustrated in Figure 3.2a. Each individual in the population is made up of three trees, one for each of the three autonomous agent types (AT, FB, and PF). Multiple agents of the same type all use the same GP tree during simulation (e.g. all ATs use the same decision logic). Distinct trees are necessary for different types of agents as their tasks vary greatly. Yet, utilizing homogeneous individuals during simulation simplifies the evolution process by only needing to evolve one AT, one FB, and one PF. Past research [11] has shown that homogeneous teams of agents evolved more

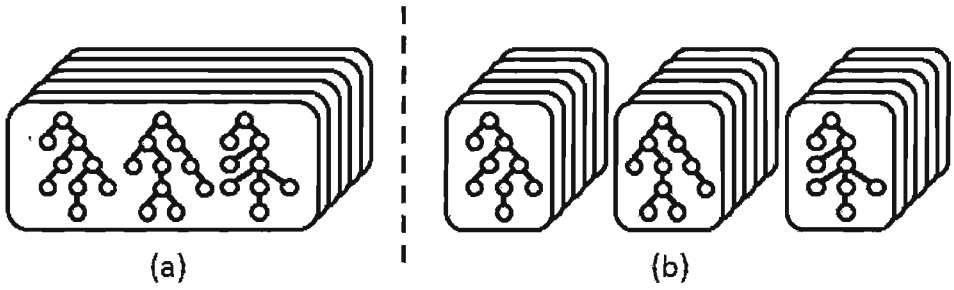

Figure 3.2: Implementation of two forms of population. a) Each individual contains the trees for all three types of agents. b) Three separate sub-populations exist, each containing the tree for a single type of agent.

readily than heterogeneous ones, so this strategy is ideal for this problem with such limiting time constraints. All three sub-trees are evaluated on their combined performance, and the GP individual will succeed or fail based on the overall proficiency of the entire individual. In the terminology defined in [25] (described in section 2.3), this system uses a hybrid team learning approach where a single learner (GP individual) controls heterogeneous squads of homogeneous agents.

A second GP system (Figure 3.2b) is developed which uses the concurrent learning approach described in [25], except it does so again with squads of homogeneous agents. In this system, three sub-populations are co-evolved. One individual from each population is selected for evaluation. Multiple clones of this individual are used to fill up teams of similar agents (as with above). There is an inherent credit assignment problem associated with the concurrent learning approach, as in this case three unrelated individuals would still share a single simulation score. The advantage to using this approach in this problem, however, is that each individual can be given a separate score. Thus the fitness of each individual is changed to be task-dependent. That is, the AT tree is evaluated based on the amount of civilian health points remaining at the end of simulation, the FB tree is evaluated based on the area of unburnt buildings, and the PF tree is evaluated on the number of cleared blockages. In this way, an effective tree for controlling one type of agent will not be overly hindered by having its score lumped together with an unfit tree controlling a different agent. Cooperative co-evolution is still in effect in this GP system as the agent tasks are still very intertwined (e.g. a burning building will quickly harm civilians).

Evaluation of a GP individual in this system consists of a single execution of the RCRSS. The trees that makeup a GP individual are passed into the 'Code Factory'. This module parses three GP trees (one for each type of agent) then produces and compiles a java class with three methods (one for each tree). Trees in the java files are in the form of nested method calls to the executing agent (except for the if statement which is converted to its ternary syntax), this prevents the needs to parse a GP tree at every use. When an agent is created in the RCRSS it is assigned a specific

tree according to the GP individual and generation numbers. When an agent is prompted to act at each step, it will call the method appropriate to its type from the tree file it has been assigned.

Each RCRSS execution runs for 300 simulation steps, taking a maximum of 1 second each, for an average execution time of approximately 2.4 minutes (after the modifications mentioned above). Executions utilizing fewer steps are unacceptable due to the drawn out nature of the problem. Many objectives cannot be completed quickly or even reached initially, and there is a growing loss of score and increased difficulty to the problem as time progresses. Since a complete 300 step RCRSS execution must be completed once for every individual in the GP population for every generation in the GP system's execution, these times quickly add up to very long experimentation times. In order to combat this, small population sizes were used and multiple evaluations are done in parallel. A typical experiment involving 50 individuals for 50 generations requires about 100 hours of execution divided by the number of parallel evaluations (1, 2 or 4 were used in this thesis depending on the capability of the computer being used).

Within each RCRSS execution there exist a number of rescue agents (in the default map setup there are 24 agents and 3 centres). Habibi *et al.* [36] describe two main approaches to implementing RCR agents: centralized and distributed. In the centralized models, the AT, FB and PF agents send all their information to the central hubs (i.e. the ambulance centre, the fire station, and the police office) and wait for instructions. In this paradigm the responsibility of decision making is entirely placed in the central agents. This does, however, have the natural benefit of coordinating all of the non-central agents. In the distributed model, each individual agent is responsible for making its own decisions based on the information its given. In this model, the central agents act only as relays for information such that every agent has a similar view of the world. This thesis operates using the distributed model, since this model more naturally lends itself to the design of a low-level decision logic.

A single agent is comprised of two parts (see Figure 3.3). The first part of the agent is concerned with the storage and communication of a world model. This part is manually coded. Put simply, the agent will remember any new or changed objects in the environment, and if they are important (e.g. a civilian discovered, or a building's fieryness changed) then it will communicate the new or changed object. The second part of the agent is concerned with the decision and action processes. This takes the form of a Genetic Programming tree utilizing language primitives that deal with the world model (from the first part of the agent) and action commands to the kernel. The specifics of the language are discussed later in this chapter. Also illustrated in Figure 3.3 are the three centres. As described earlier, these act as communication hubs. As such, they consist of only the world model and communication aspects of the agents. This component differs from the autonomous agents' in that there is a lack of visual and auditory sense in the centres. All new information to the centre 'agents' is provided via communication from agents.

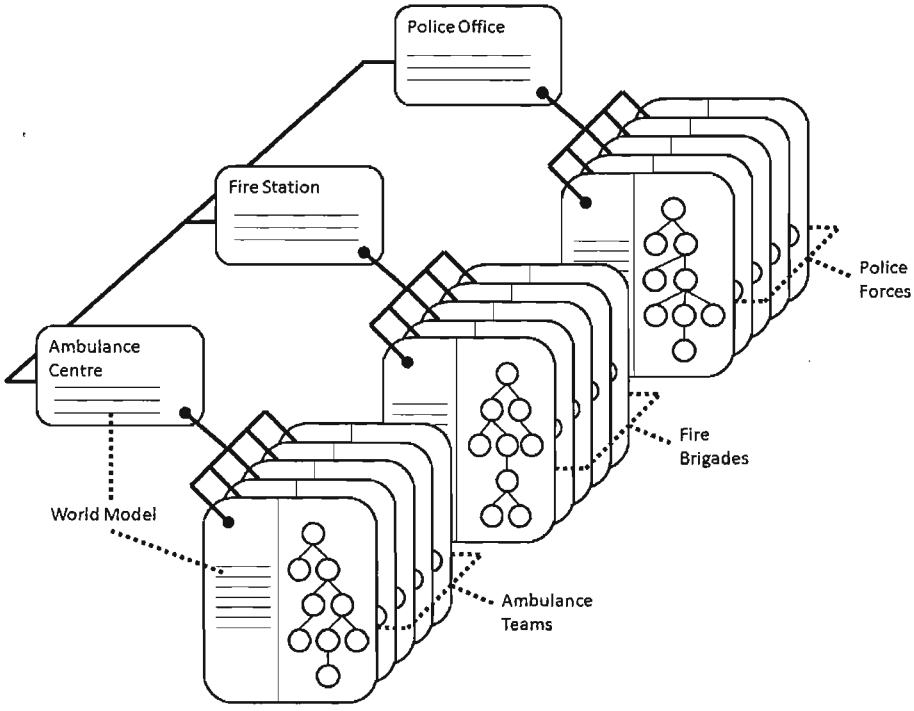

Figure 3.3: Structure of agent simulator module implementation. Each agent is made up of an evolved tree that controls its decisions and actions, and manually programmed memory and communication. Agents of the same type will share copies of the same tree, while agents of different types will have differing trees. Bold lines indication bi-directional communication links

3.2 GP Language

Perhaps the most fundamental part of the setup of a GP system is the design and implementation of the GP language. The language determines not only the expressiveness of the system, but also the solution/search space of the problem. If one creates too simple of a language the GP system may be unable to describe meaningful solutions to the problem. Conversely, a language that is too complicated may create too large of a solution space for evolution to search. Koza discusses the idea of *sufficiency* in the terminal and function sets of a GP language in [3], stating that "the sufficiency property requires that the set of terminals and the set of primitive functions be capable of expressing a solution to the problem." Although there is no general means for determining sufficiency, it must be given consideration.

As an automatic programming technique, the ideal of genetic programming is to minimize the effort of the programmer in solving the problem. By this reasoning, the ideal language would be one that arises naturally from the problem itself. In the case of using GP for simulated agent behaviour,

a natural language could be the various aspects of sensory input given to the agent, and simple functions that tie sensory input to actuators. A low-level approach such as this, however, has been found to be more difficult for the evolutionary process to handle than combinations of higher-level primitives. This difficulty is commonly attributed to the large gap between the low-level building blocks and the evaluation of high-level solutions. This serves to illustrate the importance of the GP language, in that ultimately the design of the GP language can determine whether the task at hand is within GP's capabilities.

When designing any GP language, there are several complicating factors. The first issue in designing the GP language is the importance of providing ample functionality without dictating strategy. That is, the primitives used in the GP language must be capable of combining into meaningful solutions, but themselves, should not limit or overtly bias the search. This concept arises from the idea that one of the primary motivations for applying GP to difficult problems is its potential for discovering unexpected solutions. To address this issue the language developed in this thesis aims to provide as much of the environmental knowledge as possible without colouring it. Primitives that abstract lower-level knowledge are either natural abstractions (such as identifying a fire as opposed to a building with a large fire attribute), or are part of a broad set of abstractions (such as the nearestBuilding primitive as part of the set including farthestBuilding, largestBuilding, randomBuilding, fieryestBuilding, mostBrokenBuilding, etc). The second challenge faced when designing a GP language deals with the efficiency of the language. Manual coding of a solution offers the programmer the opportunity to optimize the overall execution of the program. Whereas constructing solutions from smaller pre-made modules (primitives) makes this considerably more difficult. While an individual primitive may be optimized to efficiency (e.g. Using quickSort as a sorting primitive), the combinations of primitives is somewhat unpredictable by design, and thus difficult to optimize (e.g. Multiple potentially redundant uses of the sort method would be unnecessary in a manual program, but potentially unavoidable in a GP constructed solution). This complication is particularly relevant in this thesis due to the strict time constraints imposed on each execution of the solution trees and the representation of sense and memory as lists of objects.

The language used in this thesis consists of functions that describe aspects of the environment (sensors), actions that can be used within the environment (actuators), and some logical constructs to enable meaningful decision-making. In addition, there are some primitives added to this language that simplify this scheme for the sake of evolvability. For example, the at_Fire primitive serves one single function: to determine whether a given Fire Brigade is close enough to extinguish a building. So rather than providing a language with a Max_Extinguish_Range, a Distance, and a LessThan primitive and hoping that evolution learns to combine them properly and in the right context, a single primitive (at_Fire) is provided that returns a boolean value describing if the Fire Brigade is in range of the fire.

In order to maintain a meaningful structure in the logic trees, this thesis uses a strongly typed

Symbol	Name	Description
Atomic Types		
A	Action	Root type, the actions an agent can take
B	Boolean	True or False
T	Target	Any object in the simulation environment
L	List	A list of non-specific-typed elements
Subtypes of T		
T_{agent}	Agent	Any AT,FB, or PF
T_{build}	Building	Any building (includes fires and refuges)
T_{civ}	Civilian	Any Civilian
T_{road}	Road	Any Road (blockaded or not)
Subtypes of L		
L_{agent}	List of AT agents	A list containing only agents
L_{build}	List of buildings	A list containing only buildings
L_{civ}	List of civilians	A list containing only civilians
L_{road}	List of roads	A list containing only roads

Table 3.1: Types in the GP language

GP language [39]. This means that nodes in the tree have return types, and expected child types. Table 3.1 lists the various types used in this language.[3] The root of the tree returns an Action type, and often times action-type primitives require a Target-typed child. The language is designed such that terminal nodes return a list of some type of Target, as is the natural setup for the problem. The list is then potentially refined to only the 'important' targets and the 'best' is then selected from the list, where 'important' and 'best' are generic terms to be defined by the specific primitives used. Much of the remaining structure of the logic trees is made up of if-statement primitives which are controlled by environmental information in the form of boolean terminals and functions. The complete listing of the language used is presented in Tables 3.2 through 3.9, sorted and described by topic in the remainder of this section.

Actions

The action primitives (see Figure 3.2) are modelled almost directly on the actions available to the RCR agents during simulation. A divergence occurs for the sake of evolvability at the point of deciding on a target. For the primitives Load, Remove (known as Rescue in the simulator but renamed for clarity), and Clear, the targeting is done automatically if a suitable target is within

[3]ECJ version 19 (used here) provides atomic-types and set-types. Set types can be used similar to subtyping so long as precautions are taken. For instance, the mechanism of set typing, allows the use of any of the set-type elements in place of the set-type itself or vice versa. So a node expecting a Target argument could instead use a T_{agent}. However, a primitive that returns Target can also be used in place of an expected T_{agent}. This can lead to unsafe typing as a T_{road} could be used in place of a Target which is then used in place of a T_{agent}. The rule used in this thesis to counteract these effects is that no primitives in the language return the general set-type (ie. Target is only ever used as a parameter's type, never a return type).

Action	Returns	Description
if(B,A_1,A_2)	A	If B evaluates true then return A_1 else return A_2
move(T)	A	Move the agent to target T
moveB(T)	A	Move the agent to target T, ignore blockades
clear()	A	Remove a roadblock if within reach (PF only)
extinguish(T_{build})	A	Extinguish fire at target T (FB only)
load()	A	Load civilian if within reach (AT only)
unload()	A	Unload a carried civilian (AT only)
remove()	A	Rescue a trapped civilian (AT only)
rest()	A	The agent does nothing this turn

Table 3.2: Action primitives in the GP language

range. This was not acceptable for Extinguish as often multiple targets are in range, where one may be more desirable than another. For the Move primitives, the targets are selected via evolution, but the path to the target is determined via an A* search [40]. The default Move operation attempts to find an unblocked path to the goal, while MoveB returns the straightest path despite blockades. This was found beneficial during initial testing to make PF agents clear main pathways. If an agent's chosen action for a turn cannot be completed it is ignored for a turn, equivalent to that agent performing a Rest action that turn. Finally, the if-statement is included in this section as it is used to decide between two Action-returning subtrees. This is the source of branching in this language, enabling decision making.

Environment

There are two main sources of environmental information in this language, stemming from two main types of information. Information about objects in the environment are stored in and accessed as lists of specific types. Information about conditions or states of the environment are queried via functions that return boolean values. The listings of primitives providing these types of information are found in Tables 3.3 and 3.4 respectively.

The list primitives are mainly comprised of full collections of single object types, such as allBuildings or allBlockades. Some of these collections are of generic objects such as buildings or roads, while others are collections of more specific subsets such as fires (buildings that are burning), or blockades (roads that are blocked). This was a simplification provided to the GP language to enhance evolvability. A few other list primitives were developed to utilize the interconnectedness of the map.

Boolean environmental primitives come in various styles. Primitives beginning with 'any' are queries about the agents global awareness. Primitives prefixed with 'at' are examining the agent's immediate surroundings. Primitives starting with 'has' are queries of other agents, which are hoped to provide a level of intentionality to cooperation. Additionally, some boolean primitives are designed

Primitive	Returns	Description
allAgents()	L_{agent}	Returns a list of all Agents
allATs()	L_{agent}	Returns a list of all Ambulance Teams
allBlockades()	L_{road}	Returns a list of all Blockades
allBuildings()	L_{build}	Returns a list of all Buildings
allCivs()	L_{civ}	Returns a list of all Civilians
allFBs()	L_{agent}	Returns a list of all Fire Brigades
allFires()	L_{build}	Returns a list of all burning Buildings
allPFs()	L_{agent}	Returns a list of all Police Forces
allRoads()	L_{road}	Returns a list of all Roads
allRefuges()	L_{build}	Returns a list of all Refuges
allTrapped()	L_{civ}	Returns a list of all trapped Civilians
getEntrances(T_{build})	L_{road}	Returns all roads with access to T_{build}
getAdjacentRoads(T_{road})	L_{road}	Returns all roads connected to T_{road}
getAdjacentBuilds(T_{road})	L_{build}	Returns all buildings accessible from T_{road}

Table 3.3: Environmental List Primitives of the GP language

Primitive	Returns	Description
anyBlockades()	B	Returns true if there are any known blockades
anyFires()	B	Returns true if there are any known fires
anyTrapped()	B	Returns true if there are any known trapped civilians
atBlockade()	B	Returns true if the agent is currently near a blockade
atFire()	B	Returns true if the agent is currently near a fire
atInjured()	B	Returns true if the agent is currently near an injured civilian
atRefuge()	B	Returns true if the agent is currently in a refuge
atTrapped()	B	Returns true if the agent is currently near a trapped civilian
hasAgent(T_{agent})	B	Returns true if T_{agent}'s current agent pointer is not null
hasBuild(T_{agent})	B	Returns true if T_{agent}'s current build pointer is not null
hasCiv(T_{agent})	B	Returns true if T_{agent}'s current civ pointer is not null
hasRoad(T_{agent})	B	Returns true if T_{agent}'s current road pointer is not null
isNullTarget(T)	B	Returns true if T is null
isLoaded()	B	Returns true if there is a loaded civilian on this AT
isEmpty(L)	B	Returns a true if list L is empty
pathExists(T)	B	Returns true if a path exists to T from current position
waterIsEmpty()	B	Returns true if this FB's water tank is empty
waterIsFull()	B	Returns true if this FB's water tank is full
waterIsUnderHalf()	B	Returns true if this FB's water tanks is less than 50%

Table 3.4: Environmental Boolean Primitives of the GP language

as potential checks for the results of other primitives. IsNullTarget, and isEmpty both provide the agent with a means of verifying that some level of target selection was effective, while pathExists will verify if a Move operation is possible. The remaining primitives (isLoaded, waterIs...) are agent-specific checks that allow the agent to verify its own condition.

List Refiners

In order to make use of the large collections of environmental objects, the agents are provided with a wide array of type-specific list-refining primitives. These primitives are a simple way for the agent to reduce the size of potential targets based on very specific properties of the targets. There is however, a large potential for error while using these primitives as multiple overlapping uses can either be redundant or simply empty out the list. A reasonably small sized depth limit should counteract this effect. The collection of list refining primitives developed for this thesis is presented in Figure 3.5.

Primitive	Returns	Description
buildBurning(L_{build})	L_{build}	Removes all non-burning buildings from the given list
buildExtinguished(L_{build})	L_{build}	Removes all burning buildings from the given list
buildFieryness^f(L_{build})	L_{build}	Maintains only those buildings with fieryness values of f
buildHasCiv(L_{build})	L_{build}	Removes all buildings that don't contain civilians
buildInRange(L_{build})	L_{build}	Removes all buildings outside of extinguish range
buildIsExplored(L_{build})	L_{build}	Removes all unexplored buildings
buildIsUnexplored(L_{build})	L_{build}	Removes all explored buildings
civDead(L_{civ})	L_{civ}	Removes all living civilians from the given list
civNearDeath(L_{civ})	L_{civ}	Removes all civilians with more than 24% health
civInjured(L_{civ})	L_{civ}	Maintains only civilians with health in the range 25%-49%
civWounded(L_{civ})	L_{civ}	Maintains only civilians with health in the range 50%-74%
civHurt(L_{civ})	L_{civ}	Maintains only civilians with health in the range 75%-99%
civUnhurt(L_{civ})	L_{civ}	Removes all civilians with less than 100% health
civAlive(L_{civ})	L_{civ}	Removes all dead civilians from the given list
civBuried(L_{civ})	L_{civ}	Removes all unburied civilians from the given list
civDamaged(L_{civ})	L_{civ}	Removes all undamaged civilians from the given list
agentIsAt(L_{agent})	L_{agent}	Removes all FBs and PFs from the given list
agentIsFb(L_{agent})	L_{agent}	Removes all ATs and PFs from the given list
agentIsPf(L_{agent})	L_{agent}	Removes all ATs and FBs from the given list

Table 3.5: List-Refining Primitives of GP language

The use of lists and list refiners allows for a bottom-up approach to decision making. This seems like a more natural description of the solution in this setting as the agents must maintain awareness of all of the potential threats and act accordingly on those that require it most or that will have the greatest impact.

Selectors

Tables 3.6 through 3.9 display the vast number of potential target selection primitives. These primitives are used for extracting a single element out of a list, based on some type-specific criteria. This should allow agents to fully optimize their decision making process as tailored to whatever strategy is deemed optimal. As with any primitive, the wide range of options may have the detrimental effect of expanding the search space too wide, though it is hoped that the strategies taken during evolution

here will provide sufficient focus. Table 3.6 lists the set of general target selectors. These primitives can be used on any type of list, though in fact, separate instantiations of these primitives exist for each type. The rest of the target selectors select based on extremes of various attributes for each type of object.

Primitive	Returns	Description
farthest(L)	T	Returns the target from L with the largest distance from the agent
randomElement(L)	T	Returns a random element from list L
nearest(L)	T	Returns the target from L that is nearest to the agent

Table 3.6: General Target Primitives of the GP Language.

Primitive	Returns	Description
selBuildFieryest(L_{build})	T_{build}	Returns the T_{build} with the highest fieryness from L_{build}
selBuildLeastFiery(L_{build})	T_{build}	Returns the T_{build} with the lowest fieryness from L_{build}
selBuildMostArea(L_{build})	T_{build}	Returns the T_{build} with the most area from L_{build}
selBuildLeastArea(L_{build})	T_{build}	Returns the T_{build} with the least area from L_{build}
selBuildMostBroken(L_{build})	T_{build}	Returns the T_{build} from L_{build} that is most broken
selBuildLeastBroken(L_{build})	T_{build}	Returns the T_{build} from L_{build} that is least broken

Table 3.7: Building Target Primitives of the GP Language.

Primitive	Returns	Description
selCivHealthiest(L_{civ})	T_{civ}	Returns the healthiest T_{civ} from L_{civ}
selCivLeastHealth(L_{civ})	T_{civ}	Returns the T_{civ} from L_{civ} that is nearest to death
selCivMostBuried(L_{civ})	T_{civ}	Returns the most buried T_{civ} from L_{civ}
selCivLeastBuried(L_{civ})	T_{civ}	Returns the least buried T_{civ} from L_{civ}
selCivMostDamaged(L_{civ})	T_{civ}	Returns the most damaged T_{civ} from L_{civ}
selCivLeastDamaged(L_{civ})	T_{civ}	Returns the least damaged T_{civ} from L_{civ}

Table 3.8: Civilian Target Primitives of the GP Language.

3.3 Main Objectives of this Thesis

The main objective of the RCR competition is to develop intelligent systems to control the Ambulance Team, Fire Brigade, and Police Force agents in the RCRSS. The primary goal of this thesis is to excel at this objective using Genetic Programming by focusing on evolving the behaviours necessary. This thesis focuses not just on the ability of the agents to achieve high scores (as measured by the simulation system), but also on the specific strengths and weaknesses of GP to evolve individual behaviours. Since the overall task of performing well according to the RCRSS default

Primitive	Returns	Description
selRoadMostBlocked(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the most blockage
selRoadLeastBlocked(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the least blockage
selRoadWidest(L_{road})	T_{road}	Returns the widest T_{road} from L_{road}
selRoadNarrowest(L_{road})	T_{road}	Returns the narrowest T_{road} from L_{road}
selRoadMostOpenLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the most open lanes
selRoadMostBlockedLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the most blocked lanes
selRoadFewestOpenLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the least open lanes
selRoadFewestBlockedLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the least blocked lanes
selRoadMostLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the most total lanes
selRoadFewestLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the fewest total lanes
selRoadHghestCost(L_{road})	T_{road}	Returns the T_{road} from L_{road} with highest repair cost
selRoadLowestCost(L_{road})	T_{road}	Returns the T_{road} from L_{road} with smallest repair cost

Table 3.9: Road Target Primitives of the GP Language.

scoring formula is a complex one involving multiple dynamic objectives and constraints, it is unsuitable to evaluate the entire result on a success or fail basis. Ideally, it is hoped that agents will be evolved to rival the competence of human-created agents from past RCR competitions. However, as genetic programming is not typically designed for such difficult tasks, the objective of this work will be to evaluate the developed system's ability to evolve behaviours capable solving individual subproblems. The intent of this work was to examine each of the behaviours described below in this section, starting from scratch with minimal assumptions. However, the challenge posed in applying Genetic Programming to the development of these behaviours within the time constraints of this thesis proved the topic to be too broad. Thus this thesis focusses primarily on the *appropriateness* or *operational* behaviours, relegating the remaining behaviours to ongoing and future work.

Appropriateness

The first class of behaviours to be examined is the selection of appropriate actions. The ability of an agent to intelligently decide on the action best suited to its current situation can be described as the base-stage of the agent development. Tasks such as deciding to extinguish a nearby fire, rescue a nearby civilian, or move to an objective when not near one are among the 'simple' objectives in this class of sub-problems. Other tasks in this category, such as FB agents knowing to return to the refuge to refill their water tanks, are not as straight forward but are just as necessary. This suggests a gradient of difficulty within this class of sub-problems. This will be addressed in two steps. Step 0 aims at evolving the core structure of the agent logic. The ideal is to create agents that have intelligence enough to move to an objective and act on it. Following this, step 1 aims at having agents with logic to complete their entire complex objective. For example, an AT should be capable of finding a trapped civilian, freeing them, loading them, transporting them to a refuge and then unloading them. By the end of evolution in this class of behaviours it is hoped that agents will

be effective and efficient at the performance of their core tasks. Also, it is expected that as a result there will be a reduction in behaviours that are either detrimental or nonsensical.

Task Assignment

The second behaviour class is the allocation of resources. Before a FB agent moves to extinguish a fire, or before an AT agent moves to rescue a civilian, it ought to make an intelligent decision on which fire or civilian is the most important or efficient to act upon. Also, while each agent has a specific objective to accomplish and all must work diligently at mitigating the simulated disaster, effective exploration of all corners of the map is required in order for all agents to be aware of the current total state of the city. Thus, a proper balance between the delegation of exploration and objective action is a necessary task-assignment problem in this domain. Effective prioritization of tasks is what elevates the winning teams of past RCR competitions above the rest. This form of strategizing behaviour is clearly a step above the action selection objectives of the previous class. The expectation here is that individuals evolved to perform the simpler tasks of the previous class will be better suited for evolution of more complicated behaviours than individuals with no prior adaptation.

Cooperation

The third behaviour mechanism encompasses several forms of cooperation. Agents of different teams must cooperate in order to achieve mutual benefits in their individual objectives. Most notably, PF agents must clear roadblocks allowing FB and AT agents to reach their intended targets. Also, civilians trapped in burning buildings will take more damage than those trapped in buildings that are not lit, so cooperation is required between all types of agents. In addition to this, agents of the same type must often cooperate. Most tasks that agents perform can be hastened by the use of multiple agents acting at once. Conversely, many objectives may also benefit from the dispersion of agents to increase the total area of awareness and number of tasks being completed. Thus a balance of these two variants of cooperation may be required. Since agents of the same type will share the same GP program tree, intra-type cooperation may be considered equivalent to cooperating with oneself, while inter-type cooperation will present cooperation in a more conventional form. In general, the definition of cooperation in this thesis will be the gathering of knowledge by one agent about another agent and subsequent use of that knowledge to guide their own actions. Thus there is a notion of intention to the cooperation that is expected here. Without this one could rightly argue that cooperation is an in-built element of the RCR environment itself as all agents acting independently ultimately have the shared experience of working towards a common goal.

3.3.1 Challenges of Evolution

The above classes of behaviours specify two types of challenge that this problem poses to the developed GP system: *operational behaviours* and *strategizing behaviours*. Operational behaviours are those found in the Appropriateness behaviour class. These are behaviours that are necessary parts of the rescues agents as defined by their roles in the RCR environment itself. The remaining behaviour classes describe strategizing behaviours. These are a form of optimization in the sense that the behaviours developed in these classes aim to use the operational behaviours in the best way possible. The distinction between operational and strategizing behaviours is the presence of an expected solution. Operational behaviours are predefined by the RCRSS. For example, the complete operational behaviour of an AT agent is to rescue civilians, load them, transport them to a refuge, and then unload them. In contrast, strategizing behaviours have no predefined solutions. For example, there is no given rule for determining the best order in which to rescue each civilian.

Typically, competitors in the RCR competitions will construct complex solutions to the determination of the best strategizing behaviours while assuming the underlying operational behaviours. This thesis, on the other hand, aims to examine the suitability of GP to build solutions from the ground up. Therefore, the operational behaviours are not overlooked as an objective for evolution. This increases the challenge of automating the development of rescue behaviours, but is necessary to gain a clear understanding of GP's limitations.

In practice, determination of operational behaviours through evolution requires GP to discover the missing parts of an environment. The agent roles are predefined in the RCRSS and remain unfulfilled until GP evolves solutions that satisfy them. For the evolution of strategizing behaviours, GP must learn the best ways of utilizing the operational behaviours.

In order to examine the effectiveness of GP for evolving the controlling behaviours of RCR agents, GP should be utilized to evolve both operational and strategizing behaviours. The focus of this thesis is on the evolution of operational behaviours. The evolution of strategizing behaviours is left for future work.

3.4 Step-wise Learning Paradigm

As described in Section 2.3, *hierarchical evolution* is known to improve the ability of evolutionary algorithms by breaking down a problem into component parts and gradually combining their individual solutions. Commonly this is either done as isolated learning tasks on smaller sub-problems, by using fitness functions that emphasize particular aspects of the problem, or via the combination of primitives into higher-order primitives. Composition of primitives was thought to be too complex and time consuming for this thesis, and decomposition of the problem seemed unfeasible considering the complexity of the simulator. Use of task-specific fitness functions was considered, but were found to be insufficient to reduce the complexity of the problem.

The approach to hierarchical evolution taken in this thesis is a step-wise increment of the size of the GP language. The step-wise learning paradigm was designed here as a means of simplifying the search space as well as focusing the evolution on certain specific aspects of the RCR problem. The intention here is to initially limit the size of the search space by using a very small subset of the GP language. Once evolution has occurred on one step, then expand the language with specific primitives according to the goals of the next step. Each step (with the exception of step 0) has a seeded population that comes from the final population of the step before it. The objectives listed in the previous section are each represented here as one or more steps. In this way, the implementation of language steps are used to simultaneously address the objectives of the thesis as well as present the search space as a gradually increasing challenge for the developed system. Each step is comprised of a specific subset of the GP language that was listed in Section 3.2, as well as the collection of subsets from the steps that came before it. An exception to this is the "Cooperation" steps which are considered as a side step at each phase of the evolution to determine if cooperation evolves and if it is beneficial.

The language sets for each step are selected from the full set based on the aspects of the environment which they describe, and thus the apparent potential information they provide to solutions. Under the assumption that the entire GP language is a reasonably complete description of the environment, the primitives that make up each step are complete descriptions of a section of the environment required to promote the desired behaviours. Thus when it is said that this thesis evaluates the system's ability to evolve such behaviours, it is evaluating the system's ability to create such behaviours out of building blocks that are sufficient to do so.

Appropriateness

The appropriateness behaviour is broken down into two steps. The objective of the first step (step 0) is to establish an evolutionary convergence amidst an initially random sampling of the solution space. This is a task that proved difficult in previous experiments that used this thesis's entire GP language. This initial step in the evolution of RoboCup Rescue agents uses a core subset of the overall language as presented in Table 3.10. The primitives selected had to be a semi-complete description of the bare-bones of the problem in order to have the search space contain meaningful solutions. This step's language consists of the various actions available to the agents, access to known targets (fires, trapped civilians, blockades), simple environment queries (e.g. atFire), and an if-statement to provide structure. The goal of this step is to start from a random population and develop the decision logic structure necessary to have the agents perform the simplest form of their tasks. Since the only target selection mechanism is "nearest", well adapted agents from this step should exhibit a greedy behaviour, moving to the nearest target and acting on it.

The second half of appropriateness, step 1, was designed to extend the language from step 0 to allow a more complete description of the environment and wider range of expression in the potential

Primitive	Returns	Description
if(B,A_1,A_2)	A	If B evaluates true then return A_1 else return A_2
move(T)	A	Move the agent to target T
moveB(T)	A	Move the agent to target T, ignore blockades
extinguish(T_{build})	A	Extinguish fire at target T (FB only)
load()	A	Load civilian if within reach (AT only)
unload()	A	Unload a carried civilian (AT only)
remove()	A	Rescue a trapped civilian (AT only)
clear()	A	Remove a roadblock if within reach (PF only)
nearest(L)	T	Returns the target from L that is nearest to the agent
allBlockades()	L_{road}	Returns a list of all Blockades
allFires()	L_{build}	Returns a list of all burning Buildings
allRefuges()	L_{build}	Returns a list of all Refuges
allTrapped()	L_{civ}	Returns a list of all trapped Civilians
atBlockade()	B	Returns true if the agent is currently near a blockade
atFire()	B	Returns true if the agent is currently near a fire
atInjured()	B	Returns true if the agent is currently near an injured civilian
atRefuge()	B	Returns true if the agent is currently in a refuge
atTrapped()	B	Returns true if the agent is currently near a trapped civilian
getEntrances(T_{build})	L_{road}	Returns all roads with access to T_{build}

Table 3.10: Language Step 0: Core problem structure

Primitive	Returns	Description
anyBlockades()	B	Returns true if there are any known remaining blockades
anyFires()	B	Returns true if there are any known fires
anyTrapped()	B	Returns true if there are any known trapped civilians
rest()	A	The agent does nothing this turn
isLoaded()	B	Returns true if there is a loaded civilian on this AT (AT only)
waterIsEmpty()	B	Returns true if this FB's water tank is empty (FB only)
waterIsFull()	B	Returns true if this FB's water tank is full (FB only)
waterIsUnderHalf()	B	Returns true if this FB's water tank is less than 50% (FB only)
allFiresInRange()	L_{build}	Returns a list of all extinguishable fires (FB only)
pathExists(T)	B	Returns true if a path exists to T from current position

Table 3.11: Language Step 1: Appropriate action selection

solutions. Several primitives were added to the language (see Table 3.11) that may enhance the individuals' decision making ability. Ideally, the addition of type-specific personal (as opposed to environmental) queries will enable agents to cope with the full complexity of their tasks. Specifically this step aims to develop agents with the ability to perform the more difficult aspects of their duties, such as refilling water tanks for FB agents, and loading and transporting civilians for AT agents. By the end of this step well-adapted individuals should exhibit complete rescue behaviours, choosing the appropriate action for a given situation.

Primitive	Returns	Description
allBuildings()	L_{build}	Returns a list of all Buildings
allCivs()	L_{civ}	Returns a list of all Civilians
allRoads()	L_{road}	Returns a list of all Roads
buildIsExplored(L_{build})	L_{build}	Removes all unexplored buildings
buildIsUnexplored(L_{build})	L_{build}	Removes all explored buildings
getAdjacentRoads(T_{road})	L_{road}	Returns all roads directly connected to T_{road}
getAdjacentBuilds(T_{road})	L_{build}	Returns all buildings accessible from T_{road}
farthest(L)	T	Returns T from L with the largest distance from the agent
randomElement(L)	T	Returns a random element from list L

Table 3.12: Language Step 2: Exploration

Task Assignment

The task assignment behaviour class is broken down into three steps. The first, (termed step 2 to follow from the appropriateness steps 0 and 1), deals with developing the exploration ability. To accomplish this, primitives were added to this subset that would increase an agents ability to represent a wider range of targets. The first two list-refining primitives are used which explicitly denote buildings as explored or unexplored, and two new target selection primitives are added to reduce the system's tendency to only consider the nearest target.

Following this step, the remainder of the task assignment behaviour class is addressed in two steps, separated so as not to be decidedly too large of a step. First, step 3 provides an increased variety of target selection primitives as seen in Table 3.13. Next, step 4 provides a large number of list refining primitives, listed in Table 3.14. Both of these steps have the effect of saturating the language with many potential options. Although this may be a substantial increase in the size of the search space, it is hoped that GP will determine which of the added primitives are beneficial to the overall solution. From this point of view the development of task selection behaviour becomes an exercise in reduction of dimensionality.

Cooperation

Cooperation in this thesis is considered to be the gathering of knowledge by one agent about another agent and subsequent use of that knowledge to guide their own actions. Thus the primitives which make up the cooperation step have the common attribute of giving access to information about other agents. Since it is not clear from the outset whether cooperation is a difficult behaviour to evolve or whether its effects will be generally beneficial, experiments involving cooperation will be considered as a side-step at each of the previously mentioned evolution steps. This way cooperative behaviours can be evaluated in terms of their prevalence in the population and their effect on the resulting score at each stage of evolution. Table 3.15 lists the primitives used in each cooperation step. The primitives in these steps are selected to match with the general concepts of their immediately

Primitive	Returns	Description
selBuildFieryest(L_{build})	T_{build}	Returns the T_{build} with the highest fieryness from L_{build}
selBuildLeastFiery(L_{build})	T_{build}	Returns the T_{build} with the lowest fieryness from L_{build}
selBuildMostArea(L_{build})	T_{build}	Returns the T_{build} with the most area from L_{build}
selBuildLeastArea(L_{build})	T_{build}	Returns the T_{build} with the least area from L_{build}
selBuildMostBroken(L_{build})	T_{build}	Returns the T_{build} from L_{build} that is most broken
selBuildLeastBroken(L_{build})	T_{build}	Returns the T_{build} from L_{build} that is least broken
selCivHealthiest(L_{civ})	T_{civ}	Returns the healthiest T_{civ} from L_{civ}
selCivLeastHealth(L_{civ})	T_{civ}	Returns the T_{civ} from L_{civ} that is nearest to death
selCivMostBuried(L_{civ})	T_{civ}	Returns the most buried T_{civ} from L_{civ}
selCivLeastBuried(L_{civ})	T_{civ}	Returns the least buried T_{civ} from L_{civ}
selCivMostDamaged(L_{civ})	T_{civ}	Returns the most damaged T_{civ} from L_{civ}
selCivLeastDamaged(L_{civ})	T_{civ}	Returns the least damaged T_{civ} from L_{civ}
selRoadMostBlocked(L_{road})	T_{road}	Returns the T_{road} from L_{road} that has the most blockage
selRoadLeastBlocked(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the least blockage
selRoadWidest(L_{road})	T_{road}	Returns the widest T_{road} from L_{road}
selRoadNarrowest(L_{road})	T_{road}	Returns the narrowest T_{road} from L_{road}
selRoadMostOpenLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the most open lanes
selRoadMostBlockedLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the most blocked lanes
selRoadFewestOpenLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the least open lanes
selRoadFewestBlockedLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the least blocked lanes
selRoadMostLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the most total lanes
selRoadFewestLines(L_{road})	T_{road}	Returns the T_{road} from L_{road} with the fewest total lanes
selRoadHghestCost(L_{road})	T_{road}	Returns the T_{road} from L_{road} with highest repair cost
selRoadLowestCost(L_{road})	T_{road}	Returns the T_{road} from L_{road} with smallest repair cost

Table 3.13: Language Step 3: Task Assignment (a)

Primitive	Returns	Description
isNullTarget(T)	B	Returns true if T is null
isEmpty(L)	B	Returns a true if list L is empty
buildBurning(L_{build})	L_{build}	Removes all non-burning buildings from the given list
buildExtinguished(L_{build})	L_{build}	Removes all burning buildings from the given list
buildFieryness^f(L_{build})	L_{build}	Refines the list to those buildings with fieryness values of f
buildHasCiv(L_{build})	L_{build}	Removes all buildings that don't contain civilians
buildInRange(L_{build})	L_{build}	Removes all buildings outside of extinguish range
civDead(L_{civ})	L_{civ}	Removes all living civilians from the given list
civNearDeath(L_{civ})	L_{civ}	Removes all civilians with more than 24% health
civInjured(L_{civ})	L_{civ}	Maintains only civilians with health in the range 25%-49%
civWounded(L_{civ})	L_{civ}	Maintains only civilians with health in the range 50%-74%
civHurt(L_{civ})	L_{civ}	Maintains only civilians with health in the range 75%-99%
civUnhurt(L_{civ})	L_{civ}	Removes all civilians with less than 100% health
civAlive(L_{civ})	L_{civ}	Removes all dead civilians from the given list
civBuried(L_{civ})	L_{civ}	Removes all unburied civilians from the given list
civDamaged(L_{civ})	L_{civ}	Removes all undamaged civilians from the given list

Table 3.14: Language Step 4: Task Assignment (b)

Primitive	Returns	Description
Cooperation Step 0		
nearest(L_{agent})	T_{agent}	Returns the nearest T_{agent} in L_{agent}
allATs()	L_{agent}	Returns a list of all Ambulance Teams
allFBs()	L_{agent}	Returns a list of all Fire Brigades
allPFs()	L_{agent}	Returns a list of all Police Forces
lastAgent(T_{agent^1})	T_{agent}	Returns the last T_{agent} that T_{agent^1} considered
lastBuild(T_{agent})	T_{build}	Returns the last T_{build} that T_{agent} considered
lastCiv(T_{agent})	T_{civ}	Returns the last T_{civ} that T_{agent} considered
lastRoad(T_{agent})	T_{road}	Returns the last T_{road} that T_{agent} considered
Cooperation Step 1		
hasAgent(Tagent)	B	Returns true if Tagent's current agent pointer is not null
hasBuild(Tagent)	B	Returns true if Tagent's current build pointer is not null
hasCiv(Tagent)	B	Returns true if Tagent's current civilian pointer is not null
hasRoad(Tagent)	B	Returns true if Tagent's current road pointer is not null
Cooperation Step 2		
allAgents()	L_{agent}	Returns a list of all Agents
farthest(L_{agent})	T_{agent}	Returns the farthest T_{agent} in L_{agent}
random(L_{agent})	T_{agent}	Returns a random agent from L_{agent}
Cooperation Step 3		
–	–	No new primitives this step.
Cooperation Step 4		
agentIsActing(L_{agent})	L_{agent}	Removes all agents whose last action was not a successful rescue action

Table 3.15: Cooperation Language Steps

preceding steps, so earlier cooperation steps have simplified sets of primitives to reduce the step size. In general, the cooperation primitives either provide access to agents or their status. Each time an agent considers a target it is saved as its last considered target of a given type. This information is then accessible by other agents to enable 'intentional' cooperative actions. Thus considerations regarding cooperation can be evaluated in terms of the use and subsequent reward (or lack thereof) of these specific primitives.

3.5 Training and Validation

The training data used throughout experimentation is the default initialization of the RCRSS. The map and map setup used is the standard Kobe map presented earlier in Figure 2.2. Each individual in GP population is evaluated once based on its performance on this map. While this is not an ideal situation for the evolution of robust behaviours it is necessary in order to limit the amount of execution time required. It should be noted that the GP tree itself is used over 300 times per evaluation, in a constantly changing environment.

The validation of the results of this thesis must vary depending on the step being evaluated. In the case of validating GP's ability to evolve appropriate action behaviour, the validation must take into account the behaviours themselves. These first two steps have specific expected solutions, that are the behaviours required by the RCRSS in order for the rescue agents to attain the base-line of functionality. This functionality is assumed by any manual solution to this problem. Thus the goal of these first steps are not the development of finely-tuned strategies, but the analysis of GP's ability to determine the necessary courses of action given the simulation environment. That being said, the validation of these steps will take the form of a manual analysis of the most commonly converged-to behaviours, and the frequency of expected solutions in the final populations.

The remaining steps build upon this basic functionality with attempts to improve the abilities of the rescue agents evolved. In the case of these later steps the GP system is tasked with learning strategic behaviours, in contrast to the appropriateness steps where the GP is learning more operational behaviours. In these later steps the behaviours developed can be compared in terms of their relative effectiveness between each other and ultimately compared to the strategies manually developed by past competitors in the RCR competition.

Chapter 4

Experiments and Results

4.1 Experimental Setup

Before providing the experimental setup details, a distinction must be made explicit between the terms *Score* and *Fitness*. Here score refers to the RCRSS scoring formula (formula 2.1) while fitness refers to the mechanism for evaluating an individual in the GP population. In many cases the fitness is equal to the score, however it is not necessarily so.

The experiments in this thesis were run on a variety of machines from the Brock University Computer Science department. In total, 46 computers were used, their specifications are as follows:

-20x Intel(R) Pentium D CPU 3.40GHz, 1GB RAM

-20x Intel(R) Core(TM)2 Duo CPU 3.00GHz, 4GB RAM

-3x AMD Phenom(TM) II X4 955 3.20GHz, 4GB RAM

-3x Intel(R) Core(TM) i7 CPU 2.67GHz, 6GB RAM

The parameters used in the GP system developed here were initially given values commonly found in GP implementations. After some preliminary experiments the values seen in Table 4.1 appeared to be the most effective. Many values remained standard. However, an unexpected result initially suggested that 100% mutation should be used in place of any crossover. This parameter is discussed further in Section 4.2. The values in Table 4.1 are used throughout experimentation unless otherwise specified. When crossover is examined in Section 4.2, subtree crossover is used with crossover rates from 0% to 100% and depth 10. All experiments are averaged over 10 runs. While more runs would be ideal, the computational cost of each run limits the execution of more runs under the time constraints.

A number of initial experiments were performed utilizing the developed GP system in the standard format (i.e. with no problem decomposition). The results from these experiments (Section 4.2.1) suggested the need for an alternative GP technique. Existing layered and hierarchical GP

Parameter	Value
Initialization	Ramped half-and-half
Initialization Depth	1 - 10
Crossover Rate	0%
Mutation Rate	100%
Mutation	Subtree Mutation
Mutation Depth	10
Selection	Tournament
Tournament Size	3 individuals
Elitism	1 individual
Generations	50
Population Size	50

Table 4.1: Default parameter set used throughout experimentation unless otherwise specified.

techniques were considered but were decided unsuitable for the heavy constraints of this task (see Section 2.3). An inspired peer of these techniques was designed and implemented for the purposes of this thesis and is described in the section that follows.

4.2 Results

4.2.1 Some Initial Experiments

Once the construction and combination of GP and RCR systems were complete a simple experiment was run. The fitness for this first experiment was the score output from the simulation system (equation 2.1, equation 4.1). This experiment (deemed Exp0) is the ideal of GP for this problem. A low-level language is presented (described in Section 3.2) and the fitness is equated to the overall objective of the problem. An effective evolution in this experiment would not only result in the achievement of the intended objective, but would also prove GP capable of effective performance in problems with high-level abstraction between the constructed individuals and the complex environment of their interaction.

$$fitness_0 = Score = (P + \frac{H}{H_{init}}) * \sqrt{\frac{B}{B_{max}}} \tag{4.1}$$

Unfortunately, GP was not capable of evolving in this experiment. The convergence graph for Exp0 (see Figure 4.1) fully illustrates this lack of performance. While regrettably unpromising, this experiment does identify an important issue that remains central to the remainder of experiments throughout this thesis. The reason GP fails to optimize on this fitness function is due in no small part to the disparity between the GP language and the calculation of the score. Not one single primitive has a direct impact on the simulation score. Primitives such as Extinguish and Remove operate on one or more intermediate simulators which then (if successful) operate on the score. Also, if considered alone, not one single primitive is capable of even indirectly affecting the score. Before extinguishing, a FB agent must move to a burning building (or wait for the blaze to reach their position). To rescue a civilian, an AT agent must first remove them from debris, load them, transport them to a refuge and finally unload them. This separation between genotype and phenotype creates an unpredictable fitness landscape that clearly becomes difficult to optimize outside of a random search through the possibilities. One final note explaining GP's difficulty with this evolution is the complexity of the environment. As an example, consider the FB agent who waits for the blaze to approach its position. Even if this FB agent were capable of extinguishing all fires within its range, the delay taken in waiting for the fires to reach them would allow the blaze to spread wide in all other directions. Thus the overall score would be nearly unchanged despite their valiant 'efforts'. This means that simpler or partially-correct solutions to the problem may go unrewarded or unnoticed. These factors combine to present the evolutionary mechanism of GP with a difficult issue. With such a low causality between solution and fitness, convergence of solutions (as is the evolutionary mechanism) does not coincide with convergence of fitness. This leaves the GP with no clear direction the search space to follow towards maximizing the fitness.

The question may then arise, 'why not simply decrease the level of abstraction between the language and the score?' That is, why not create higher-level primitives that operate directly on

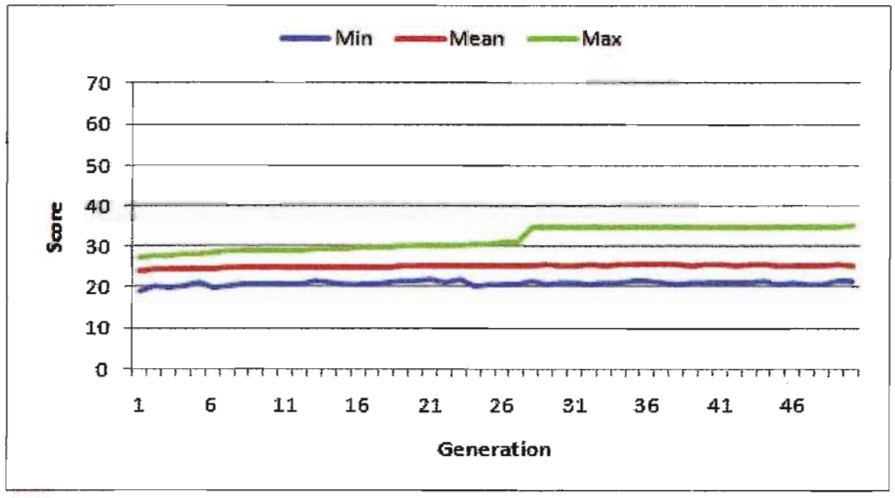

Figure 4.1: Experiment 0 convergence curve, min, mean and max

the environment. For example, a primitive called extinguishNearestFire which first moves to the nearest fire and then extinguishes it, would clearly be simpler for GP to use than the equivalent combination of primitives in the GP language used here. The reason this is not done in this thesis is threefold. Firstly, the structure of a given agent in the simulation does not allow for multiple actions to be executed in a single simulation step. While this could be easily circumvented with some state variables used over several consecutive simulation steps, the GP language used here was constructed as a replacement for the existent agent decision module. As such, primitives have been restricted to use within a single simulation step. The second reason why such high-level primitives are not implemented here is because such primitives would limit GP's 'creativity'. GP is often employed to find solutions that a human creator may have overlooked. The creation of primitives that dictate strategy are in opposition to this potential purpose of GP. A counter-question to the above could be proposed, 'why not create a primitive such as extinguishAllFires?' A line of abstraction had to be drawn at some point and it was decided to reside somewhere near to the sensory data and actions that agents natively have access to. Thirdly, this thesis aims to explore the capabilities of GP. Thus following the ideals of a 'natural' language that does not dictate strategy despite the challenges such a description may pose is within the scope of the ideals of this thesis.

Following the lessons learned from Exp 0, several attempts were made at tailoring the fitness function for simple improvements to the evolution. The concept of a fault was developed to describe situations during the simulation where the language performed an invalid action. Such situations occurred when an agent attempted an action on a target of an incorrect type, for example a FB agent extinguishing a civilian (all such faults have since been removed from the language). Another common source of faults was the use of operations on an empty list. For the experiments here a

penalty was subtracted from the fitness score for the occurrence of each fault. The penalty was designed to reduce the score by 75 points on the worst case scenario of 25 agents faulting for each of the 300 simulator steps. Conversely, the concept of 'safety' primitives was monitored in a similar fashion to represent the use of primitives that check the state of the environment before acting (e.g. atFire, anyTrapped, isEmpty, etc). The fitness functions attempted are presented as formulas (4.2) through (4.5).

$$fitness_1 = score - faults \tag{4.2}$$

$$fitness_2 = -faults \tag{4.3}$$

$$fitness_3 = score - faults + safety \tag{4.4}$$

$$fitness_4 = score + safety \tag{4.5}$$

Common throughout experiments with each of these fitness functions was GP's ability to optimize on the faults or safety objectives. However, these optimizations often came at the detriment of the score objective. Rather than an intelligent or judicious organization of primitives into an effective decision logic, the GP individuals were biased towards those primitives that fault less or simply don't fault at all. Primitives such as Rest and Move will not fault. Primitives such as Extinguish or any of the list-refining primitives are likely to cause faults if not properly used. Thus the resulting GP individuals contained very short logic trees, often consisting of a small Move operation or a single Rest primitive. Ultimately this led to a population that was unwilling to risk using potentially faulty primitives, and thus there was no reward in terms of simulation score. Similarly, to optimize the safety objective, trees became bloated with if statements utilizing safety primitives but with little or no useful actions.

These tailored fitness functions failed to reduce the gap between genotype and fitness score. However, while *very* inconsistent, some good individuals did appear in occasional populations proving that it was possible within this scheme, but suggesting that the fitness landscape is too rugged for this approach. In order to simplify the search, it was decided to narrow the search space by limiting the number of primitives in the language to a small subset. This led to the step-wise evolutionary strategy described in the next section, following a brief discussion about parameters.

Commonly known good values were initially used for the parameters, however it was found that having a very high mutation and little to no crossover seemed to give the GP a better chance of generating a good individual. This saw the best individual in a population occasionally jump to a reasonable fitness despite a general lack of ability to evolve throughout the population as a whole.

The estimated reason for this is that mutation is freer than crossover to generate novel subtrees that may be useful in removing the population from local optima. In this regard, one could consider crossover to be a form of mutation that is constrained to only generating pre-existing subtrees. While the value of 100% mutation rate is based on a version of the system that used an older variant of the GP language and non-step-wise learning, there may still be a practical benefit to it. The seeding of one step's population into the next creates a population composed entirely of primitives from the previous step, and none from the new step. The typically high value for crossover would reinforce this undesirable state, but high mutation values should work to counteract it.

Further testing of this value was performed with crossover rates of 0%, 20%, 40%, 60%, 80% and 100% in the step 0 single population setup (used below). While the quality of solutions in terms of fitness_0 appear to converge faster and to lower values with increased use of crossover, it is not drastically so. A two tailed t-test was performed between the 0% mutation set and each other experimental set. P values ranged from 0.37 to 0.83, all well within the range of a 95% confidence interval, suggesting that the use (or lack of use) of crossover has no statistically significant impact on the results. While this is somewhat unusual, it is not without precedent. Two independent studies comparing crossover to mutation found that the results of mutation were generally more favourable in situations involving smaller populations [41, 42]. This thesis uses small populations out of necessity as larger populations would be too time consuming to evolve. Therefore, for the reasons of its utility in maintaining high diversity throughout such compressed evolutionary executions, as well as for its practicality in spreading new primitives to a seeded population, a value of 100% mutation rate and 0% crossover rate was adopted as the standard for the remainder of experimentation. This parameter is further examined utilizing the best solutions from the work in the following step, in order to support this theory of small population sizes.

4.2.2 Step-wise Evolution Results

Appropriateness, Step 0: Single Population Fitness_0

The first set of results obtained by this thesis proved the stepping procedure to be immediately better than the non-stepping procedure. While the non-stepping procedure was occasionally capable of finding a good result after many generations of searching amidst large populations, the step-wise evolution consistently found better-than-random results in short time spans with smaller populations. A comparison of the average convergence of both experiment 0 and step 0 is presented in Figure 4.2. As expected a smaller search space leads to improved performance.

The fitness distribution of the population providing the average min, max, and IQR of the population's scores as it evolves is illustrated in Figure 4.3. Perhaps the most obvious characteristic of the distribution is that the minimum values of the population scores remain relatively constant throughout the entire evolution. This is likely a result of the difficult solution space (dynamic, uncertain, etc), where small changes to an individuals solution may have a large impact on its

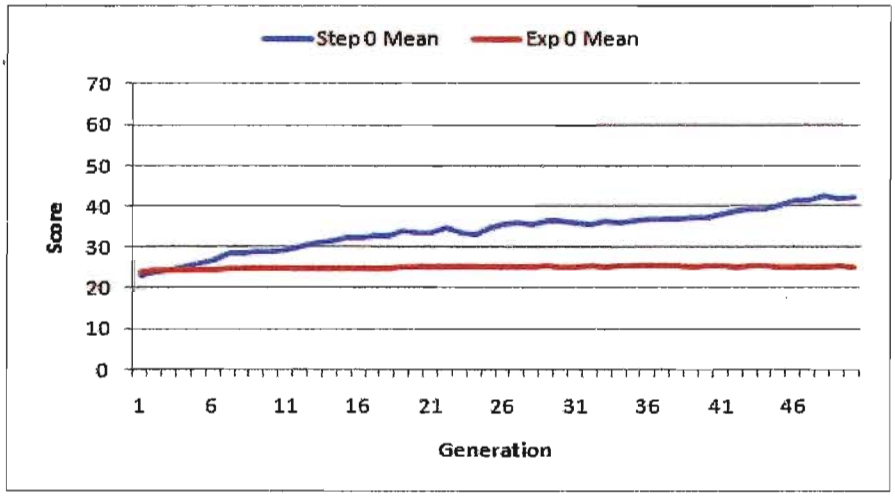

Figure 4.2: Comparison of the means of Step 0 and Exp 0.

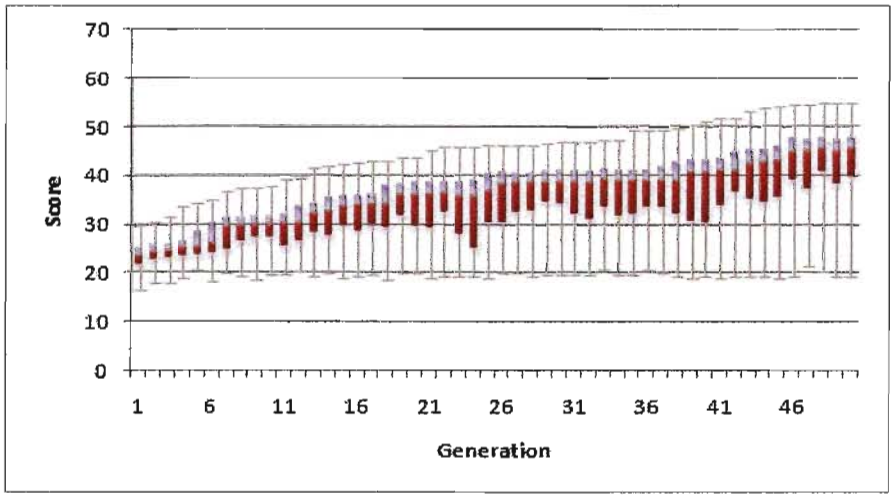

Figure 4.3: Average population distribution of step 0. From bottom to top in each column, lines indicate the min, Q1, median, Q3, and max of the population scores at each generation.

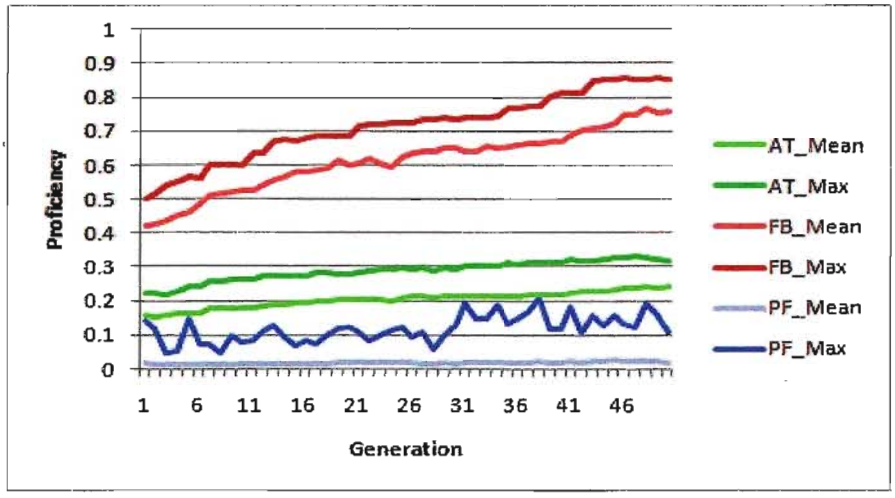

Figure 4.4: Max and mean proficiency of each type of rescue team over time.

fitness score. As evolution progresses these weaker individuals form a minority of decreasing size, as evident by the increasing gap between min and Q1, and the relatively smaller gaps between Q1 and up. This asserts that some amount of convergence is taking place.

Figure 4.4 illustrates the maximum and mean proficiency over time of each type of rescue agent. The proficiency values here represent the ratio of remaining civilian health, the ratio of unburned buildings, and the ratio of cleared blockades, for AT, FB, and PF agents respectively. Note that these proficiencies do not represent the fitness function for these experiments. From this figure it is clear that the improvements in score are due to the evolution of effective FB logic and little else. In fact, the slight improvement seen in the AT proficiency can be explained as a lack of fires harming civilians leading to more civilian health remaining at the end of simulation.

A manual analysis of the decision logic trees of the final generations from these experiments was performed to determine the developed behaviours. It was found that FB agents evolved to the expected solution for this step or logically equivalent variants to this solution in the majority of runs. Figure 4.5a) presents the fire brigade agent tree evolved in the best-scoring GP individual of this step. The logic of this solution is simple: move to the nearest fire and extinguish it. This simple logic was found independently in all of the experimental runs (almost ubiquitously in some). Most populations even made correct use of the getEntrances primitive to avoid having FB agents harm themselves by walking into the buildings they were extinguishing.

In contrast to the very effective FB decision tree in 4.5a), the decision tree in 4.5b) performs quite poorly. Both trees have an identical structure and all but a single terminal node is changed from a) to b). Despite having only a single differing node the fitness is a clear decrease, as is expected given the change in logic. This highlights the volatility of solutions in this search space. Such volatility

could be the cause of the lack of performance found in both the AT and PF behaviours. Neither of these types of agents seem to have evolved consistently well-defined behaviours appropriate to the task at hand. In the final generation, a handful of individual PF trees managed to clear more than 10% of the roadblocks with convoluted behaviours. One noteworthy PF behaviour cleared all roadblocks between it starting point and the nearest refuge accounting for 20% of the total amount, which could clearly be beneficial to a properly operating team of individuals (remember FB agents will need to refill their water tanks at the refuge, and AT agents must bring wounded civilians to a refuge). One single individual out of the final populations of 10 executions evolved a PF decision tree that cleared 55% of road blocks utilizing a refined greedy approach. During evolution a number of good PF trees appeared, which used the greedy approach, often skewed amidst many nonsensical combinations of primitives. These solutions however, having no direct impact on the score, were not preferred by selection. This is evident by the PF's low mean, seen in Figure 4.4.

The most effective AT in terms of score as well as percentage of civilian health preserved, maintained 36% of the total initial health of all civilians. This however was achieved utilizing the logic seen in the AT tree of Figure 4.6. It is clear from looking at this AT's decision tree, that the AT had no hand in preserving the health of any civilian. Not surprisingly, the FB tree from this individual managed to preserve 89% of the surface area of buildings, this combined with the randomized damage calculations for trapped civilians brought this solution to the top. This hints that the difficulty in evolving AT logic may be the unpredictability resulting from the effects of external forces. So, unlike the FB and PF logics, small pieces of semi-correct logic will not benefit the fitness. Instead AT trees that are in the same GP individual as an effective FB tree have the selective advantage regardless of their own merit. Again, small elements of proper behaviour logic were found in some AT trees, but convoluted similar to the PFs.

Finally, the fact that the tree in 4.6 has the highest score of all 500 individuals (50 individuals, 10 runs, final generation only), illustrates the bias of the scoring function. Here we have a somewhat well adapted FB, along side a useless AT, and a confused PF (note that this PF will only clear when NOT at a blockade). The scoring function is heavily weighted to the survival of individuals, and rightly so given the context of this simulator. However this clouds the evaluation of the behaviours of the individual agents. As seen in this case, a lucky break in the random damage coupled with a somewhat effective (although not optimal) FB has allowed the score to favour two ineffective trees. This issue is addressed later in this section with the introduction of multiple populations.

Appropriateness, Step 0: Single Population Fitness$_5$

The primary reasons for the convergence to sub-optimal solutions appear to be a result of an inability to evolve operational AT and PF behaviours. It is theorized that this is due to the credit assignment problem that is inherent in having three trees control 25 agents in a single individual that is given a single fitness score. This single fitness score is heavily weighted towards the number of surviving

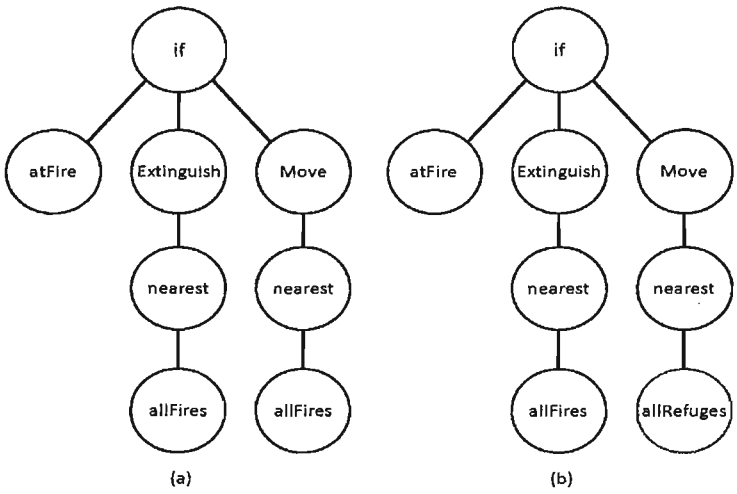

Figure 4.5: Example Fire Brigade trees from final population of step 0. (a) Tree from the best scoring individual. (b) Similar tree that performs poorly.

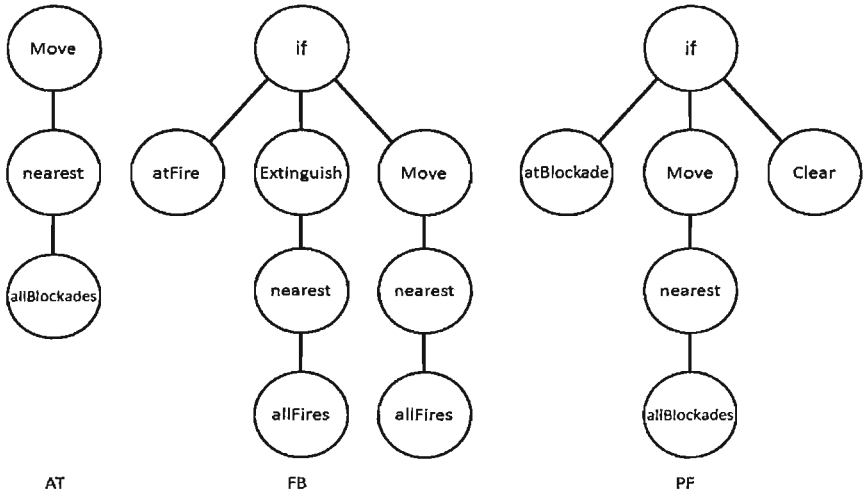

Figure 4.6: The best individual from the first step 0 experiment. Note that the FB tree is the only contributing member of this individual.

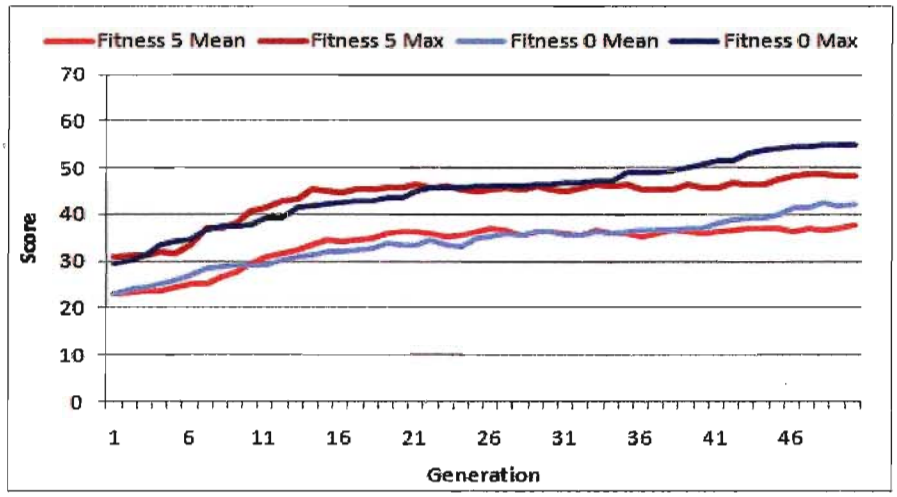

Figure 4.7: Mean and max scores of fitness_0 (coloured blue) versus fitness_5 (coloured red).

individuals (whole numbers), less weighted towards the amount of unburnt buildings (ratio: [0,1)), and ignorant of the work done by the police force. To address this a new fitness formula was first designed that put all three agent types on similar footing. This formula is as follows:

$$fitness_5 = \frac{HP_{final}}{HP_{init}} + \frac{UnburnedArea}{Area} + (1 - \frac{B_{final}}{B_{init}}) \tag{4.6}$$

where HP is the amount of civilian health ($_{final}$ is at the end of simulation, $_{init}$ is at the beginning), UnburnedArea is the total area of unburned buildings at the end of simulation, Area is the total building area, and B is the amount of blockage (blockades).

The simulator scores achieved in this experiment appear slightly worse in terms of max and mean than those of the previous experiment. Examination of Figure 4.7 suggests that fitness_5 converges to a slightly lower optima than fitness_0. This is apparently due to the similarly decreased FB proficiency (see Figure 4.8) evolved in these experiments. On average the FB proficiencies in the final generation of fitness_5 are slightly less than those of fitness_0. However the single best individual from the fitness_5 experiment outperforms the single best individual of the fitness_0 experiment by approximately 2%. This can be attributed to the statistically significant improved PF proficiencies. With the PFs capable of opening roads, FBs will have increased access to burning buildings. This means that identical FB behaviours in both experiments will have better results in the fitness_5 experiments, however, having to share the evolutionary spotlight (in terms of fitness) with PF trees has the effect of slightly holding back the evolution of the FB trees. Thus while the single best FB solution preserved 94% of all buildings, the average best solution per run preserved just above 80%. The best behaviours of these FB agents is the 'move to the nearest target and act' behaviour that was found in both experiments. Similar behaviours were found in the PF agents (as with before as

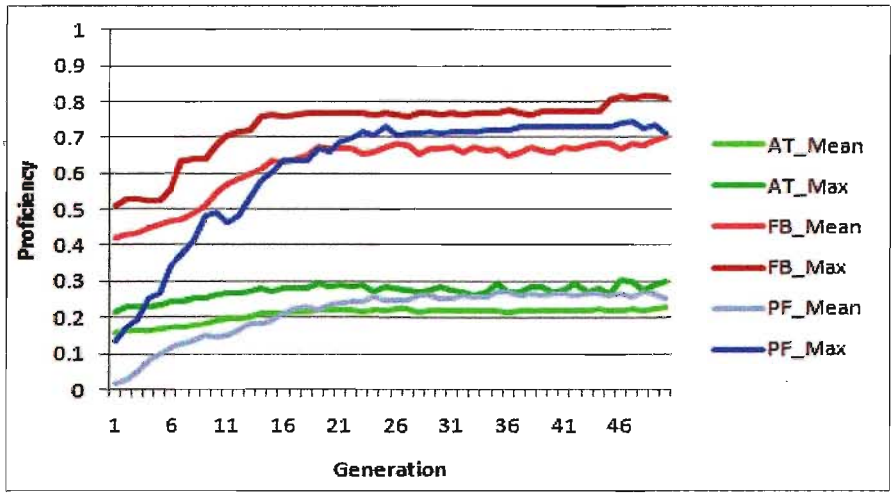

Figure 4.8: Max and mean proficiency of each type of rescue team utilizing fitness_5. FB values are indicated with red lines, PF are blue lines, and AT are green lines.

well), however in this experiment there is a reward to the PF behaviours so they are not overlooked and thus maintained in the population. The PF behaviours demonstrating the highest proficiency at clearing blockades are found in large convoluted trees containing many Clear and Move actions. This leads to a more chaotic behaviour that appears to be a simple form of exploration. This will be further explored in the Task Assignment section of the results with the introduction of exploration primitives.

The AT proficiencies remain unevolved. The trees developed are typically single action trees containing no if-statements. It is suspected that this is due to the fact that it is still a single fitness to evaluate three distinct plans of action. Since the FB and PF plans are evidently more readily evolvable, improvements to the AT logic may be going unnoticed by the evaluation.

Appropriateness, Step 0: Multiple Populations Fitness_5

A second form of population was implemented to address the lack of evolution occurring in the AT trees. This population consists of three distinct cooperating sub-populations, one for each type of agent. This way, each agent type could be evaluated and evolved with increased independence, utilizing a score that is directly representative of the work done by a given tree. The fitnesses used for each subpopulation are the respective parts of formula (4.6). Apart from these two changes, the experimental parameters remain identical to those used in the discussion so far.

A comparison between the mean and maximum scores of experiments using a single population versus multiple sub-populations is presented in Figure 4.9. From this it is clear that the use of multiple populations has a great impact on the speed of evolution. While both experiments reach a

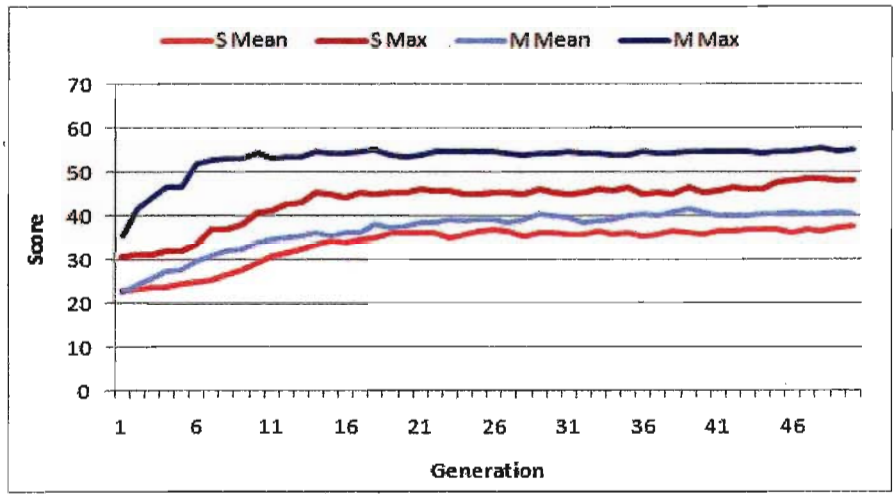

Figure 4.9: Single versus multiple sub-populations comparison. Mean and max for both single (labelled S, coloured red), and multiple (labelled M, coloured blue) subpopulations.

similar state in terms of mean population score, the multiple population experiments find top scores near 55 within 10 generations on average, while the single population peaks around scores of 50. Similar to the single population, step 1 does not improve the solutions in the multiple population tests.

Figure 4.10 presents the max and mean proficiencies of each type of agent. When compared to Figure 4.8 the multiple populations appear to offer an advantage to both speed of evolution and maximum proficiency, while the mean proficiencies for each agent type appear similar between both experiments. FB proficiency at preserving buildings is found at a maximum of 92%-94% in the final population of every run of this experiment as compared to only a handful of times in the single population experiment. PF agents again demonstrate some evolution, clearing 20% of roadblocks on average and achieving a maximum proficiency of 76%; roughly equivalent to their single population counterpart. AT however remain unevolved, any slight improvements in proficiency at preserving civilian health is attributable to increased FB proficiency at stopping fires from reaching civilians. The change in convergence speed and maximum proficiencies may be attributable to the presence of separate elitism individuals in the multiple population experiments as opposed to a shared elitism individual in the single population, which may alone be the critical difference when deciding between these two forms of population. The trees developed here are similar to those of the single population.

Appropriateness, Step 0: Fitness_7

The introduction of multiple populations did not stimulate the evolution of AT behaviours as was hoped. The best AT proficiencies continue to coincide with effective FBs. The issue here seems to

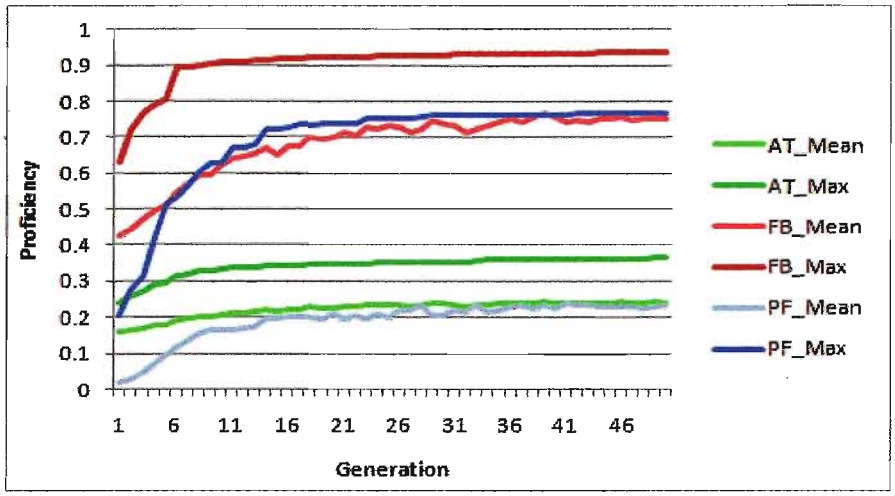

Figure 4.10: Max and mean proficiency (fitness) of each type of rescue team over time in the multiple population experiment utilizing fitness_5. FB values are indicated with red lines, PF are blue lines, and AT are green lines.

be the random amounts of damage done to civilians causing an unclear evaluation of AT behaviours. Also, the task of the AT must be performed completely in order for it to have any effect. Simply clearing a civilian does not preserve its health, the civilian must be brought to a refuge in order to stop taking damage. Formula (4.7) was created to address these issues.[1]

$$fitness_7 = (\frac{Bury_{final}}{Bury_{init}} \times \frac{HP_{final}}{HP_{init}}) + \frac{UnburnedArea}{Area} + (1 - \frac{B_{final}}{B_{init}}) \tag{4.7}$$

where Bury is the amount of total 'buriedness' of all civilians. This describes how badly buried a civilian is within a building, and consequently how much work and time are required for the AT to successfully remove them from the building. As with fitness_5, this formula is separated into its component parts (at each '+') for experiments using multiple populations. It was decided to multiply the unburying proficiency with the health preservation proficiency so that AT trees that did not perform the former task were given a score of zero. This provided the evolution of AT populations with a clear starting point in terms of fitness that reflected meaningful constructions as opposed to random influence.

The experiments involving this formula did not improve the AT proficiency at preserving civilian HP, as seen in Figure 4.11. The maximum proficiency of ATs at this task actually dropped slightly due to the loss of elitism specific to this task, the mean proficiency however remained the same. It

[1] Fitness_6 was first created which averaged the buriedness with civilian damage, but was found that the random health still overwhelmed the buriedness scores. Fitness_7 corrected this by having a score of 0 assigned to any AT that did not start rescuing civilians. Although fitness_6 is not analyzed in this thesis, the numbering was preserved to coincide with the database of results collected.

is important to understand here that it is not necessarily the evolution of scores, but the evolution of behaviours that is sought in this experiment. To this end, fitness_7 does in fact provide an improvement, as many ATs learn simple clearing behaviours. This is seen as an increase in both mean and max values of the 'bury' task in Figure 4.11, as compared to the zero development prior to this experiment. While the maximum proficiency does not rival the competence of the FB and PF agents (which remain expectedly unchanged in this experiment), task-appropriate behaviours are developed. The specific behaviours developed here appear to be of two main schools. The majority of runs converge towards the expected solution for the step 0 language, namely 'move to a target and act on it', often containing significant bloat around this simple core of logic. However a minority (3 out of 10 runs) converged towards an alternative solution stateable as 'if at a blockade then move to the nearest trapped civilian, if not a blockade then remove'. This behaviour is generally less effective than the expected one, although it does result in the removal of a few civilians from their peril. This can be seen as a sort of commentary on the prevalence of blockades in the environment, ultimately suggesting a potential reason for the limited competence at this task. Later steps should explore this further.

For the sake of comparison, an experiment utilizing fitness_7 on a single population was performed. The proficiency scores developed for AT agents in this experiment suggests that the expected behaviour was much less evolvable in this experiment, likely due to the combination of a poorly weighted function for a relatively more complicated task (as compared to the other two types of agents). An attempt was made to weigh the fitness to favour AT development more heavily by multiplying the AT's combined HP and Bury proficiency by 10. Theoretically this shifts the AT proficiency values from [0,1] to [0,10], however in practice these values will likely not exceed 1 or 2 at maximum and will typically reside in the [0,0.1] range as opposed to the [0,0.01] range in which they resided previously. The results of this reweighting (presented in Figure 4.12), raised the evolvability of AT behaviours without effecting the mean FB and PF behaviours. As compared to the multiple population scheme using fitness_7, the mean proficiencies of each task remain similar, while the speed of convergence and maximum proficiency values appear to be decreased in the single population (just like with previous single versus multiple population comparisons).

Appropriateness, Step 0: Summary

To summarize the results of this step, Table 4.2 lists each experiment and the most common behaviours developed in final generations of each population. It is important to recall that there is an expected outcome for this step. Ideally, each agent should learn the simple goto-and-act logic akin to that of the ideal FB agent found in Figure 4.5a). Other well-performing behaviours are welcome solutions to this problem.

A common occurrence amongst nearly all populations for all runs in the multiple population experiments is a lack of convergence to a single identical tree due to the presence of bloat. Bloat

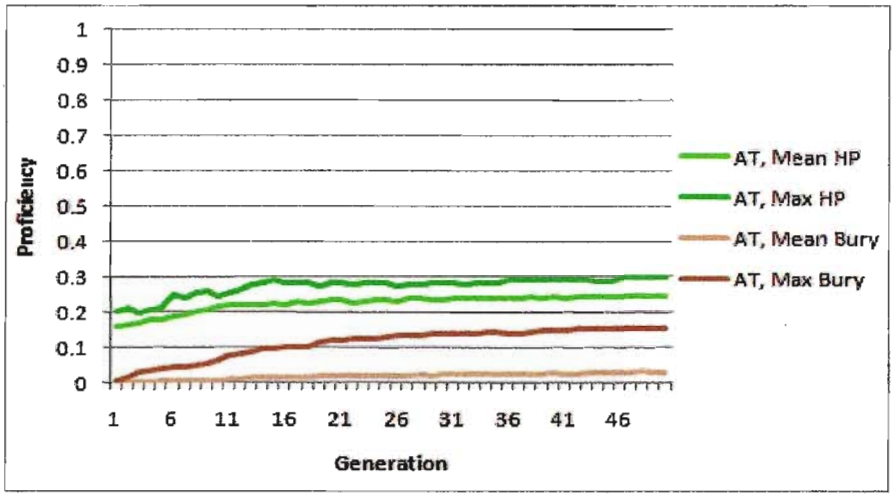

Figure 4.11: Max and mean proficiency of AT agents in the *multiple* population experiment utilizing fitness_7, at the tasks of preserving civilian HP (seen in green), and removing buried civilians (seen in brown).

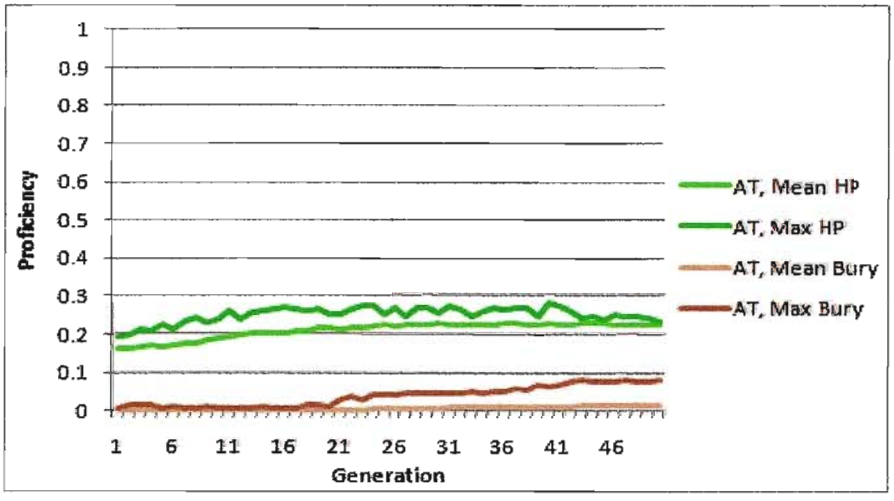

Figure 4.12: Max and mean proficiency of AT agents in the *single* population experiment utilizing fitness_7, at the tasks of preserving civilian HP (seen in green), and removing buried civilians (seen in brown).

Population	Prominent Behaviour
Single Population, $fitness_0$	
AT	No useful development.
FB	Expected behaviour (9 of 10 runs), often with bloated solutions (5 of 9 times).
PF	No useful development.
Single Population, $fitness_5$	
AT	No useful development.
FB	Expected behaviour (8 of 10 runs), often with bloated solutions (4 of 8 times).
PF	Expected behaviour with bloated solutions or alternate destinations (10 of 10 runs).
Multiple Populations, $fitness_5$	
AT	No useful development.
FB	Expected behaviour with bloated solutions (10 of 10 runs).
PF	Expected behaviour with bloated solutions (10 of 10 runs).
Single Population, $fitness_7$	
AT	Expected behaviour (3 of 10 runs), blockade based decision (2 of 10 runs)
FB	Expected behaviour (7 of 10 runs), often with bloated solutions (5 of 7 times)
PF	Expected behaviour with bloated solutions or alternate destinations (10 of 10 runs)
Multiple Populations, $fitness_7$	
AT	Expected behaviour (7 of 10 runs), blockade based decision (3 of 10 runs), bloated (10 of 10 runs).
FB	Expected behaviour with bloated solutions (10 of 10 runs).
PF	Expected behaviour with bloated solutions (10 of 10 runs).

Table 4.2: Summary of behaviours developed in the Step 0 experiments.

in these trees can be defined in two ways. Some occurrences of bloat are logically bloated trees containing branches of primitives that will never be executed in any circumstance (e.g. two nested if statements using the same condition), and other occurrences are contextually bloated trees that contain primitives that are not logically ignorable, but given the initial conditions of the simulated run, will never be executed (e.g. primitives following an if(atRefuge) where no complementary command to move to refuge exists). In both cases the bloat serves to protect other parts of the program from harmful mutations by providing a large number of mutation points that will have no effect on the outcome. The high frequency of bloated variations on the expected solution especially in the multiple population experiments leads to a wide variety of logically similar yet syntactically distinct individual trees in each population. Interestingly, in the PF trees this limited form of convergence has the effect of creating similar clearing logic but to alternative destinations, such as 'if at blockade then clear else move to refuge/fire/trapped'.

A number of unique, unexpected, and interesting solutions were created throughout this step's experiments without being converged to by the general population. Such behaviours include an AT with the appearance of self-preservation logic, that would rescue a civilian in the expected way unless the building were suddenly on fire, at which time the ambulance would move outside the building. Another interesting behaviour was a PF tree that said to clear paths between refuges and

fires, which could be quite useful in the next step where FB agents need to learn to refuel.

Finally, summary tables of the best trees of each agent type found in each experiment, are presented in Tables 4.3 through 4.5. The best trees are determined by their proficiency at their desired task (regardless of the fitness function used). AT proficiencies are listed as pairs in the format (HP,bury) and chosen as the tree which provided the highest overall proficiency (HP times bury). In the case of two identical proficiencies, the best is chosen as the one with the higher proficiency in the 'bury' task, then by HP, then finally at random. The trees here are presented in the form of LISP-style s-expressions. Tree format illustrations of these trees are presented in Appendix A. Also included in this table is the simulator score attained during the trees evaluation. Recall, that this score is based on the performance of the given tree as well as a tree from each of the two remaining agent types.

Returning to the issue of the suitability of mutation versus crossover in this work, recall that experiments using 100% mutation were initially found to be equivalent in effectiveness to experiments using crossover. It was hypothesized that this was due to the small population sizes. Here, results from four experiments utilizing the improved setup developed over the past section are compared. The previously established results using multiple populations and $fitness_7$ with a population size of fifty (50) individuals and a 0% crossover rate, is compared to similar experiments, first using 80% crossover (20% mutation) and a 50-individual population, and second using 80% crossover and a population of 200 individuals. As a control a fourth experiment is run using 100% mutation (0% crossover) with a population of 200 individuals. Table 4.6 presents the mean and max proficiencies on all 4 agent tasks (two AT agent tasks, one FB, and one PF) for each experiment. As expected, both large population experiments outperform their smaller population counterparts on most proficiencies, however the evaluation of the larger populations requires four times the amount of computation time (approximately 400 hours per run divided by the number of parallel evaluations, based on an estimate of 2.4 minutes per evaluation).

In the experiments utilizing 50-individual populations, the comparison of mutation versus crossover appears inconclusive, similar to the results mentioned in the previous discussion. Both the max and mean proficiency for the AT task of unburying civilians are higher in the experiment using 0% crossover than in the 80% crossover experiment, likely due to tendency of mutation to escape local optimal. The 80% crossover experiment has higher mean proficiencies in the remaining tasks, likely caused by the higher rate of convergence caused by the crossover operator. These results appear to indicate, as before, that there is no clear bias of proficiency towards using crossover or mutation with a small population size.

The experiments utilizing 200-individual populations have a clearer distinction. Here the experiments utilizing the crossover operator consistently outperform those using only mutation. This appears to be in affirmation of the hypothesis that as population sizes increase, so to does the importance of crossover. Referring to Figures 4.13 through 4.16, it appears that experiments utilizing

Population	Tree	Proficiency	Score
Single, $fitness_0$	(Move (nearestRoad allBlockades))	(0.365,0.000)	59.04
Single, $fitness_5$	(if atBlockade (Move (nearestRoad (getEntrances (nearestBuild allRefuges)))) Remove)	(0.346,0.000)	49.73
Multiple, $fitness_5$	(Move (nearestRoad (getEntrances (nearestBuild allFires))))	(0.370,0.000)	58.57
Single, $fitness_7$	(if atTrapped Remove (Move (nearestCiv allTrapped)))	(0.237,0.208)	43.95
Multiple, $fitness_7$	(if atBlockade (if atTrapped Remove (Move (nearestRoad (getEntrances (nearestBuild allFires))))) (if atTrapped Remove (Move (nearestCiv allTrapped))))	(0.302,0.224)	50.99

Table 4.3: Summary of the best AT trees (by proficiency) developed in the Step 0 experiments.

Population	Tree	Proficiency	Score
Single, $fitness_0$	(if atFire (Extinguish (nearestBuild allFires)) (Move (nearestBuild allFires)))	0.908	53.80
Single, $fitness_5$	(if atFire (Extinguish (nearestBuild allFires)) (Move (nearestRoad (getEntrances (nearestBuild allFires)))))	0.942	49.93
Multiple, $fitness_5$	(if atFire (if atInjured (Move (nearestRoad (getEntrances (nearestBuild allRefuges)))) (Extinguish (nearestBuild allFires))) (if atInjured (if atTrapped (Move (nearestCiv allTrapped)) (Extinguish (nearestBuild allRefuges))) (if atInjured (Move (nearestRoad (getEntrances (nearestBuild allFires)))) (if atInjured (Move (nearestBuild allRefuges)) (if atInjured (Extinguish (nearestBuild allRefuges)) (Move (nearestBuild allFires)))))))	0.925	50.42
Single, $fitness_7$	(if atRefuge (Move (nearestRoad allBlockades)) (if atFire (Extinguish (nearestBuild allFires)) (Move (nearestBuild allFires))))	0.911	50.05
Multiple, $fitness_7$	(if atFire (if atRefuge (if atRefuge (if atFire (Extinguish (nearestBuild allRefuges)) (Move (nearestCiv allTrapped))) (Extinguish (nearestBuild allRefuges))) (Extinguish (nearestBuild allFires))) (Move (nearestBuild allFires)))	0.942	53.86

Table 4.4: Summary of the best FB trees (by proficiency) developed in the Step 0 experiments.

Population	Tree	Proficiency	Score
Single, $fitness_0$	(if atTrapped Clear (if atBlockade (if atRefuge (MoveB (nearestCiv allTrapped)) Clear) (if atRefuge Clear (if atTrapped Clear (MoveB (nearestRoad allBlockades))))))	0.554	50.47
Single, $fitness_5$	(if atInjured (MoveB (nearestCiv allTrapped)) (if atBlockade (if atBlockade Clear (MoveB (nearestCiv allTrapped))) (if atFire (if atTrapped Clear (MoveB (nearestRoad (getEntrances (nearestBuild allRefuges))))) (MoveB (nearestRoad allBlockades)))))	0.774	50.62
Multiple, $fitness_5$	(if atTrapped Clear (if atBlockade (if atInjured (if atFire Clear (MoveB (nearestCiv allTrapped))) Clear) (if atInjured (if atTrapped (MoveB (nearestBuild allFires)) (MoveB (nearestCiv allTrapped))) (MoveB (nearestRoad allBlockades)))))	0.764	45.92
Single, $fitness_7$	(if atBlockade Clear (MoveB (nearestRoad allBlockades)))	0.775	42.52
Multiple, $fitness_7$	(if atBlockade (if atTrapped (if atRefuge (MoveB (nearestCiv allTrapped)) Clear) (if atRefuge (if atInjured (MoveB (nearestCiv allTrapped)) (MoveB (nearestRoad allBlockades))) Clear)) (if atFire (MoveB (nearestRoad (getEntrances (nearestBuild allRefuges)))) (MoveB (nearestRoad allBlockades))))	0.724	18.76

Table 4.5: Summary of best PF trees (by proficiency) developed in the Step 0 experiments.

crossover have mean fitnesses that converge faster and to slightly greater proficiencies than their mutation-based counterparts in both small and large population sizes. What appears to change in the crossover rate comparison between large and small populations is the consistency with which crossover outperforms mutation as the population size increases.

Population Size	Crossover Rate	AT_{HP} Mean	AT_{HP} Max	AT_{Bury} Mean	AT_{Bury} Max	FB Mean	FB Max	PF Mean	PF Max
50	0%	0.2435	0.2937	0.0291	0.1508	0.7761	0.9345	0.2445	0.7580
50	80%	0.2590	0.2671	0.0279	0.1214	0.8339	0.9306	0.3205	0.7642
200	0%	0.2454	0.2972	0.0430	0.2441	0.7711	0.9372	0.3557	0.8138
200	80%	0.2570	0.2913	0.0541	0.2618	0.8281	0.9490	0.3999	0.8206

Table 4.6: Comparison of the size of a population versus the usefulness of high and low crossover rates

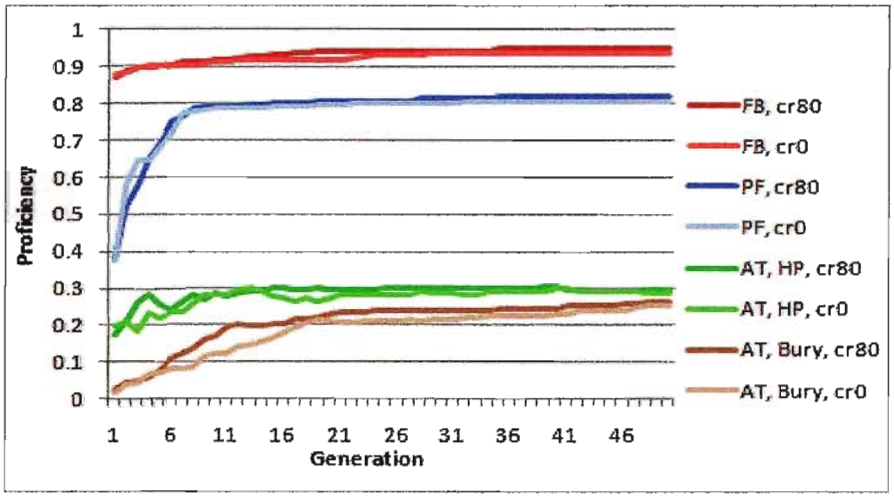

Figure 4.13: Comparison of maximum convergence between 80% crossover and 0% crossover using populations of 200 individuals. Darker colours represent 80% crossover (cr80) and lighter colours represent 0% crossover (cr0).

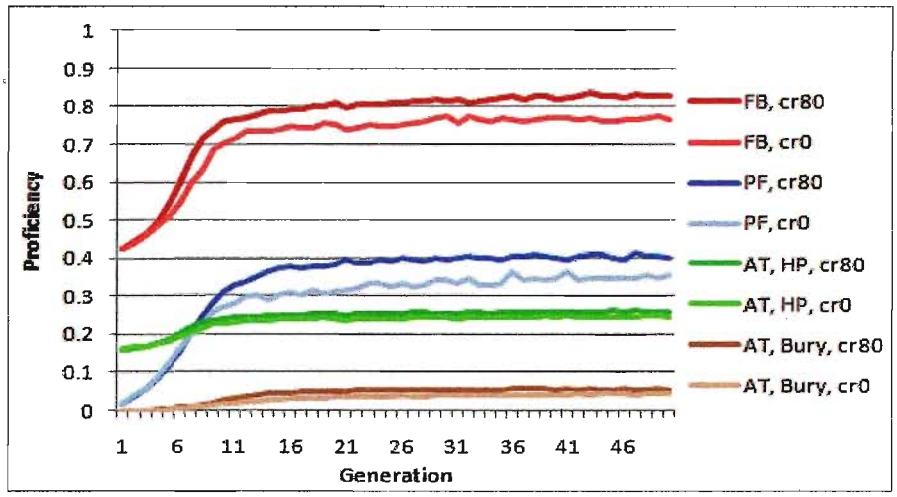

Figure 4.14: Comparison of mean convergence between 80% crossover and 0% crossover using populations of 200 individuals. Darker colours represent 80% crossover (cr80) and lighter colours represent 0% crossover (cr0).

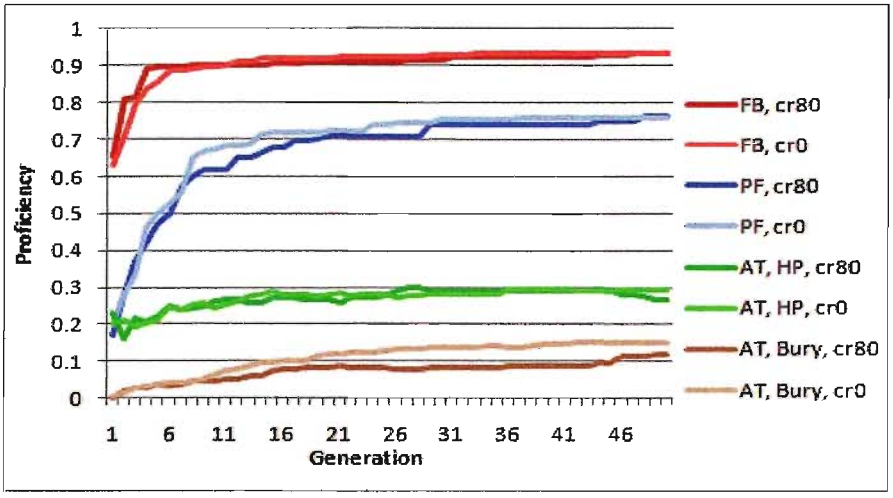

Figure 4.15: Comparison of maximum convergence between 80% crossover and 0% crossover using populations of 50 individuals. Darker colours represent 80% crossover (cr80) and lighter colours represent 0% crossover (cr0).

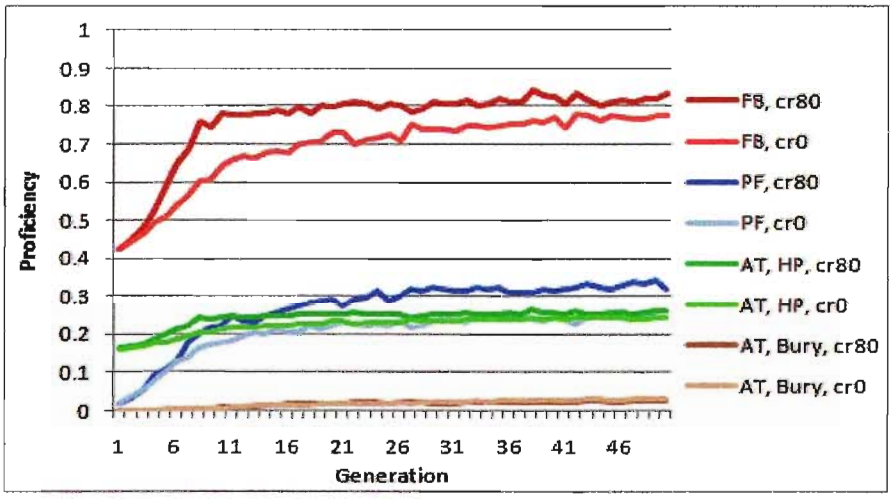

Figure 4.16: Comparison of mean convergence between 80% crossover and 0% crossover using populations of 50 individuals. Darker colours represent 80% crossover (cr80) and lighter colours represent 0% crossover (cr0).

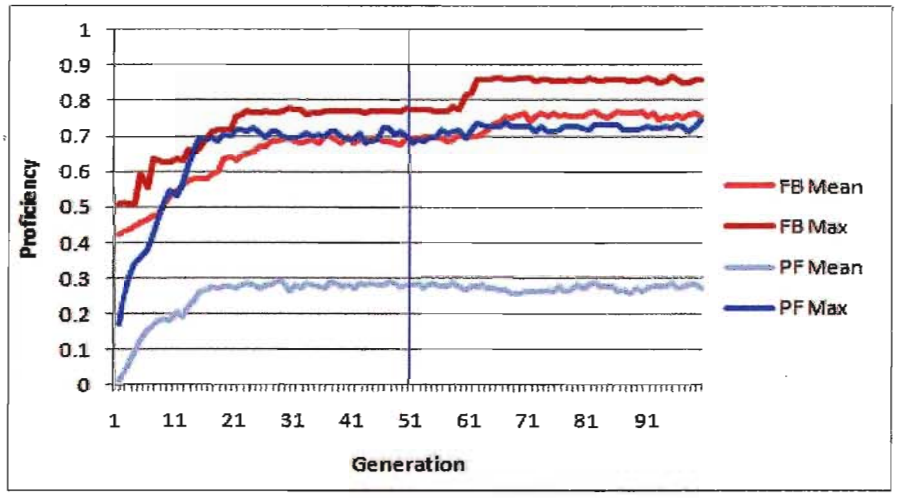

Figure 4.17: Max and mean proficiency of each type of rescue team utilizing a single population and the language of step 1. FB values are indicated with red lines, PF are blue lines.

Appropriateness, Step 1

Building on the well-developed results of step 0, step 1 aims to complete the evolution of rescue behaviours by adding primitives that deal with the more complicated steps of the agent's tasks. The expected behaviours of this step are continuations or additions to the pre-existing 'goto-and-act' behaviours. For FBs the expected behaviour is extended to include returning to a refuge (and waiting there) to refill the water tanks. ATs are expected to load the civilians they unbury, and transport them to a refuge before unloading them. The task of PF agents remains unchanged. Figures 4.17 through 4.20 illustrate the proficiencies of each agent type for the single and multiple population experiments. The vertical lines in these figures represent the transition from step 0 to step 1, and the sudden drop in the multiple population maximum value curves at this transition is due to the loss of the elitism individuals. The nearly flat mean convergence curves in these figures illustrates the lack of improvement found in this step. Examination of the trees developed reveals that the added primitives were not used successfully and were often found amidst bloat rather than effective structures.

Comparison with RCR Competition Results

For completeness, the best individual from each run of the past two steps was tested on a number of maps previously used in competition. Specifically the individuals evolved here are compared to the results of the first three days of the 2006 RCR competition[2]. Recall, however, that the

[2]The results from 2006 are used because, as of writing this, they are the most complete and readily available results and map sets available for download that use v0.49.9 of the RCRSS

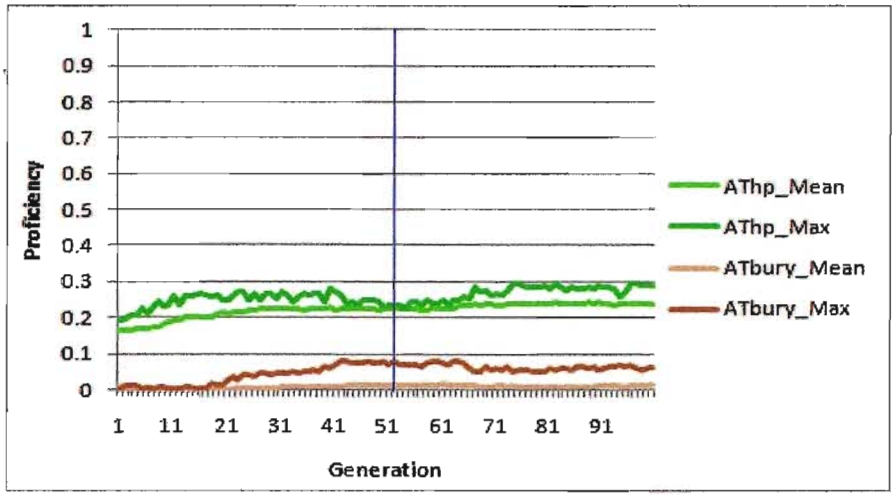

Figure 4.18: Max and mean proficiency of AT agents utilizing a single population and the language of step 1. The tasks are preserving civilian HP (seen in green), and removing buried civilians (seen in brown).

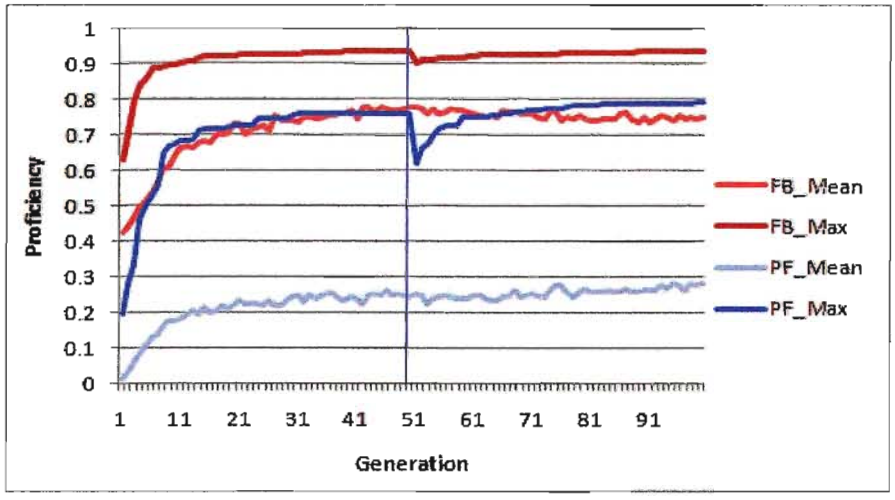

Figure 4.19: Max and mean proficiency of each type of rescue team utilizing multiple populations and the language step 1. FB values are indicated with red lines, PF are blue lines.

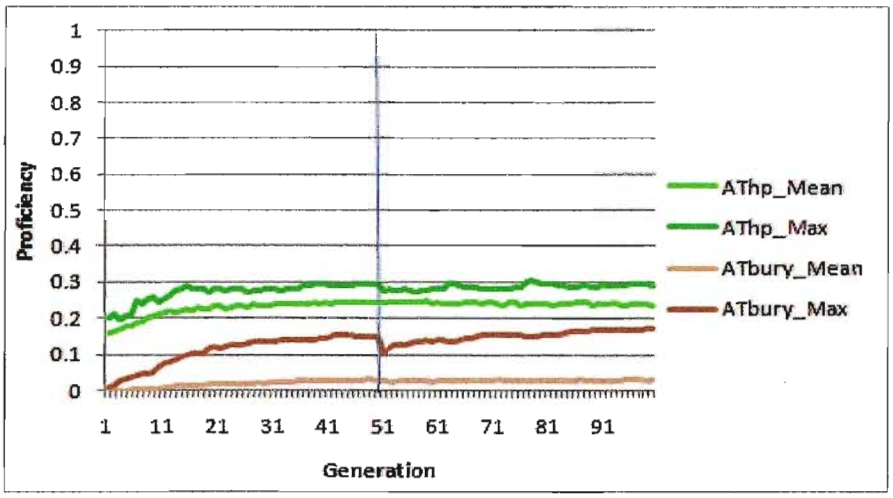

Figure 4.20: Max and mean proficiency of AT agents utilizing multiple populations and the language of step 1. The tasks are preserving civilian HP (seen in green), and removing buried civilians (seen in brown).

goal of steps 0 and 1 are the evolution of the operational behaviours required by the RCRSS, not strategizing behaviours. The results of the competing teams on the other hand, most likely presume this operational behaviour in the process of designing their strategies. The results of the comparison are presented in Tables 4.7 through 4.9, sorted alphabetically by team with the results of this thesis using single and multiple populations for steps 0 and 1, separated at the bottom. Each column represents one map (i.e. one problem instance) specified by the organizers of the competition. During competition each team's solution is run once on the given map, and their score is recorded. For the purposes of experimentation, ten populations exist (one per run), and the best individual is selected from each population and tested on the competition maps (for the multiple-subpopulation experiments the best of each type of agent was selected). The experimental results presented here are the best scores of those 10 individuals. From this comparison it can be seen that the evolved solutions are not on par with the highly competitive solutions prepared by the teams in the RCR league. However, on a number of maps the evolved solutions do achieve scores that surpass the lowest few competitor scores, suggesting that there is potential for GP in this domain given some further evolution and focus into development of strategizing behaviours.

Day1	Kobe1	VC1	Foligno1	Kobe2
AladdinRescue	97.13	71.44	81.91	73.93
Bam	85.31	47.2	63.66	40.18
CSU_YunLu	46.75	65.57	72.71	50.97
DAMAS-Rescue	46.18	49.53	51.23	38.85
FCPortugal	80.1	52.73	50.11	58.33
Hinomiyagura	96.48	39	48.46	28.57
Impossibles	106.4	60.41	63.03	56.49
Incredibles	93.6	64.63	82.42	64.88
ITANDROIDS	77.49	62.88	67.54	47.61
IUST	104.5	79.52	97.19	92.38
Kosar	38.89	65.1	86.72	69.11
KSHITIJ	64.64	56.09	69.67	41.55
MRL	107.8	70.71	101.3	77.01
NITRescue06	70.9	39.49	65.49	44.26
Persia	88.77	70.57	64.37	51.5
Poseidon	104.1	75.36	86.64	83.77
RoboAkut	77.05	72.37	62.92	44.43
S.O.S	105.2	80.07	93.23	58.97
SBCe_Saviour	107.4	81.64	85.25	88.14
Single, Step0	37.72	28.97	38.12	22.37
Multiple, Step0	32.69	28.23	40.60	21.40
Single, Step1	38.53	28.10	38.91	22.43
Multiple, Step1	38.59	25.66	41.64	20.69

Table 4.7: Comparison of step 0 and step 1 scores to past RCR competitors (2006 competition, preliminaries day 1).

Day2	VC2	Random1	Kobe4_1	Random2	Kobe4_2
AladdinRescue	62.23	83.97	62.84	15.42	59.41
Bam	43.63	45.11	75.13	12.87	64.5
CSU_YunLu	47.05	35.27	78.45	17	71.74
DAMAS-Rescue	43.79	67.35	61.75	12.52	72.04
FCPortugal	47.97	24.12	67.81	14.48	66.94
Hinomiyagura	50.11	17.19	62.29	10.62	58.62
Impossibles	86	83.91	83.53	22.72	76.69
Incredibles	77.05	63.27	64.99	24.44	71.25
ITANDROIDS	58.22	60.69	82.47	18.81	76.2
IUST	92.57	63.77	101.9	25.37	93.32
Kosar	82.43	64.18	97.06	18.08	88.05
KSHITIJ	60.04	47.78	70.83	14.76	69.83
MRL	80.29	87.98	88.68	33.32	93.33
NITRescue06	41.68	43.62	73.32	12.56	67.15
Persia	76.22	60.56	81.44	17.55	76.51
Poseidon	82.86	95.33	89.53	0	102.1
RoboAkut	76.29	58.71	84.52	21.68	77.06
S.O.S	92.01	83.79	90.4	25.99	88.46
SBCe_Saviour	99.7	92.98	88.99	28.26	97.02
Single, Step0	30.11	20.21	63.37	9.91	57.69
Multiple, Step0	31.73	20.52	62.80	10.50	55.51
Single, Step1	28.62	22.66	63.71	9.91	58.26
Multiple, Step1	29.72	20.48	62.74	9.16	55.89

Table 4.8: Comparison of step 0 and step 1 scores to past RCR competitors (2006 competition, preliminaries day 2.

Day3	Kobe3	Random3	Random4	Random5	TOTAL
AladdinRescue	60.92	106.98	95.459	60.59	932.23
Bam	52.82	95.921	80.072	54.624	761.03
CSU_YunLu	71.81	90.028	80.157	60.249	787.75
DAMAS-Rescue	58.88	102.1	77.468	52.942	734.64
FCPortugal	63.56	85.904	70.534	47.024	729.63
Hinomiyagura	55.19	67.924	47.514	39.727	621.72
Impossibles	76.14	110.27	98.452	70.727	994.76
Incredibles	74.57	103.24	94.234	70.014	948.60
ITANDROIDS	68.46	103.41	56.372	65.686	845.83
IUST	84.88	108.33	99.524	82.699	1125.91
Kosar	78.51	106.25	97.477	67.68	959.55
KSHITIJ	59.1	89.903	83.18	0	727.37
MRL	82.72	110.31	98.49	82.481	1114.45
NITRescue06	62.47	87.494	80.427	52.997	741.87
Persia	72.92	100.09	89.084	70.012	919.61
Poseidon	87.52	109.44	98.298	82.763	1097.65
RoboAkut	75.05	106.39	91.397	0	847.87
S.O.S	74.94	107.24	74.902	77.513	1052.68
SBCe_Saviour	73.52	108.22	96.487	87.895	1135.47
Single, Step0	55.20	58.58	49.96	40.51	512.72
Multiple, Step0	55.10	60.71	46.98	37.67	504.43
Single, Step1	55.55	66.46	53.18	39.13	525.45
Multiple, Step1	53.73	56.73	49.54	37.72	502.27

Table 4.9: Comparison of step 0 and step 1 scores to past RCR competitors (2006 competition, preliminaries day 3.

Chapter 5

Conclusions and Future Work

Genetic Programming is termed an 'Automatic Programming Technique' due to its process of developing program-structured solutions, yet there is a considerable amount of manual effort involved in the design and initialization of a GP system. Perhaps most notably so in the design and implementation of the GP language. It seems that the utility of Genetic Programming is in bridging the gap between a 'natural' description of the problem and a desirable description of the solution. In order to hierarchically decompose a problem into smaller sub-problems, or even to have GP focus on sub-areas of a problem the manual effort in designing the scenario begins to outweigh the amount of effort saved by having GP cover the remaining distance. So the design of simpler methods to improve GP's capabilities is a necessary part of utilizing GP as a means for the automatic creation of complex solutions such as control of autonomous agents. To this end, this thesis introduces the step-wise learning paradigm in order to initially reduce the size of the search space, and gradually expand it while maintaining focus on known good areas of search. It is found that when compared to a standard GP the step-wise procedure is capable evolving more rapidly and to better results. In a complex search space that seems unyielding to a standard GP, by the judicious selection of a simpler initial search space the step-wise learning was capable of first evolving simple behaviours and building upon them. While it is within the ability of a GP algorithm to ignore useless primitives in a language, for a problem with such heavy constraints and large search space as the RCR, it is infeasible to do so.

This thesis applied Genetic Programming to an environment with a very rugged fitness landscape. This means that small changes in the solution can lead to large changes in the fitness of that solution. This requires a very fine grained search through the solution space so that small and isolated areas of good solutions are not skipped over entirely. To further complicate this, the strict time-line and costly evaluation forced the GP system to be developed to evaluate fewer samples from the solution space. Despite these difficult parameters some reasonable solutions to the problem were evolved creating the basic expected decision logic as well as some unexpected behaviours. However, mixed

success was observed as the core rescue behaviours were developed, but not the extended rescue behaviours.

It was found that control of Fire Brigade (FB) agents was the most readily evolved of the three types of agents. When utilizing a single population where individuals consisted of a tree for each type of agent, the FB agents shaped the evolution of the entire individual. It was found that a more expressive fitness function lead to better solutions, allowing the Police Force (PF) agents to evolve their own behaviours. Ambulance Teams remained unevolved until a correction to the fitness function motivated the development of simple rescuing behaviours but the overall task of preserving civilian health remained elusive.

A comparison between a single population of individuals containing three trees and multiple co-evolved subpopulations each containing a single tree is carried out alongside the evolution of behaviours to ascertain if either of these benefit the evolution. While the multiple sub-population is capable of maintaining higher maximum solutions and faster convergence due to the presence of elitism specific to each type of agent, it is determined that the mean quality of solutions remains unchanged within the set limit of generations.

The expensive operation in this GP system is the evaluation of an individual. Given the setup of this system, three trees can be simultaneously evaluated, whether those three trees all belong to the same individual or three separate individuals. In the case of those three trees belonging to separate individuals from three separate populations, three times the number of selection operators are applied, and on a much more specific comparison. This suggests that the multiple population experiments have the advantage of a more accurate selection without a significant increase in execution time.

Ongoing and Future Work

Although this thesis shows that GP has potential for use in developing rescue agent control structures, a number of experiments that were anticipated to further explore the suitability and effectiveness of the application of GP to the RCR domain could not be completed. The challenge that the RCR domain poses to the evolution of solutions was much greater than what was initially thought, and thus the time constraints would not allow for the completion of all experiments. A number of experiments have been started, and will be completed as ongoing and future work.

In order to proceed with the remaining steps that were presented in Section 3.4 but not completed, the core structure of the agent logic must first be in place. Since the GP system developed here was unable to evolve the second (more complicated) variation of operational behaviours, manually constructed trees were added to the population to bias the search to that area of space. Seeding the population with a small percentage of expected solutions makes sense in this situation as the expected solution comprises the necessary structure required by the RCRSS domain, which can then be built upon in the steps that follow. This was not done initially for the purposes of exploring GP's

ability to create this structure on its own, but for the sake of examining the remaining behaviours in the absence of evolved solutions, a seed population is a reasonable alternative. To create this seeded population, five trees of each type (10% of the population) from the evolved population resulting from step 1 were replaced by manually constructed trees. These will be evolved for a full 50 generations to create a population that is essentially converged.

From here the exploration and target selection behaviours will be examined. Both of these behaviours (three steps) are expected to improve the quality of the solution trees. In the case of exploration, the evolved behaviours will be examinable in terms of an expected solution similar to the previous steps. Also the use of explorative behaviours in proper conjunction with the basic behaviours expected from the previous steps should have a positive influence on proficiency and score as more targets are found and acted upon. The target selection behaviours on the other hand come with no expected solution. Thus, the changes in proficiency and score will constitute the effectiveness of these steps.

Finally the cooperative behaviours developed following each of the previous steps will be examined. Similar to the exploration behaviour, proper use of the cooperation primitives should have noticeable behaviours, and improvements to the performance of those individuals utilizing them.

At first consideration, following the results of steps 0 and 1 it seems unreasonable to expect GP to perform well in the steps mentioned above. However there is a clear distinction between the types of tasks of the first two steps and that of the remaining steps. In the first two steps the goal of the GP system was to evolve individuals that perform the task expected of them in the environment. Without this ability, agents often have little to no effect on the fitness of the GP individual. Following the creation of a population that has these abilities, the goal of the remaining steps of evolution is to adjust and ideally enhance these already existent behaviours. So while the former consists of a population of sharp contrast between somewhat fit and completely unfit individuals, the latter ideally consists of a population of gradual improvements concentrated in an area of the search space known to contain competent solutions.

Bibliography

[1] *RoboCup-Rescue Simultaor Manual version 0 revision 4*, The Robocup Rescue Technical Committee, July 2000. [Online]. Available: http://www.robocuprescue.org/docs/robocup-manual-v0-r4.pdf

[2] M. Chen, K. Dorer, E. Foroughi, F. Heintz, Z. Huang, S. Kapetanakis, K. Kostiadis, J. Kummeneje, J. Murray, I. Noda, O. Obst, P. Riley, T. Steffens, and Y. W. X. Yin, *RoboCup Soccer Server User Manual version 7.07 and later*, January 2003. [Online]. Available: http://sourceforge.net/projects/sserver/files/Manual\%20Dev\%20Snapshot/9-20030211/manual-20030211.pdf/download

[3] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA, USA: MIT Press, 1992.

[4] L. A. Panait and S. Luke, "Learning ant foraging behaviours," in *Artificial Life XI Ninth International Conference on the Simulation and Synthesis of Living Systems*, J. Pollack, M. Bedau, P. Husbands, T. Ikegami, and R. A. Watson, Eds. Boston, Massachusetts: The MIT Press, 12-15 Sep. 2004, pp. 575–580.

[5] S. Luke and L. Spector, "Evolving teamwork and coordination with genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Stanford University, CA, USA: MIT Press, 28–31 Jul. 1996, pp. 150–156. [Online]. Available: http://www.cs.gmu.edu/~sean/papers/cooperation.pdf

[6] T. Francisco and G. M. J. dos Reis, "Evolving predator and prey behaviours with co-evolution using genetic programming and decision trees," in *GECCO-2008 Workshop: Defense Applications of Computational Intelligence (DAC)*, M. Ebner, M. Cattolico, J. van Hemert, S. Gustafson, L. D. Merkle, F. W. Moore, C. B. Congdon, C. D. Clack, F. W. Moore, W. Rand, S. G. Ficici, R. Riolo, J. Bacardit, E. Bernado-Mansilla, M. V. Butz, S. L. Smith, S. Cagnoni, M. Hauschild, M. Pelikan, and K. Sastry,

Eds. Atlanta, GA, USA: ACM, 12-16 Jul. 2008, pp. 1893–1900. [Online]. Available: http://www.cs.bham.ac.uk/~wbl/biblio/gecco2008/docs/p1893.pdf

[7] A. Bilski and S. Kakuchi, "Kobe earthquake," *Maclean's*, January 1995.

[8] T. Morimoto, *How to Develop a RoboCupRescue Agent for RoboCupRescue Simulation System version 0*, 1st ed., The Robocup Rescue Technical Committee. [Online]. Available: http://www.robocuprescue.org/docs/yab-api-manual-1_00.pdf

[9] M. O'Neill, R. Poli, W. B. Langdon, and N. F. McPhee, "A field guide to genetic programming," *Genetic Programming and Evolvable Machines*, March 2008. [Online]. Available: http://dx.doi.org/10.1007/s10710-008-9073-y

[10] "Genetic programming homepage," last accessed July 21, 2010. [Online]. Available: http://www.genetic-programming.org

[11] S. Luke, C. Hohn, J. Farris, G. Jackson, and J. Hendler, "Co-evolving soccer softbot team coordination with genetic programming," in *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997. [Online]. Available: http://www.cs.gmu.edu/~sean/papers/robocupc.pdf

[12] R. Crawford-Marks, L. Spector, and J. Klein, "Virtual witches and warlocks: A quidditch simulator and quidditch-playing teams coevolved via genetic programming," in *Late Breaking Papers at the 2004 Genetic and Evolutionary Computation Conference*, M. Keijzer, Ed., Seattle, Washington, USA, 26 Jul. 2004. [Online]. Available: http://www.cs.bham.ac.uk/~wbl/biblio/gecco2004/LBP046.pdf

[13] A. Hauptman and M. Sipper, "GP-endchess: Using genetic programming to evolve chess endgame players," in *Proceedings of the 8th European Conference on Genetic Programming*, ser. Lecture Notes in Computer Science, M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, Eds., vol. 3447. Lausanne, Switzerland: Springer, 30 Mar. - 1 Apr. 2005, pp. 120–131. [Online]. Available: http://www.cs.bgu.ac.il/~sipper/papabs/eurogpchess-final.pdf

[14] J. R. Koza, "Genetic evolution and co-evolution of computer programs," in *Artificial Life II*, ser. SFI Studies in the Sciences of Complexity, C. T. C. Langton, J. D. Farmer, and S. Rasmussen, Eds. Santa Fe Institute, New Mexico, USA: Addison-Wesley, Feb. 1990 1991, vol. X, pp. 603–629. [Online]. Available: http://www.genetic-programming.com/jkpdf/alife1990.pdf

[15] J. Grefenstette, "Credit assignment in rule discovery systems based on genetic algorithms," *Machine Learning*, vol. 3, no. 2, pp. 225–245, 1988.

[16] T. Haynes, S. Sen, D. Schoenefeld, and R. Wainwright, "Evolving a team," in *Working Notes for the AAAI Symposium on Genetic Programming*, E. V. Siegel and J. R. Koza,

Eds. MIT, Cambridge, MA, USA: AAAI, 10–12 Nov. 1995, pp. 23–30. [Online]. Available: http://www.mcs.utulsa.edu/~rogerw/papers/Haynes-team.pdf

[17] J. R. Koza, *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge Massachusetts: MIT Press, May 1994.

[18] J. R. Koza, D. Andre, F. H. Bennett III, and M. A. Keane, "Use of automatically defined functions and architecture-altering operations in automated circuit synthesis using genetic programming," in *Genetic Programming 1996: Proceedings of the First Annual Conference*, J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds. Stanford University, CA, USA: MIT Press, 28–31 Jul. 1996, pp. 132–149. [Online]. Available: http://www.genetic-programming.com/jkpdf/gp1996adfaa.pdf

[19] J. R. Koza and R. Poli, "Genetic programming," in *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, E. K. Burke and G. Kendall, Eds. Springer, 2005, ch. 5, pp. 127–164. [Online]. Available: http://cswww.essex.ac.uk/staff/poli/papers/KozaPoli2005.pdf

[20] J. R. Koza, "Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations," in *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, J. R. McDonnell, R. G. Reynolds, and D. B. Fogel, Eds. San Diego, CA, USA: MIT Press, 1-3 Mar. 1995, pp. 695–717. [Online]. Available: http://www.genetic-programming.com/jkpdf/ep1995.pdf

[21] J. P. Rosca and D. H. Ballard, "Genetic programming with adaptive representations," University of Rochester, Computer Science Department, Rochester, NY, USA, Tech. Rep. TR 489, Feb. 1994. [Online]. Available: ftp://ftp.cs.rochester.edu/pub/papers/robotics/94.tr489.Genetic_programming_with_adaptive_representations.ps.Z

[22] P. Stone and M. Veloso, "Layered learning," in *Machine Learning: ECML 2000: 11th European Conference on Machine Learning, Barcelona, Catalonia, Spain, May/June 2000. Proceedings.* Springer, 2000, pp. 231–248.

[23] S. M. Gustafson, "Layered learning in genetic programming for a co-operative robot soccer problem," Master's thesis, Kansas State University, Manhattan, KS, USA, Dec. 2000. [Online]. Available: http://www.cs.nott.ac.uk/~smg/research/publications/msthesis-2000.ps

[24] B.-T. Zhang and D.-Y. Cho, "Evolving complex group behaviors using genetic programming with fitness switching," *Artificial Life and Robotics*, vol. 4, no. 2, pp. 103–108, 2000. [Online]. Available: http://bi.snu.ac.kr/Publications/Journals/International/AROB4-2.ps

[25] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Autonomous Agents and Multi-Agent Systems*, vol. 11, no. 3, pp. 387–434, 2005.

[26] S. Raik and B. Durnota, "The evolution of sporting strategies," in *Complex Systems: Mechanisms of Adaption*, R. J. Stonier and X. H. Yu, Eds. IOS Press, 1994, pp. 85–92. [Online]. Available: http://www.csse.monash.edu.au/publications/1994/sd-P94-1.ps.gz

[27] C. Yong and R. Miikkulainen, "Coevolution of Role-Based Cooperation in Multiagent Systems," *IEEE Transactions on Autonomous Mental Development*, vol. 1, no. 3, pp. 170–186, 2009.

[28] K. Iwata, N. Ito, K. Toda, and N. Ishii, "Analysis of Agents Cooperation in RoboCupRescue Simulation," *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pp. 227–236, 2008.

[29] J. Kummeneje, "Robocup as a means to research, education, and dissemination," Ph.D. dissertation, Department of Computer and Systems Sciences, Stockholm University and the Royal Institute of Technology, June 2003.

[30] S. Luke, "Genetic programming produced competitive soccer softbot teams for robocup97," in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, Eds. University of Wisconsin, Madison, Wisconsin, USA: Morgan Kaufmann, 22-25 Jul. 1998, pp. 214–222. [Online]. Available: http://www.cs.gmu.edu/~sean/papers/robocupgp98.pdf

[31] P. Wilson, "Development of genetic programming strategies for use in the robocup domain," Honours Thesis, Department of Computer Science, Royal Melbourne Institute of Technology, October 1998.

[32] J. Aronsson, "Genetic programming of multi-agent system in the robocup domain," Master's thesis, Lund Institute of Technology, Sweden, 2003.

[33] A. Kleiner, M. Brenner, T. Brauer, and C. Dornhege, "Successful search and rescue in simulated distaster areas," *RoboCup 2005*, pp. 323–334, 2006.

[34] S. Paquet, N. Bernier, and B. Chaib-draa, *DAMAS-Rescue 2004 Report*, Laval University, Canada, 2004.

[35] J. Habibi, A. Fathi, S. Hassanpour, M. Ghodsi, B. Sadjadi, H. Vaezi, and M. Valipour, *Impossibles Team Description*, Sharif University of Technology, Tehran, Iran, February 2005.

[36] J. Habibi, M. Ghodsi, H. Vaezi, M. Valipour, S. Aliari, and N. Hazar, *Impossibles RoboCup Rescue 2006 Team Description Paper*, Sharif University of Technology, Tehran, Iran, April 2006.

[37] J. Habibi, A. Nowroozi, A. B. Farahany, M. Habibi, S. H. Yeganeh, S. H. Mortazavi, M. Salehe, and M. Vafadoost, *Impossibles 2007 Team Description*, Sharif University of Technology, Tehran, Iran, 2007.

[38] I. C. Martinez, D. Ojeda, and E. A. Zamora, "Ambulance decision support using evolutionary reinforcement learning in robocup rescue simulation league," *RoboCup2006*, pp. 556–563, 2007.

[39] D. J. Montana, "Strongly typed genetic programming," Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, BBN Technical Report #7866, Mar. 1994. [Online]. Available: http://www.cs.ucl.ac.uk/staff/W.Langdon/ftp/ftp.io.com/papers/stgp2.ps.Z

[40] S. Russell, P. Norvig, J. Canny, J. Malik, D. Edwards, and S. Thrun, *Artificial Intelligence: A Modern Approach*. Prentice hall Englewood Cliffs, NJ, 2003.

[41] W. Spears and V. Anand, "A study of crossover operators in genetic programming," *Methodologies for Intelligent Systems*, pp. 409–418, 1991.

[42] S. Luke and L. Spector, "A comparison of crossover and mutation in genetic programming," in *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds. Stanford University, CA, USA: Morgan Kaufmann, 13-16 Jul. 1997, pp. 240–248. [Online]. Available: http://www.cs.gmu.edu/~sean/papers/comparison/comparison.pdf

Appendix A

Best Trees of Step 0

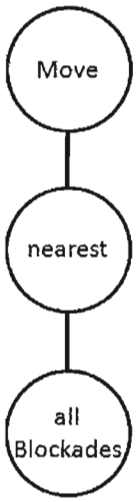

Figure A.1: AT Single Population Fitness_0 Best Tree

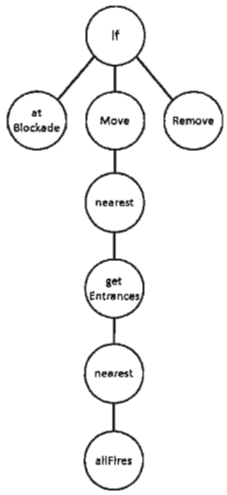

Figure A.2: AT Single Population Fitness_5 Best Tree

Figure A.3: AT Multiple Population Fitness_5 Best Tree

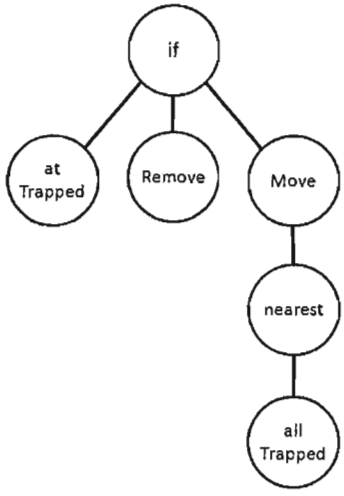

Figure A.4: AT Single Population Fitness_7 Best Tree

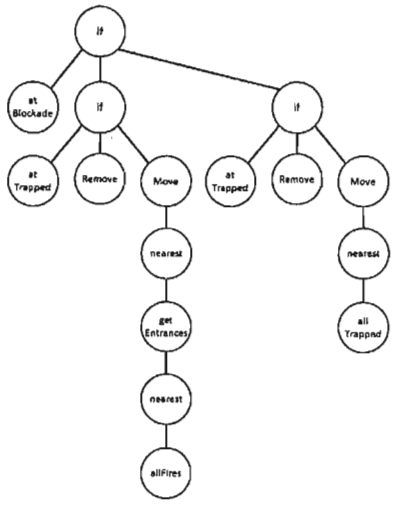

Figure A.5: AT Multiple Population Fitness_7 Best Tree

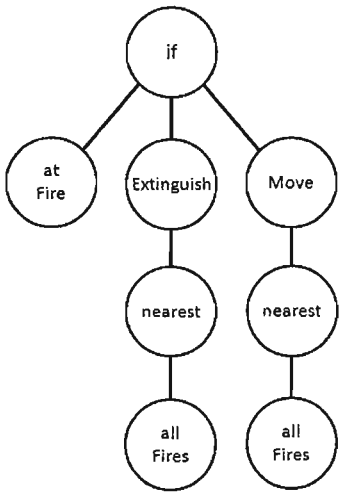

Figure A.6: FB Single Population Fitness_0 Best Tree

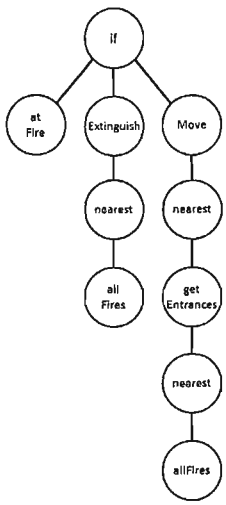

Figure A.7: FB Single Population Fitness_5 Best Tree

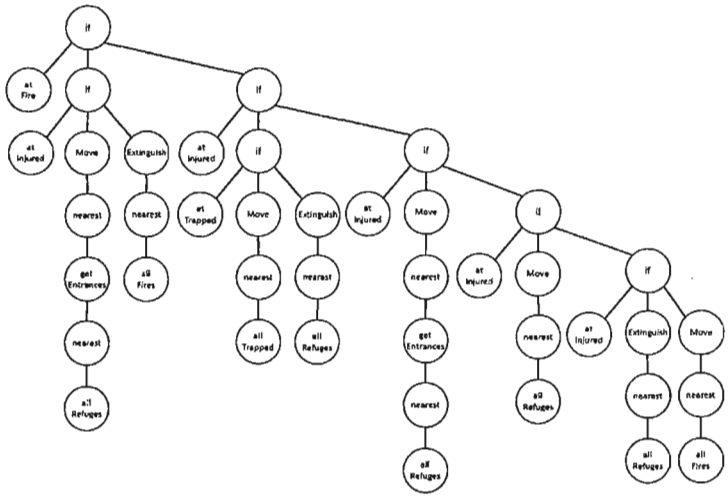

Figure A.8: FB Multiple Population Fitness$_5$ Best Tree

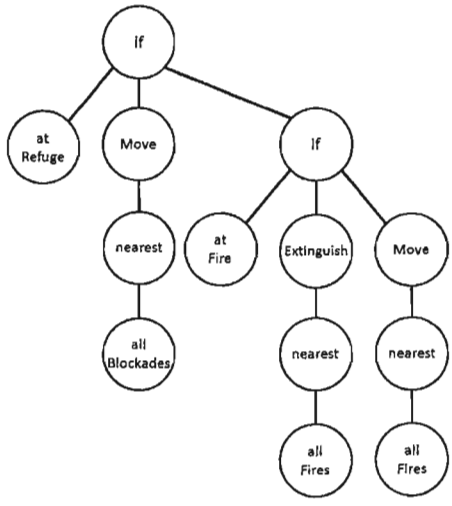

Figure A.9: FB Single Population Fitness$_7$ Best Tree

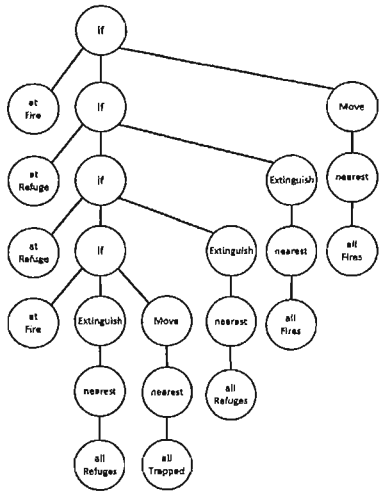

Figure A.10: FB Multiple Population Fitness$_7$ Best Tree

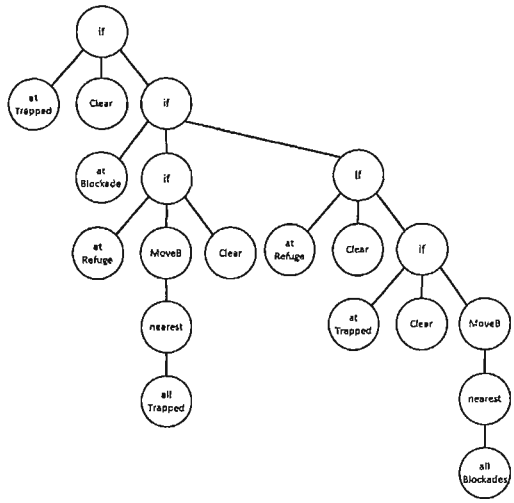

Figure A.11: PF Single Population Fitness$_0$ Best Tree

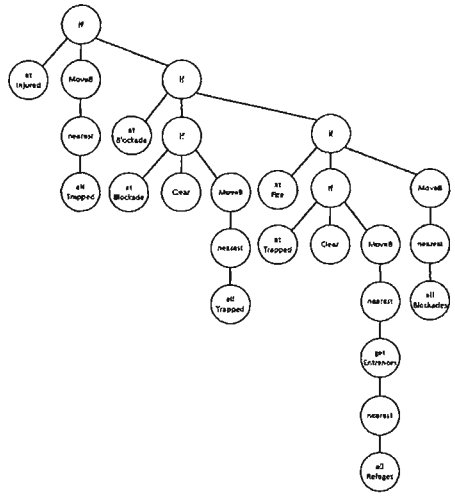

Figure A.12: PF Single Population Fitness$_5$ Best Tree

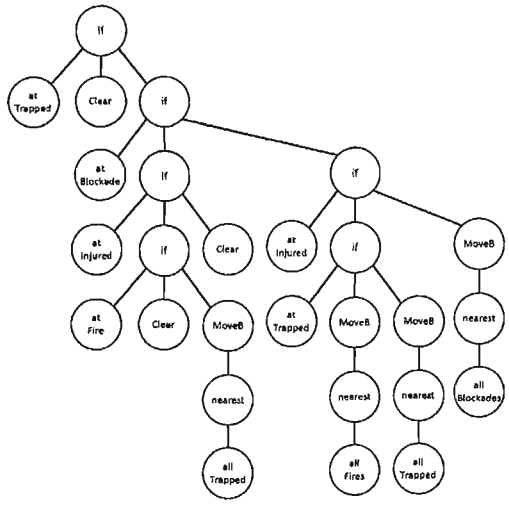

Figure A.13: PF Multiple Population Fitness$_5$ Best Tree

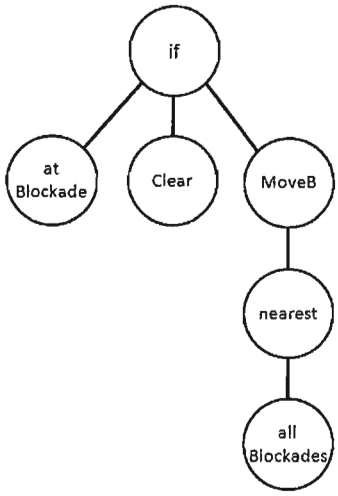

Figure A.14: PF Single Population Fitness$_7$ Best Tree

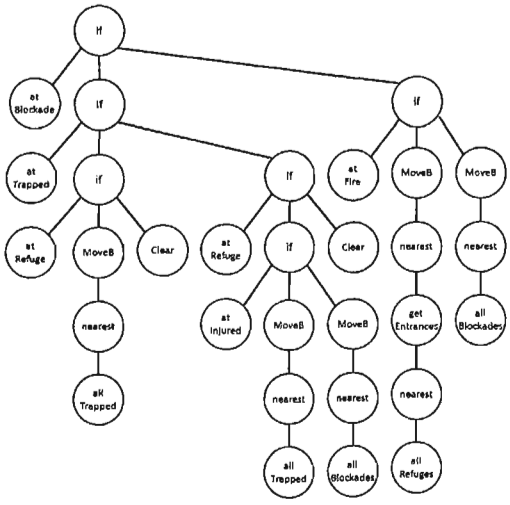

Figure A.15: PF Multiple Population Fitness$_7$ Best Tree