FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

# Automatic Generation of 2-AntWars Players with Genetic Programming

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Computational Intelligence**

eingereicht von

**Johannes Inführ**
Matrikelnummer 0625654

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Univ.-Prof. Dr. Günther R. Raidl

Wien, 19.07.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

## Erklärung zur Verfassung der Arbeit

Inführ Johannes
Kaposigasse 60, 1220 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19.07.2010

_____

(Inführ Johannes)

## Acknowledgements

In no particular order I would like to thank Univ.-Prof. Dr. Günther R. Raidl for allowing me to do research on a topic of my own choice and his continued support and encouragement during the creation of this thesis. The advice that stuck with me the most was that voodoo is never a satisfactory explanation of strange software errors. My girlfriend and my family have my eternal gratitude for letting me code day in and day out, encouraging me when the work seemed never ending and paying the electricity bill I racked up, and of course big thanks to everyone who refrained from dousing me in insecticide whenever I was talking nonstop about swarming ants.

**Abstract**

In the course of this thesis, the feasibility of automatically creating players for the game 2-AntWars is studied. 2-AntWars is a generalization of AntWars which was introduced as part of a competition accompanying the Genetic and Evolutionary Computation Conference 2007. 2-AntWars is a two player game in which each player has control of two ants on a playing field. Food is randomly placed on the playing field and the task of the players is to collect more food than the opponent.

To solve this problem a model of the behaviour of a 2-AntWars player is developed and players are built according to this model by means of genetic programming, which is a population based evolutionary algorithm for program induction. To show the feasibility of this approach, the players are evolved in an evolutionary setting against predefined strategies and in a coevolutionary setting where both players of 2-AntWars evolve and try to beat each other.

Another core part of this thesis is the analysis of the evolutionary and behavioural dynamic emerging during the development of 2-AntWars players. This entails specific characteristics of those players (e.g. which ant found how much food) and on a higher level their behaviour during games and the adaption to the behaviour of the opponent.

The results showed that it is indeed possible to create successful 2-AntWars players that are able to beat fixed playing strategies that oppose them. This is a solution to an important problem of game designers as a well balanced game needs to have a feasible counter strategy to every strategy and with the help of the proposed method such counter strategies can be found automatically.

The attempt to create 2-AntWars players from scratch by letting the developed players battle each other was also successful. This is a significant result as it shows how to automatically create artificial intelligence for games (and in principle for any problems that can be formulated as games) from scratch.

The developed solutions to the 2-AntWars problem were surprisingly diverse. Ants were used as bait, were hidden or shamelessly exploited weaknesses of the opponent. The population model that was chosen enabled the simultaneous development of players with different playing strategies inside the same population without resorting to any special measures normally associated with that like explicitly protecting a player using one strategy from a player using another one. Both mutation and crossover operators were shown to be essential for the creation of high performing 2-AntWars players.

**Zusammenfassung**

Im Rahmen dieser Arbeit wird die Möglichkeit der automatischen Generierung von Spielern für das Spiel 2-AntWars untersucht. 2-AntWars ist eine Generalisierung von AntWars. AntWars wurde für einen Wettbewerb der Genetic and Evolutionary Computation Converence 2007 erfunden. 2-AntWars ist ein Spiel für zwei Spieler, wobei jeder Spieler die Kontrolle über zwei Ameisen auf einem Spielfeld hat. Auf diesem Spielfeld ist Futter an zufälligen Orten platziert und die Aufgabe der Spieler ist es, mehr Futter zu finden als der jeweilige Gegner.

Um das Problem zu lösen wird ein Modell für das Verhalten eines 2-AntWars Spielers entwickelt und Genetic Programming, eine populationsbasierte evolutionäre Methode zur Programminduktion, wird verwendet um Spieler basierend auf diesem Modell zu erstellen. Die Machbarkeit dieses Ansatzes wird gezeigt, indem Spieler sowohl per Evolution im Kampf gegen fixe Spielstrategien als auch per Koevolution im Kampf gegeneinander entwickelt werden.

Ein weiterer Kernpunkt dieser Arbeit ist die Analyse der Dynamik die während der Entwicklung der Spieler auftritt, sowohl von der evolutionären Perspektive als auch von den zur Schau gestellten Verhaltensweisen der Spieler her. Das beinhaltet spezielle Eigenschaften der Spieler (wie zum Beispiel welche Ameise wieviel Futter sammelt) aber auch die Strategien der Spieler auf höherer Ebene und wie sie sich an ihre Gegner anpassen.

Die Ergebnisse zeigen, dass es in der Tat möglich ist erfolgreiche 2-AntWars Spieler zu erzeugen die in der Lage sind, fixe Strategien ihrer Gegner zu schlagen. Das ist ein Resultat das vor allem für Spieldesign-Probleme wichtig ist, da es für eine gute Spiel-Balance unumgänglich ist, dass für jede Spielstrategie eine Gegenstrategie existiert. Mit Hilfe der dargelegten Methode ist es möglich, solche Gegenstrategien automatisiert aufzufinden.

Der Versuch 2-AntWars Spieler von Grund auf durch Spiele gegeneinander zu entwickeln war ebenfalls von Erfolg gekrönt. Das zeigt, dass es möglich ist, künstliche Intelligenz für Spiele (und im Prinzip für alle Probleme die als Spiele formuliert werden können) zu erzeugen, ohne Spielstrategien von Hand entwerfen zu müssen.

Die Verhaltensweisen die die entwickelten 2-AntWars Spieler an den Tag legten waren überraschend vielfältig. Ameisen wurden als Köder verwendet, versteckt und wurden generell verwendet um Schwächen im Spiel des Gegners schamlos auszunutzen. Das gewählte Populationsmodell machte die simultane Entwicklung von Spielern mit verschiedenen Spielstrategien in derselben Population möglich, ohne dies explizit zu fördern, beispielsweise indem Spieler einer Strategie vor Spielern einer anderen Strategie geschützt werden. Es zeigte sich, dass sowohl Mutations- als auch Crossover-Operationen für die Entwicklung von leistungsfähigen 2-AntWars Spielern notwendig sind.

# Contents

**Abstract** ii

**Zusammenfassung** iii

**Contents** v

**I Introduction** 1

**1 Introduction** 3

**2 Genetic Programming and Coevolution** 5

**3 2-AntWars** 11
3.1 AntWars Rules . . . . . 11
3.2 2-AntWars Rules . . . . . 12
3.3 Strategies . . . . . 14

**II Genetic Programming System** 17

**4 Genetic Programming System** 19
4.1 The GP-Algorithm . . . . . 19
4.2 Individual Structure . . . . . 19
4.3 Population Model . . . . . 21
4.4 Population Initialization . . . . . 22
4.5 Selection . . . . . 22
4.6 Crossover . . . . . 22
4.7 Mutation . . . . . 23
4.8 Evaluation . . . . . 24

**5 Modelling the 2-AntWars Player** 25
5.1 Data Types . . . . . 26
5.2 Available Statements . . . . . 28
5.3 Belief Function . . . . . 33

5.4 Predict Functions . . . 33
5.5 Movement Functions . . . 34
5.6 Decision Function . . . 35
5.7 Settings . . . 36

**III Results** **39**

**6 No Adversary** **41**
6.1 Fitness development . . . 41
6.2 Belief . . . 46
6.3 Prediction . . . 47
6.4 General Performance Observations . . . 48
6.5 Best Individuals . . . 51
6.6 Conclusion . . . 52

**7 Strategies Version 1** **53**
7.1 Greedy . . . 53
7.2 Scorched Earth . . . 60
7.3 Hunter . . . 67

**8 Strategies Version 2** **73**
8.1 Greedy . . . 73
8.2 Scorched Earth . . . 79
8.3 Hunter . . . 84

**9 Coevolutionary Runs** **91**
9.1 Run with Standard Settings . . . 91
9.2 Run with Asymmetric Evaluation . . . 97
9.3 Long Run . . . 101
9.4 Long Run with Asymmetric Evaluation . . . 104

**10 Special Analysis** **105**
10.1 Mixed Opponent Strategies . . . 105
10.2 Stability of Results . . . 107
10.3 Playing against Unknown Opponents . . . 109

**11 Conclusion** **115**

**IV Appendix** **119**

**A Strategy Evaluation** **121**

**Bibliography** **125**

# Part I

# Introduction

CHAPTER 1

# Introduction

The main aim of this thesis is to generalize AntWars [1] to 2-AntWars and to show how to automatically create artificial intelligence capable of playing this new game. 2-AntWars is a two-player game. Each player controls two ants on a rectangular playing field and tries to collect more randomly distributed food than the opponent. Chapter 3 on page 11 describes the rules of 2-AntWars and how they were derived from AntWars in detail.

Being able to automatically generate competent artificial intelligence has a lot of advantages. Since this thesis uses it to play a game, the first group of advantages directly concerns game development. The most obvious one is to use the developed artificial intelligence as opponent for humans in single-player games and skip the complex task of handcrafting an artificial opponent. However, there are equally important uses for automated gameplay during the development of a game. For instance, one of the first steps of creating a game is to define its rules. The rules determine under which conditions certain actions are available to the player. The authors of [2] describe two pitfalls when defining the rules of a game. The first one is that the rules are chosen in a way that a dominant strategy, which is a sequence of actions that always leads to victory, exists. In this situation, the player of the game simply has to execute this strategy to win, no skill or adaption to the current game situation is required. Dominant strategies make a game boring and as a consequence unsuccessful. The second pitfall is the availability of actions that are never advantageous. After the player learns of them he will of course avoid them, making their definition and implementation a waste of time and effort. The only way to avoid those pitfalls (especially for games with complex rules) is to play the game and try to find dominant strategies and useless actions. This is a costly and time intensive process if humans are involved. With a method to automatically create players for a game, the search for dominant strategies and useless actions can be sped up immensely. If a player cannot be beaten by any other player, a dominant strategy has been discovered. If an action is never used by any of the players, a useless action has been uncovered. With an automated method to create players it becomes easier to try a lot of different rules and evaluate their effect on the set of successful strategies. Improved testing of the game implementation is an additional benefit. Salge et al. [2] describe the development of strategies that crashed the game because that meant that they did not lose it. Automatically

created strategies will try everything that might give them an advantage, without being as biased as human players. As a result, game situations that were not anticipated by the game designer and subsequently are not handled correctly by the game logic may arise. This of course does not mean that testing by humans becomes unnecessary, there are whole classes of problems that automatic strategy generation cannot uncover. For example, the method presented in this thesis uses the set of actions that the game rules specify to build strategies. It does not know what the actions are supposed to do, it simply chooses actions that are beneficial. If an action that should be beneficial is actually detrimental because of an implementation error, the developed strategies will try to work around that and the error remains unnoticed.

Automatic generation of artificial intelligence is not only applicable in various stages of game development. It can also be used to solve real world problems, especially if they can be formulated as a game or an agent based description is available. Imagine two competing companies A and B. A wants to lure customers away from B. It has various actions at its disposals. It can improve the own product, start a marketing campaign and place advertisements in various media and at different physical locations or denounce the products of B. B can react in a lot of different ways to this and A wants to be able to anticipate possible reactions. Based on previous attempts to improve the market share, A has an elaborate model of the behaviour of the potential customers. The game is based on this model. A uses its planned strategy as one player and an automatically generated strategy as approximation of the behaviour of B. The company that increases its market share wins. The automatically created strategies for B give A an insight into the weaknesses of its own strategy. The game can also be reversed, the current marketing strategy of B is implemented as a fixed player and strategies of A are automatically developed so that A has a good answer to B's marketing.

The method used in this thesis to automatically create gaming strategies is genetic programming, an evolutionary algorithm that applies the principles of biological evolution to computer programs. Using genetic programming to develop players of a game is not a particularly new idea. Even the first book of John Koza [3] (the inventor of genetic programming) contained the automatic generation of a movement strategy for an ant that that tries to follow a path of food (artificial ant) and a lot of research has been done since then. Already mentioned was the work presented in [2] where genetic programming was used to develop players of a turn based strategy game. In [4] space combat strategies were created. Other forms of predator-prey interaction were analyzed in [5] and [6]. Genetic programming has also been used to develop soccer [7] and chess end game players [8]. However, the conducted research is focused on the end result and emerging evolutionary dynamics that occur during the development are neglected. In this thesis not only the end results of evolution, but also the developments that led to those results will be presented to gain insight into the evolutionary process of genetic programming.

The next chapter will introduce the central concepts of genetic programming. Chapter 3 contains the complete definition of 2-AntWars and a discussion of possible strategies for this game. This is followed by a description of the genetic programming implementation that was used for this thesis in chapter 4 and the 2-AntWars player model in chapter 5. Chapters 6 to 9 contain the main results of this thesis, which are supplemented by experiments reported in chapter 10. A summary and directions for future work can be found in chapter 11.

CHAPTER 2

# Genetic Programming and Coevolution

Genetic programming is an evolutionary algorithm (EA) variant developed by John Koza [3]. The primary difference between genetic programming and other EAs is the representation of an individual. While individuals of genetic algorithms or evolution strategies are typically fixed-length vectors of numbers, genetic programming individuals (in their original form) are program trees of variable structure. The program trees consist of functions and terminals. The leaf nodes are terminals and all inner nodes are functions. The children of functions supply the arguments of the function when a program tree gets evaluated. A simple example is shown in figure 2.1.

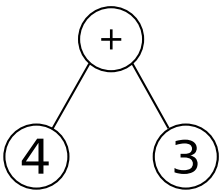



Figure 2.1: Example of an genetic programming solution. The arguments of the binary + function are supplied by the terminals 4 and 3.

A genetic programming implementation is supplied with a set of functions and terminals that it can use to solve a problem. One important constraint for the functions and terminals is the closure property: every argument of every function can be supplied by every function and terminal available without producing an error. One consequence of this is that, for example, even when the dividend that is supplied to a division function is zero the result has to be defined.

A program tree is not the only possibility of representing a program, over time other representations have been developed. In [9] a stack based program representation is introduced. An individual is a simple vector of operations. These operations are executed on a virtual stack-based machine. Every operator pops its arguments from the execution stack and pushes its result. If the stack does not contain enough arguments the operation is ignored. Flow control is hard to achieve with this type of representation. Linear genetic programming [10] uses a similar vec-

tor representation, the critical difference is that the arguments of the operations are supplied by memory cells, much like native assembler code. Before the individual is executed, the memory cells are initialized with input values. The individual manipulates the memory during execution and the output is read from one or more memory cells designated as output. This representation also has problems with flow control. The work cited uses a special operation that conditionally skips the next (and only the next) operation which eases the implementation of the crossover operator. Cartesian genetic programming [11] uses a radically different approach to map an individual to a program because it uses a genotype to phenotype transformation. The genotype (the individual) is a list of indices which specify the connections between a fixed number of logic gates and global inputs and outputs. The indices define for each gate which operation it uses and which gates (or global inputs) supply the necessary arguments for the operation. The indices also determine which gates are connected to the global output. The connected gates constitute the phenotype, i.e. the program. Other types of genetic programming include parallel distributed genetic programming [12] and grammatically-based genetic programming [13].

A variant of genetic programming that will be important for this thesis is strongly typed genetic programming [14]. It removes the closure constraint by assigning types to the arguments and return values of functions and terminals. During the construction of individuals, only functions or terminals with a compatible return type are used as arguments of a parent function. Applied to the individual of figure 2.1 on the preceding page this means that the children of the +-function have to return numbers (like the terminals 4 and 3). A terminal returning a color for instance would not be considered.

Genetic programming was successfully applied to a lot of problems, but it is not without its flaws. First and foremost, [15] cites that in most cases the function and terminal set used for solving a problem is not turing equivalent, i.e. misses loops and memory. In the work that included loop constructs, only at most two nested loops were evolved. The authors argue that evolving loops is a hard problem because small errors inside the body of a loop accumulate to large errors after multiple iterations. Building implicit loops out of lower level constructs like conditional jumps is even harder. Another focus of critique is the crossover operation as it lacks context information to select a useful part of one program and insert it at a suitable location in another program. In [16] the headless chicken crossover (no crossover at all but replacing a part of a program with randomly created code) outperforms the normal crossover operation.

Apart from these weaknesses, genetic programming typically has the problem of code bloat, i.e. programs grow in size without increasing their fitness which causes performance deterioration. In [17] and [18] six different theories of code bloat are discussed that were proposed over the years, but there is no single conclusive reason for code bloat. Those theories are:

**hitchhiking:** The hitchhiking theory states that code bloat occurs because introns (code segments without influence on the fitness of the program) that are near to advantageous code segments spread with them through the population of programs.

**defence against crossover:** According to the defense against crossover theory, code bloat emerges because large programs with a lot of intron code are more likely to survive the destructive effects of a crossover than small programs.

**removal bias:** Code removals by crossover are only allowed to be as large as an inactive code segment to not influence the fitness of the individual. However, intron code insertions by

crossover do not have any size restrictions, which causes code bloat. This argument is similar to the defense against crossover theory.

**fitness causes bloat:** The fitness causes bloat theory sees fitness as the driving factor of code bloat as experiments with random selection (without any regard for fitness) showed a complete absence of code bloat.

**modification point depth:** It was observed that the effect of a crossover on the fitness correlates with the depth of the crossover point, deeper crossover points have a smaller effect. Therefore large programs have an advantage because they can have deeper crossover points, which is the core argument of the modification point depth theory.

**crossover bias:** The crossover bias theory concentrates on the fact that repeated application of the standard subtree crossover operator creates a lot of small programs. Because small programs are generally unfit they are discarded and the average program size of the population rises, causing bloat.

Fitting for the high number of bloat theories, there are a lot of methods that aim at controlling bloat. The goal is to increase the parsimony of the found solutions or to make the evaluation of programs faster and therefore generate better solutions in the same timeframe. The bloat control originally used by Koza was a fixed limit on program tree depth. Of course, limiting the size (in total number of nodes) of a program tree is also an option. Size limits can also be applied to the whole population instead of each individual. Those limits can be static or dynamic, i.e. adapting to the current needs. There is a large number of parsimony pressure methods that produce selective pressure towards small programs. One of them is lexicographic parsimony pressure which prefers the smaller program when two programs with otherwise equal fitness are compared. Other methods punish large programs by delaying their introduction into the population or rising their probability of being discarded. Editing the programs to remove intron code is also possible to combat code growth but this can lead to premature convergence. The genetic operators are usually fixed but to mitigate code growth they can also be chosen dynamically, larger (depending on size or depth) functions are changed by operators that are more destructive. In this work, a combination of static size limits (based on the node count) and lexicographic parsimony pressure is chosen.

The second important concept necessary for this work besides genetic programming is coevolution. It refers to any situation in which the evaluation of multiple populations is dependent on each other. It is useful for competitive problems or problems for which an explicit fitness function is not known or hard to define [19]. In the domain of competitive problems, coevolution is motivated by evolutionary arms races. Two or more species constantly try to beat each other, developing higher and higher levels of complexity and performance. Coevolution can also be used to solve cooperative problems [20] by training teams of individuals. Each team member only has to solve a sub-problem. The central aspect of coevolution is the evaluation. Since no fitness measure is available, how can be determined which individuals are superior to allow any kind of progress? The answer is that the individuals of another population take the role of performance measure and to judge the fitness of one individual, it is pitted against other individuals. The intuitive solution to evaluate every individual against every other individual (complete evaluation) is usually impractical because it requires a quadratic amount of evaluations (in terms of population size) so some alternatives were developed. One of those is “All vs Best”. Each in-

dividual is evaluated by pitting it against the best individual of the previous generation. Another one is tournament evaluation [21]. The individuals of the population are paired up and evaluated. The better individual advances to the next round and is paired up with another individual that advanced from the first round. The fitness of each individual is determined by how long it stayed in the tournament.

Even though coevolution is an elegant evolutionary approach in theory, it often exhibits some rather unpleasant pathologies in practice [22, 23]:

**cycling:** especially problematic for intransitive problems like rock-paper-scissors. As soon as one population chooses mostly one answer (e.g. rock), the opposing population will converge to the appropriate answer (e.g. paper) which in turn can be exploited by the original population. Both populations will never converge as the Nash equilibrium is unstable [24, 25].

**disengagement:** happens when the evaluation does not deliver enough information to determine which individuals are better than others. In two population competitive coevolution this can happen if one population is far superior to the other one. Instead of an arms race that causes the inferior population to catch up, the evaluation labels every individual (in the inferior population) as "bad" without any gradient towards better solutions. Depending on the replacement policy, disengagement can lead to either stalling or drifting. Stalling happens when new individuals have to be better than the ones that they replace. As a result, the population will stay the same. Drift happens when individuals only have to be as good as the ones they replace.

**overspecialization:** the current population specializes to beat the current opponents without developing general problem solving capabilities.

**forgetting:** a trait lost (because at one time it does not offer an advantage) and not rediscovered when it would be beneficial again.

**relative overgeneralization:** a problem of cooperative coevolution. Individuals that are compatible to a lot of other individuals and offer moderate performance are preferred to individuals that require highly adapted partner individuals to achieve high performance.

**alteration:** instead of extending the behaviour of individuals when new opponents are encountered (elaboration), it is changed.

One approach for solving these problems is archiving. Superior individuals are archived so that newer individuals can be tested against them to ensure that the pathologies that are based on some type of trait loss (e.g. cycling, forgetting) do not occur. Archiving methods include hall of fame [26], dominance tournament [27], nash memory [28] and pareto archives [29]. These methods also help with a related problem of coevolution, the exact meaning of progress. Miconi [22] suggests that three types of progress exist in the domain of coevolution: local progress, historical progress and global progress. Local progress is the only progress that happens on its own with coevolution. When one compares the performances of a current individual and its ancestor against a current opponent, the current individual will have a higher fitness because it is adapted to its opponent. Historical progress occurs when a current individual is better than its ancestors against all opponents that were encountered. This is the situation one would expect as it describes what is suggested by the arms race argument, however, it is not a natural result

of coevolution. Archiving methods come in handy because they can be used to evaluate current individuals against the history of opponents to ensure historical progress. Global progress occurs when the current individuals are better than their predecessors against the entire search space of opponents. No method exists to ensure global progress and [22] states that "this [such a method] would involve knowledge of unknown opponents, which is absurd". This is unfortunate because global progress is the main goal of artificial coevolution, but historical progress can be used at least as indicator of global progress.

A more indirect approach to combat the pathologies of coevolution is spatial coevolution. With spatial coevolution, the individuals have assigned positions so that neighborhoods can be defined. The evaluation of an individual only regards its neighbors. The basic idea is that localized species can emerge which promotes diversity and combats the loss of traits. Its success (especially compared to complete evaluation) was demonstrated in [30].

Spatial coevolution is often combined with host-parasite coevolution, which is the most common form of competitive coevolution. It was first introduced by Hillis [19] to solve a sorting network problem. Incidentally, this is a good example for a problem where defining an explicit fitness function is infeasible because of the enormous amount of possible input permutations that would have to be tested. If the fitness function only covers a subset of permutations, the chances are high that only this subset will be sorted correctly. Host-parasite coevolution is inspired by the source of the arms race concept: the interactions of hosts and parasites in nature. Parasites will develop improved means to exploit their hosts and hosts will develop improved defences against the parasites. True to that inspiration, host-parasite coevolution uses two populations, the host and the parasite population. In [19], the host population contained sorting networks and the parasite population permutation subsets. The host population tried to evolve sorting networks that could sort the permutation subsets of the parasites and the parasites tried to evolve permutations that the sorting networks could not sort correctly. In [30] host-parasite coevolution is used to solve a regression problem. The host population contained the functions and each parasite represented one data point that had to be fitted. The hosts tried to fit the data points of the parasites while the parasites tried to use data points that the hosts could not fit. Spatial host-parasite coevolution will be used in this thesis.

CHAPTER 3

# 2-AntWars

This chapter describes the original AntWars rules as well as the changes to create 2-AntWars. Then a discussion of possible playing styles will follow to explore the strategic possibilities of 2-AntWars.

## 3.1 AntWars Rules

The rules for AntWars are defined in [1]. A short summary is given here for comparison purposes.

AntWars is a two player game that takes place on a square toroidal grid with a side-length of 11. Position $(0, 0)$ denotes the left upper corner. Both players control an ant. The ant of player 1 is located at position $(2, 5)$ and the ant of player 2 at $(8, 5)$. The aim of the game is to collect more of the 15 available pieces of food than the opponent. The food is randomly distributed on the grid, except that the starting positions never contain pieces of food and there is at most one piece of food at every position. The ants can move one field (in eight different directions) and view two fields in every direction. If an ant moves to an empty position, nothing happens. If there is a piece of food at the new position, it is eaten and the score of the ant's player is incremented. If the opposing ant is at the new position, it is neutralized and not allowed to move any more. This does not contribute to the player's score. Each ant can move 35 times. A game is won by the player with the highest score. In case of a tie the player who moved first wins. A match is won by the first player who wins three games. For the first four games, the player who is allowed to move first alternates. The player with the highest total score moves first in the final game. If there is a tie, the player with the highest score in a single game moves first. If the tie still persists, the first moving player is chosen randomly. Figure 3.1 on the next page shows the initial state of an AntWars game. The arrows indicate the movement possibilities of the ants.

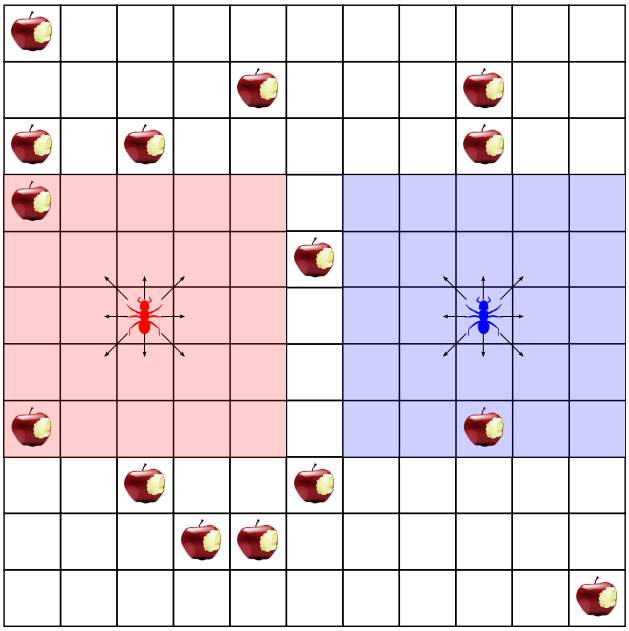


Figure 3.1: The initial state of an AntWars game.

## 3.2 2-AntWars Rules

The aim for the development of the rules of 2-AntWars was to keep them as close as possible to the original but also to make sure that the rules are flexible enough to allow for different strategies without favoring a particular strategy. The first major difference between AntWars and 2-AntWars is the playing field. The playing field of 2-AntWars is rectangular, with a width of 20 fields and a height of 13 fields. The field is no longer toroidal so that it is possible for one player to take control of a large part of the field or to hunt the ants of the other player. Hunting would not be possible on a toroidal field because the hunted ant could flee indefinitely. Each of the two players has control over two ants, which start at positions $(0, 5)$ and $(0, 7)$ for player 1 and at $(19, 5)$ and $(19, 7)$ for player 2. Every ant has the same capabilities as their AntWars brethren, i.e. they are able to move one field in every direction and view two fields in every direction. Additionally, ants can also stay at their position which might be a valid action in some situations but of course it also counts as move. Every ant can move 40 times (to compensate that the size of the playing field not just doubled). After those moves are spent, the ant is neutralized. Neutralized ants cannot move or interact with other ants in any way, but are still able to see. An ant also gets neutralized when it tries to move beyond the playing field. The field contains 32 pieces of food at random positions (excluding the starting positions of the ants and with at most one piece of food per position) to keep the food probability per position in the same range as AntWars (i.e. about 12%). There is an even number of pieces of food because games of two equal players should result in ties. To ensure some basic fairness in the random food placement each half of the playing field (10x13) contains 16 pieces of food.

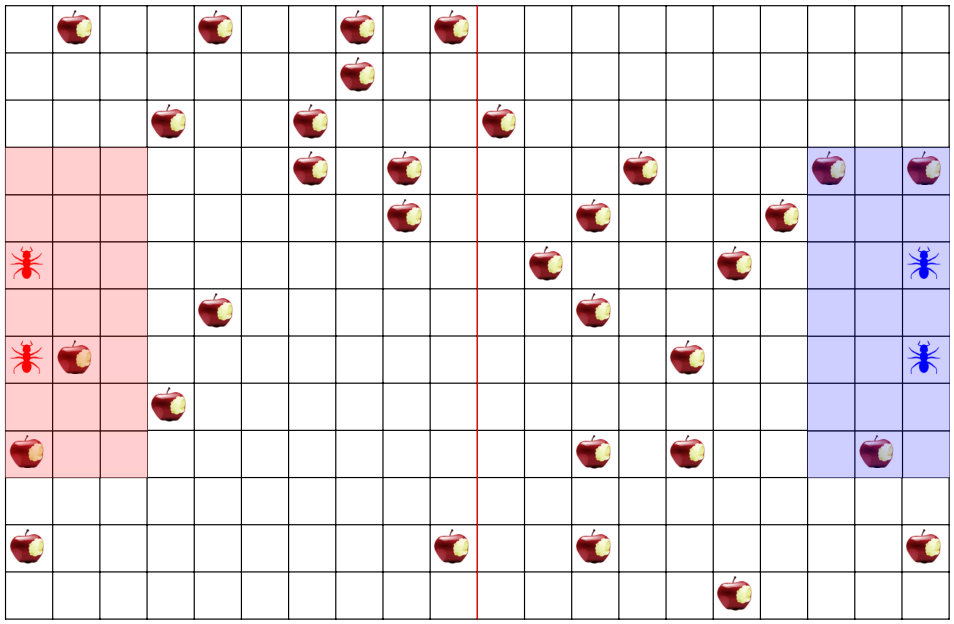


Figure 3.2: The initial state of a 2-AntWars game.

Figure 3.2 depicts the initial state of a 2-AntWars game. The red line marks the border between the two halves of the playing field (but has no direct influence on the game). It also shows the bias random food placement can introduce even with the "half the food on half the field" constraint. The food in the half of the red player (also called player 1) is clustered on the top of the field, while the food in the half of the blue player (also called player 2) is evenly distributed except the top part of the field.

The rules for battle in 2-AntWars have to be more complex than those of AntWars because now more than two ants may interact. If the ant of one player (attacker) moves to a position that already contains an ant of the other player (defender) a battle commences. Neither attacker nor defender can move away from this battle, which lasts five rounds (i.e. both players move five times) without intervention from the remaining ants. After five rounds the attacker wins the battle and the defender gets neutralized, with the same implications as above. If one of the remaining ants joins the ongoing battle (by moving to its position) then the player who has both ants in the battle wins instantly, with the losing ant being neutralized. If the attacker moves to a position occupied by both enemy ants he is immediately neutralized. After the conclusion of a battle, the winning player is free to move as before.

A game of 2-AntWars is won by collecting more food than the opponent. During a game, a player moves one of his ants before the opposing player is allowed to move. The game lasts until all food is collected, no ant is able to move (not counting ants in battle) or after 160 moves in total, whichever happens first. A match lasts five games. The player who is allowed to move first alternates during the first four games. The player who managed to collect the most food is allowed to start game five. A match is won by the player who collected the most food in total.

## 3.3 Strategies

The aim of the 2-AntWars rules was to create a game that has a varied set of possible playing strategies without preferring a specific strategy. As a consequence of this, every strategy should have a counter strategy. This section explores three such strategies (Greedy, Scorched Earth, Hunter) and their abilities to counter each other.

### Greedy

The Greedy strategy is the simplest of strategies discussed here. It mandates that ants are always moved towards the nearest food while completely ignoring the opposing ants. Matches are won by simply being very efficient at gathering the food. This strategy can be countered by Scorched Earth or Hunter. Figure 3.3 shows an example of two players using the Greedy strategy.

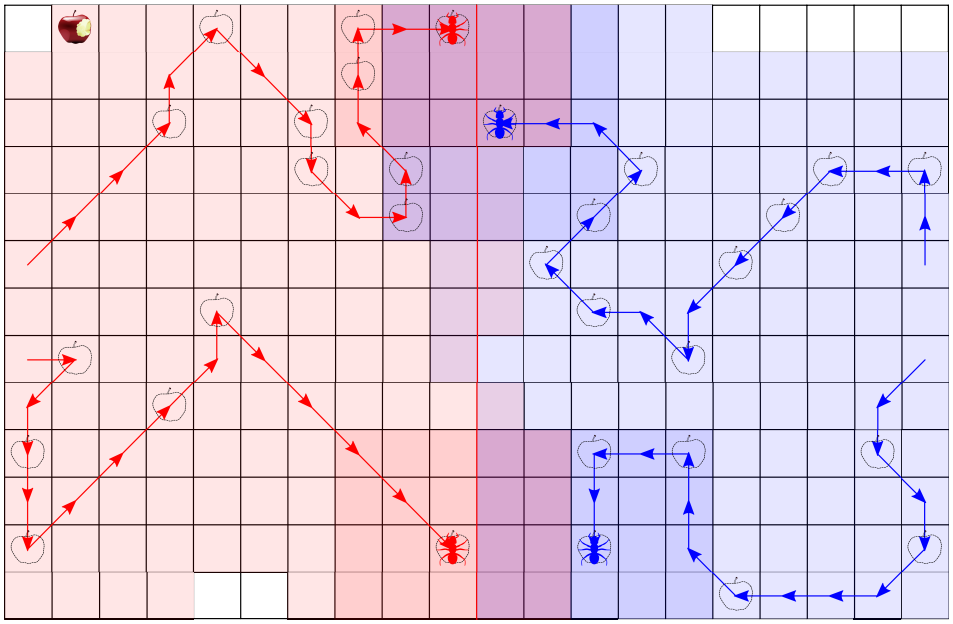


Figure 3.3: Playing field after some moves where both players use the Greedy strategy.

### Scorched Earth

The Scorched Earth strategy trades the potential of high scores that the greedy strategy provides for increased security of winning the game. To win a game, it is only necessary to collect one piece of food from the half of the playing field belonging to the opposing player, if all food in the own half of the field is collected. Therefore players playing this strategy will move the ants quickly towards the center of the playing field (possibly ignoring food on the way), collect some food from the opponent's half and then collect the food in the own half from the center of the field towards the own starting positions. Presumably, the opposing player spends his first moves

collecting the food near his starting position, so when his ants reach the center of the playing field he will discover that the food there has already been eaten. This strategy can be countered with Hunter. Figure 3.4 shows an 2-AntWars game at the critical moment when the red player (playing Greedy) finds the first eaten food of the blue player (playing Scorched Earth). When the red player explores the blue player's half of the playing field he will only find already eaten food because the blue player's ants move in front of the red player's ants towards their starting position, eating all the food.

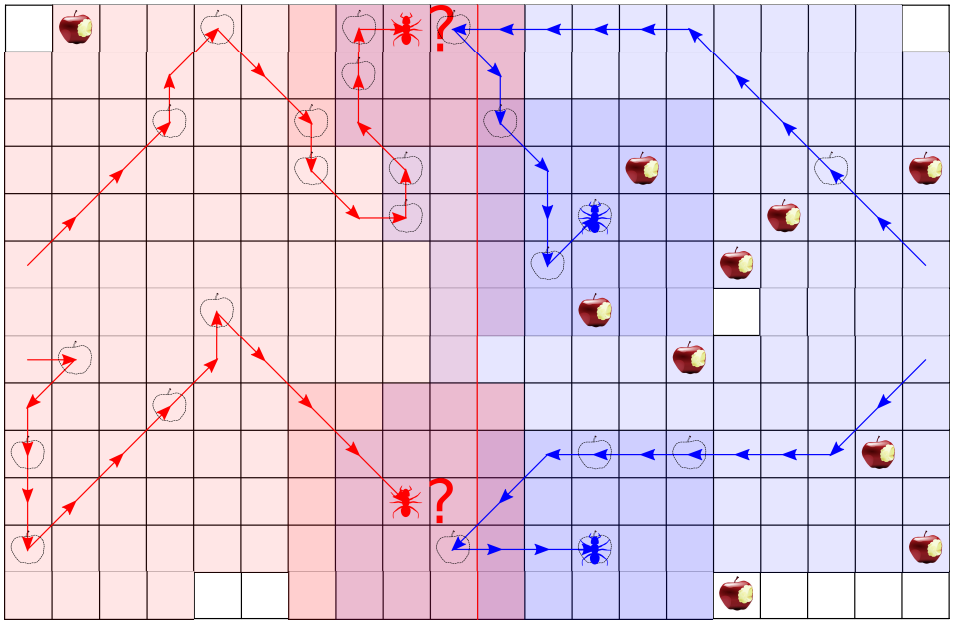


Figure 3.4: Playing field after some moves where Greedy (left) battles Scorched Earth (right).

### Hunter

The Hunter strategy is the most aggressive strategy discussed here. It relies on neutralizing one or even both ants of the opposing player fast, to gain a significant food gathering advantage. Figure 3.5 on the next page shows a game where Hunter (red) and Greedy (blue) battle each other. The red player used his ant H to hunt the prey P. To gain a speed advantage, he moved H more often than his other ant. P tried to flee but ran into the border of the playing field and was neutralized. The hunt was successful and now the red player has two ants to collect food, while the blue player has only one.

This strategy can be countered by any strategy that ensures that the own ants are close enough to support each other in battle. The rules make certain that when it comes to supporting an ant in battle the defending player has a slight advantage because he moves first after a battle started. The result of this can be seen in figure 3.6 on the following page. The ants start out as close together as possible to minimize the distance between the place of the battle and the supporting ant (a). Then the red player decides to attack (b). Now it's the blue players turn

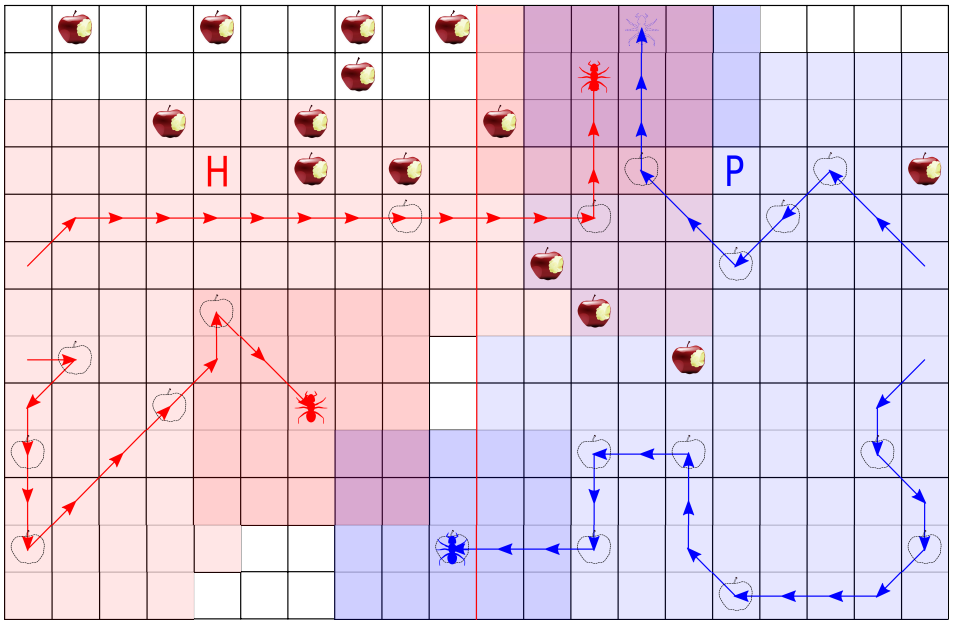
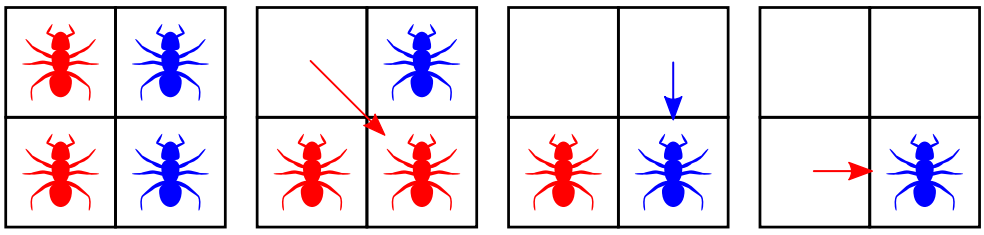



Figure 3.5: Playing field after some moves where Hunter (left) battles Greedy (right).



(a) Initial situation (b) Red attacks (c) Blue defends (d) Red loses

Figure 3.6: The aggressive red player cannot win against the defending blue player.

and he immediately supports his ant in battle (c). He wins the battle and the attacking red ant is neutralized. Then the red player makes another mistake and tries to attack again instead of fleeing with his remaining ant. He instantly loses as he battles two ants at one position (d). This shows that Hunter (and aggressive strategies in general) can always be countered.

# Part II

# Genetic Programming System

CHAPTER 4

# Genetic Programming System

This chapter describes the genetic programming system (henceforth called GPS) that was used to create 2-AntWars players. In a nutshell, it is a compiling typed tree-based (but linearly represented) evolutionary system with memory. The details will be explained in the following sections. GPS was developed for 2-AntWars but is not bound to it, it can (try to) solve any problem that implements the interface GPS expects. When a problem is mentioned in the following sections, a problem adhering to the GPS problem interface (like 2-AntWars) is meant. Also, some words will be highlighted, like *Function*, to emphasise the special meaning they have in the context of this work. Their meaning will become clear in the course of this chapter.

## 4.1 The GP-Algorithm

The GP-Algorithm illustrated in listing 4.1 on the next page is the core of GPS. First, the initial *Population* is built and evaluated, then the main loop is entered. Inside of it, a new *Population* (Pn) is built by selecting the best *Individuals* from the old *Population*. Then the crossover operator is applied with the new *Population* as receiver and a newly selected *Population* as donor (this increases the probability that good *Individuals* will be crossed with other good *Individuals*). See section 4.6 on page 22 for the semantics of donor and receiver. The new *Population* is mutated, evaluated and replaces the old *Population*. Then the cycle begins anew.

## 4.2 Individual Structure

The central data structure of GPS is the *Individual* as shown in figure 4.1 on the following page. The genetic operators of selection, crossover and mutation work on it to improve the performance of said *Individual*. *Individuals* are stored in the *Population* and at the lowest level they consist of *Statements*.

*Statements* are named and modelled after the *Statements* (and operators) of programming languages. Statements have a signature (number and type of arguments, return type) and a

```
Population gpProcedure(int maxGen){
    Population P=initPopulation();        //see 4.4 on page 22
    evaluatePopulation(P);                //see 4.8 on page 24

    for(int generation=1;generation<=maxGen;++generation){
        Population Pn=select(P);          //see 4.5 on page 22
        crossover(Pn,select(P));          //see 4.6 on page 22
        mutate(Pn);                       //see 4.7 on page 23
        evaluatePopulation(Pn);
        P=Pn;
    }
    return P;
}
```

Listing 4.1: The GP-Algorithm.

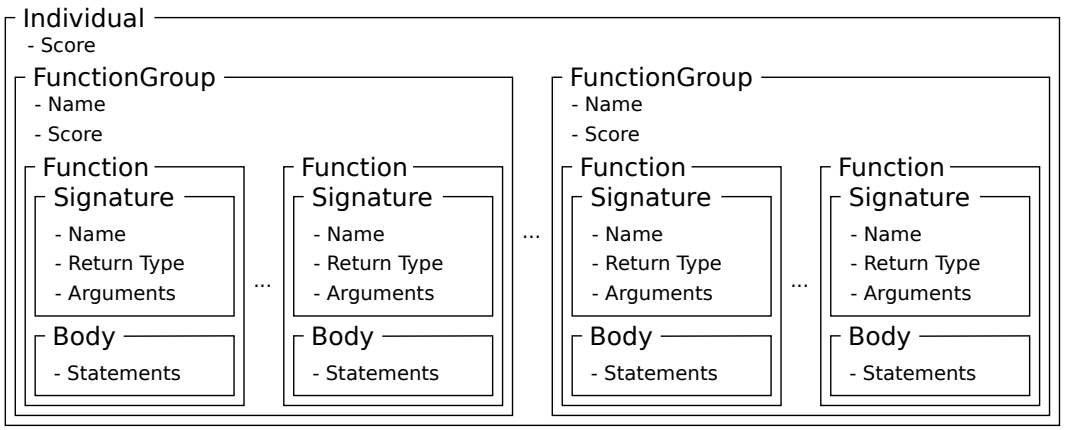



Figure 4.1: Structure of GPS *Individuals*.

name. One example of a *Statement* might be $+$: It takes two arguments of type int (the basic C++ integer data type) and returns a value of type int. More specifically it is a *FunctionStatement* because it has arguments. $5$ is another *Statement*. It returns a value of type int ($5$ according to its name) and has no arguments (which makes it a *TerminalStatement*).

A *Function* in GPS is an entity that has a signature and a body. The signature consists of the name of the *Function*, the return type and the number and types of arguments. The body gets executed when the *Function* is called. Conceptually, the body of a *Function* is represented as tree of *Statements* (similar to a parse tree). Examples can be found in Figure 4.3 on page 23. The primary difference between a *Function* and a *Statement* is that the semantic of a *Statement* is fixed, that of a *Function* can be changed by changing its *Statement* tree. Even though the concept of a *Function* is a tree of *Statements*, it is represented as array in preorder. This representation

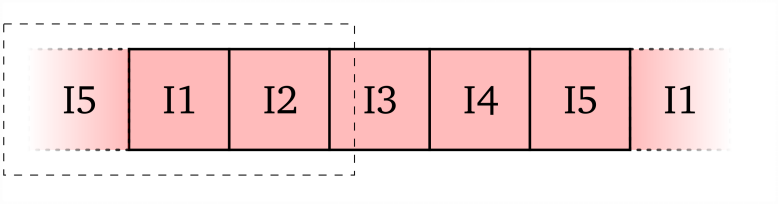
was found to deliver the best speed/size trade-off in [31]. A *Function* also manages the memory available for the *Statements*.

A *FunctionGroup* is a collection of *Functions*. It has a name and an assigned score that describes the fitness of the set of *Functions*. A *FunctionGroup* groups those *Functions* together that cannot be scored separately. GPS evolves *FunctionGroups* independently to increase their fitness.

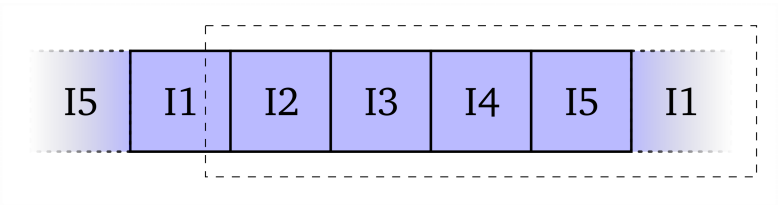
Finally, an *Individual* is a collection of *FunctionGroups*. It has an overall score that describes the fitness of the combination of *FunctionGroups*. This score is used to determine the best *Individual* inside the *Population*. GPS assumes that improving the *FunctionGroups* of an *Individual* will result in increased overall fitness.

## 4.3 Population Model

GPS supports both evolution and coevolution and the population model reflects that. The *Population* consisting of *Individuals* is split into two halves, the host half and the parasite half. The size of the *Population* is defined as the size of one half (so a *Population* in coevolutionary mode of size 10 will hold 20 *Individuals*). In evolutionary mode only the host half is used, while both halves are used in coevolutionary mode. Each *Individual* is assigned a position in its half of the population. This position is relevant for selection, crossover and evaluation. The position has only one dimension, so the *Individuals* are put next to each other forming a line. The last position on that line is adjacent to the first, so the line is actually a ring. With population size $p$ the $\Delta$-neighborhood $N$ of position $i$ is defined as $N_\Delta(i) = \{k \bmod p | i - \Delta \leq k \leq i + \Delta\}$. Figure 4.2 shows the structure of a *Population* of size five in coevolutionary mode.




(a) Host part




(b) Parasite part

Figure 4.2: The structure of a *Population* of size five (with 10 *Individuals*) in coevolutionary mode with $N_1(1)$ marked in (a) and $N_2(4)$ marked in (b).

## 4.4 Population Initialization

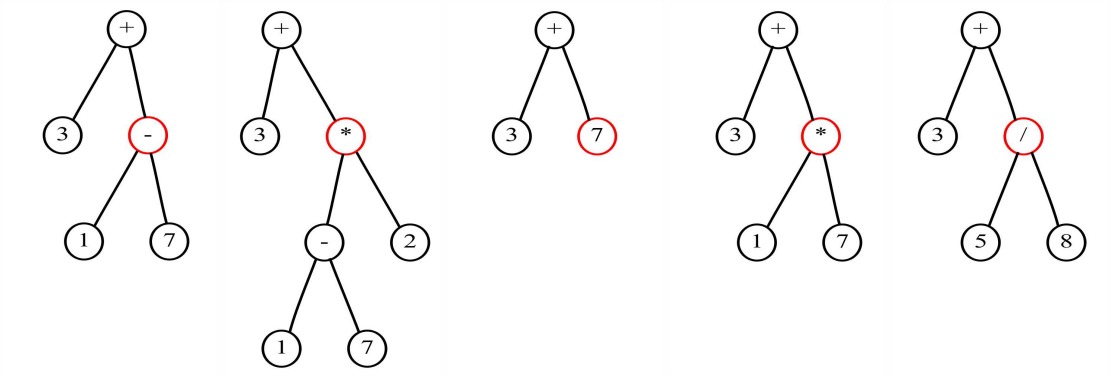
The building blocks of *Functions*, the *Statements*, are supplied by the problem. Every *Function* can have a different set of *Statements* to build its *Statement* tree out of. The *Functions* are built with the ramped half and half method. The word "ramped" refers to the depth of the trees, which is uniformly distributed between some minimum and maximum depth. The "half and half" part refers to the two building algorithms, grow and fill, which each build one half of total amount of *Functions*. The grow algorithm decides at every depth of the tree and for every argument of a *Statement* whether it is supplied by a *FunctionStatement* or *TerminalStatement*. If the target depth is reached, the growth of the tree is stopped by only using *TerminalStatements* to supply arguments. This algorithm results in sparse trees. The grow algorithm always chooses *FunctionStatements* to supply arguments unless the target depth is reached. This algorithm results in bushy trees. The built *Functions* are then assembled into *FunctionGroups* and subsequently *Individuals* which are placed in the *Population*.

## 4.5 Selection

The selection operator works on *FunctionGroup* level. That means it does not select an *Individual* based on its score but only a *FunctionGroup*. GPS tries to increase the performance of an *Individual* by increasing the performance of its *FunctionGroups*. The selection operator uses a form of localized rank selection. When a new *FunctionGroup* for position $i$ is chosen, the *FunctionGroups* at the positions $N_\Delta(i)$ (with the set selection-delta) are sorted according to their fitness. The sorted *FunctionGroups* are traversed from best to worst fitness until a *FunctionGroup* is selected. During the traversal, each *FunctionGroup* has a chance of 50% to be selected. If no *FunctionGroup* is selected during the traversal, the worst *FunctionGroup* is selected.

## 4.6 Crossover

The crossover operator works as one would expect for genetic programming. A sub-tree of *Statements* of a *Function* (donor) is copied and inserted in another *Function* (receiver). GPS ensures that the return types of the root *Statement* of the copied sub-tree and the sub-tree that is replaced in the receiver are equivalent (automatic type conversion is not supported). Since selection works on *FunctionGroup* level but crossover works on *Functions* the crossover operator has the additional task to select which *Functions* are actually crossed. To do so, the operator is given the selected donor and receiver *FunctionGroups*. Then it iterates over the *Functions* of the receiving *FunctionGroup*. Every *Function* has a probability of $p_c$ to be actually used as receiver. If a *Function* is used as one, a compatible *Function* in the donor is selected. Neither does it have to be the same *Function* nor a *Function* in the same *FunctionGroup*. The problem specifies which receiver-donor pairs are compatible. If the result of the crossover is bigger than the set limit for the *Function*, the original receiver is kept.




(a) Initial (b) Grow (c) Shrink (d) Inplace (e) Replace

Figure 4.3: The effect of the four different types of mutation available in GPS.

## 4.7 Mutation

The mutation operator works on *Statement* level. It tries to modify one *Statement* (and possibly its children) in a specific way and if that fails leaves the *Statement* unchanged. In the following the *Statement* to be modified is called the active *Statement* and the sub-tree with this *Statement* as its root is called the active sub-tree. GPS uses four different kinds of mutations: grow, shrink, inplace and replace. They are illustrated in figure 4.3. Figure 4.3a shows the initial *Statement* tree, the active *Statement* is colored red.

The grow mutation tries to replace the active *Statement* with a new *Statement* of the same return type. The new *Statement* must have at least one argument of the type of the active *Statement*'s return type, because the active sub-tree will be used as argument. If the new *Statement* needs further arguments, they are grown as detailed in section 4.4 on the preceding page. The effect of the grow mutation is shown in figure 4.3b. If a *Statement* is chosen for mutation, a grow mutation happens with probability of $m_g$.

The shrink mutation is the opposite of the grow mutation. It tries to replace the active *Statement* with one of its arguments if possible. The shrink mutation happens with probability $m_s$. Its effect is depicted in figure 4.3c.

The inplace mutation replaces the active *Statement* without changing the rest of the active sub-tree. The new *Statement* needs to have exactly the same signature as the active *Statement*. This mutation cannot fail because a *Statement* can always replace itself. The inplace mutation happens with probability $m_i$ and its effect is depicted in figure 4.3d.

The replace mutation is the most commonly used mutation in genetic programming. It replaces the active *Statement* with a freshly grown sub-tree (see section 4.4 on the preceding page). The replace mutation happens with probability $m_r$. Its effect can be seen in figure 4.3e.

All four types get their chance (with their respective probabilities) to modify the active sub-tree in the order they were explained. They are not mutually exclusive, all four types can be applied to the active *Statement* (although the replace mutation will override the effects of the

other mutations) or even none may be applied. The red markings in figure 4.3 on the previous page show which *Statements* are active after a mutation.

A part of the mutation operator works on *Function* level. It decides how many places in the *Statement* tree are mutated. The probability of a *Statement* to undergo mutation is $p_m$. A Poisson distributed random variable (depending on $p_m$ and the size of the *Statement* tree) is used to calculate how many *Statements* will be chosen randomly for mutation. If the result of the mutation is bigger that the set limit for the *Function*, the original is kept.

## 4.8 Evaluation

The job of the evaluation is to assign a score to each *Individual* in the *Population*. The first step is to transform the *Functions* of the *Individuals* into executable code. Every *Statement* has a function that allows to print it in a compilable way. During the construction of the source code file, this function is called before any of the *Statement*'s children have been printed and after each printed child. An if-*Statement* for instance will print "if(" before its arguments are printed, "){" after the first argument (the condition) and "}" after the second argument (the body). The source code is then split into parts, each containing only a fraction of the code generated from the *Population*. This has two advantages. First of all, the code can be compiled in parallel which speeds up the process immensely because every code fragment is completely independent from all other fragments. Secondly, the code can be compiled serially in smaller chunks which keeps the total amount of needed memory low. Which way (or combination) is preferable depends on compilation flags (aggressive optimization needs more memory), the *Individual* structure of the problem GPS has to solve and of course the available memory. To provide a frame of reference, for the 2-AntWars problem the typical code size was 10MB with 200000 lines of code. It was compiled in two chunks without optimization (-O0) and each compiler instance needed about 500MB of memory. After the compilation, the object files are linked to form a dynamically loadable library. This library is loaded by GPS and the function pointers for the *Functions* are extracted.

Now the actual evaluation can start. The mode of evaluation depends on whether evolution or coevolution is performed. In the case of evolution, the problem is given a set of *Functions* according to the *Individual*-structure. The problem calculates the scores (for each *FunctionGroup* and the total score) and returns them to GPS, which assigns them to the *Individual*. The coevolutionary case is a bit more complex. First of all, the problem gets two sets of functions (one from an *Individual* in the host half and one from an *Individual* in the parasite half) and returns the score. Normally, the *Individuals* that are evaluated have the same position in their respective halves of the population. GPS also allows asymmetric evaluation, where a host *Individual* is evaluated multiple times with a $\Delta$-neighborhood centered around the parasite that is used for normal evaluation. While the parasite is only assigned the score of the evaluation with the host at the same position, the host is assigned the combination of scores of all evaluations. How the scores are combined depends on the problem as it provides the particular score to use.

CHAPTER 5

# Modelling the 2-AntWars Player

The model of the 2-AntWars player is based on the successful model for AntWars presented in [32]. It consists of four *FunctionGroups*: movement, belief, predict1 and predict2.

The movement *FunctionGroup* is concerned with deciding which ant should move in which direction. To that effect, it consists of three *Functions*: decision, movement1 and movement2. The movement1 and movement2 *Functions* each calculate the movement of one ant and the decision *Function* decides which ant moves in the end. The score of the movement *FunctionGroup* is based on the food the player is able to gather during a 2-AntWars match. This is a good example of *Functions* that cannot be individually scored. It is not known which decision *Function* behaviour is advantageous and should be rewarded. Only in combination with the movement *Function* a score can be assigned.

The belief *FunctionGroup* consists of the belief *Function*. Belief in food was introduced in [32]. Ants have only a very limited view of the playing field. To support the calculation of the next move, they remember food they have seen previously (but do not see now). However, it is not certain that the food that has been seen is still there (the other player might have eaten it), hence the food belief. It is a measure of how certain a player is that a position still contains food (or that a never seen field contains food). In [32] the belief was fixed by the program. After every move it would be reduced to a fraction of its old value. It is not clear that this is the optimal method. The 2-AntWars model includes an evolvable belief *Function* to find a good way to calculate the belief, without any preconceptions. The belief *FunctionGroup* is scored by calculating the deviation between belief and reality in the following way, given position $p$ and belief $b$: If $p$ has already been seen, $1 - b$ is added to the belief deviation if $p$ contains food, $b$ is added otherwise. If $p$ has not been seen and it contains food $1 - b$ is added. Otherwise nothing is added which means believing in food at unseen positions does not contribute to the deviation. This calculation is carried out for every position after every move and the sum of all deviations gives the final score (in this case a lower score is better). As can be seen from the deviation calculation, belief is expected to be $\in [0, 1]$. This is not enforced by the model, evolution has to figure it out. What is enforced, however, is that positions that are currently seen always have

the correct food belief assigned (zero if there is no food, one if there is food), so the player can change how he believes in his memory but has to believe his eyes.

The predict1 and predict2 *FunctionGroups* each contain one *Function* with the same name. Their task is to predict the position of the enemy's ants. After every move, the distance (in moves) between the prediction and the corresponding ant is calculated. The sum of the distances during a match constitutes the score of the two predict *FunctionGroups*.

So to sum it up, a 2-AntWars player consists of six *Functions*: belief, decision, movement1, movement2, predict1 and predict2. Listing 5.1 on the facing page gives an overview on how the *Functions* are used to decide which ant to move. They (and the *Statements* that are available for them) are discussed in detail in the following sections after the basic data types have been introduced. All the scores use the size of the function as secondary criterion to decide which score is better. For instance, when two movement *FunctionGroups* find the same amount of food the smaller one is better. This introduces selective pressure towards parsimonious solutions and more so in later generations when the probability of *FunctionGroups* having the same performance rises.

## 5.1 Data Types

The data types of *Statements* (their return type and argument types) are used to decide which ones are compatible. 2-AntWars uses the following custom data types:

**AntID:** The ID of an ant. It can be zero or one and is returned by the decision function to indicate which ant has to be moved.

**Ant:** The state of an ant. It contains among other things information about the position of the ant and the amount of moves it has left.

**Direction:** A single direction, like north (N) or south-west (SW).

**Moves:** This data type stores a subset of the possible movement directions of an ant. For instance, a variable of type moves may contain the Directions NW, W and S. Set arithmetic (union, intersection etc.) is possible with variables of type Moves.

**Position:** A position on the playing field. A Position can be moved by adding a Direction, but it will always stay valid (i.e. on the playing field).

**PositionPredictionInfo:** A data structure containing information about the prediction of an enemy ant. It contains the time and position of the last sighting of the enemy ant and the current prediction of the position of the ant. The information whether the ant was seen movable is also recorded. At the begin of the game it is initialized with the starting position of the ant that is predicted.

**PlayerState:** The complete state of a player. It contains information about his ants, what they are currently seeing, what positions they have seen, how much food the player has eaten and all positions where he has seen food and what the food belief for every position on the playing field is.

```
void movePlayer(PlayerState ps,Functions f,PositionPredictionInfo& e1,
     PositionPredictionInfo& e2)
{
     //updating food belief
     for(int x=0;x<PlayingField.width;++x){
          for(int y=0;y<PlayingField.height;++y){
               const Position p(x,y);

               if(ps.PosIsVisible(p){
                    if(ps.PosHasFood(p))ps.foodBelief.at(p)=1;
                    else ps.foodBelief.at(p)=0;
               }
               else ps.foodBelief.at(p)=f.belief(...); // see 5.3 on page 33 for complete signature
          }
     }

     //updating predictions
     e1.setPrediction(f.predict1(e1,ps)); //see 5.4 on page 33
     e2.setPrediction(f.predict2(e2,ps));

     //calculating move
     Moves m1=f.movement1(...); //see 5.5 on page 34 for complete signature
     Moves m2=f.movement2(...);

     AntID antToMove=f.decision(&m1,&m2,...) //see 5.6 on page 35 for complete signature

     Direction moveDir=antToMove?m2.toRandomDirection():m1.toRandomDirection();

     //update ps to reflect the move, manage battles
     move(ps,antToMove,moveDir);
}
```

Listing 5.1: The procedure to decide on a move in 2-AntWars.

In addition to those data types, 2-AntWars uses the standard C++ data types bool (for boolean values), int, double (for double precision floating point values) and void. The data type void has a special meaning in GPS. A *Statement* with a return type of void indicates that it determines program structure and does not calculate anything on its own. For instance, the IfThenElse *Statement* has a return type of void (and two of its arguments are of type void too).

## 5.2 Available Statements

This section discusses the 91 *Statements* used by the 2-AntWars problem, sorted by their return types. Their distribution among the *Functions* is the topic of the following sections.

### Program Structure

All *Statements* listed here have a return type of void because they are used to define the program structure.

**NoOp:** *TerminalStatement* that does nothing, when printed it results in a semicolon.

**Return<T>:** *FunctionStatement* template with one argument of type T. It prints a return statement. Return<int> for instance would take an int argument (which from GPS perspective means a *Statement* tree with the root *Statement* having a return type of int) and might result in the following code: "return 3+1;".

**Program:** *FunctionStatement* that has three arguments of type void. It prints itself in the following way: "{arg1;arg2;arg3;}".

**IfThenElse:** *FunctionStatement* with three arguments, the first of type bool, the remaining two of type void. When printed, it results in "if(arg1){arg2}else{arg3}". Note that by using NoOp as arg3 an IfThen *Statement* can be built without supplying it explicitly.

### Boolean Statements

**True:** *TerminalStatement* that returns true.

**False:** *TerminalStatement* that returns false.

**Not:** *FunctionStatement* with one argument of type bool. It is used to express boolean negation and prints "!(arg1)".

**And:** *FunctionStatement* with two arguments of type bool. It is used to express boolean conjunction and prints "(arg1 && arg2)".

**Or:** like And but represents boolean disjunction and prints "(arg1 || arg2)".

**Smaller<T>, SmallerEq<T>, Eq<T>, LargerEq<T>, Larger<T>:** *FunctionStatement* templates used for comparison purposes. Each of them has two arguments of type T and prints "(arg1 OP arg2)" with OP being in order $<,<=,==,>=,>$. Most commonly int and double are used as T.

The following *Statements* (which are 2-AntWars specific) are also available:

**SeenMovable:** *FunctionStatement* with one argument of type PositionPredictionInfo. From its argument it extracts whether the enemy ant was seen movable when it was last seen.

**PositionWasSeen:** *FunctionStatement* with one argument of type PlayerState and one of type Position. It uses the PlayerState to determine whether the Position was seen or not.

**IsNorth, IsSouth, IsEast, IsWest, IsNE, IsNW, IsSE, IsSW:** *FunctionStatements* with two arguments of type Position. Returns whether the direction from the first Position to the second Position has a north, south, east, west, north-east, north-west, south-east or south-west component respectively.

**AntIsMovable:** *FunctionStatement* that returns true if the Ant supplied as argument is movable (i.e. has moves left and is not in battle or neutralized).

**AntIsPassive:** *FunctionStatement* with one argument of type Ant that returns true if the Ant cannot move or interact with anything (i.e. is neutralized).

**AntInBattle:** *FunctionStatement* that returns true if the supplied Ant argument is currently engaged in battle.

### Integer Statements

**EpInt(min,max,delta):** ephemeral constant with a value $\in [\min, \max]$. This *Statement* uses a custom mutation operator, i.e. it does not use the methods outlined in section 4.7 on page 23. Instead, it adds a uniformly distributed value $\in [-\text{delta}, \text{delta}]$ to the current value (while respecting min and max).

**AddI, SubI, ModI:** *FunctionStatements* that facilitate addition, subtraction and modulo division. They each have two arguments of type int and print "(arg1 OP arg2)" with OP being $+, -$ and $\%$. Note that modulo division is protected and returns the value of arg1 if arg2 equals zero.

In addition to these general purpose *Statements*, the following *Statements* specific to 2-AntWars are available:

**Width, Height:** *TerminalStatements* that return the width (20) and height (13) of the playing field respectively.

**TotalFood:** *TerminalStatement* that returns the total amount of food available on the playing field (32).

**MovesPerAnt:** *TerminalStatement* that returns the total amount of moves that an ant is allowed to make (40).

**BattleRounds:** *TerminalStatement* that returns the number of battle rounds before the battle is finished (5).

**PosGetX, PosGetY:** *FunctionStatements* with one argument of type Position. They extract the X and Y coordinates of their argument.

**DistanceMoves:** *FunctionStatement* with two arguments of type Position that returns the distance in moves between those Positions.

**ElapsedTime:** *FunctionStatement* that extracts the elapsed time (which equals the number of moves made by a player) from its argument of type PlayerState.

**FoundFood:** *FunctionStatement* with one argument of type PlayerState. It returns the amount of food the player has already found.

**SightingTime:** *FunctionStatement* with one argument of typePlayerState and one argument of type Position. It uses the PlayerState to return the last time the Position was seen. If the Position was never seen, it returns zero.

**AntExtractX, AntExtractY:** *FunctionStatement* with one argument of type Ant. It extracts the X (or Y) component of the Ant's position.

**AntMovesLeft:** *FunctionStatement* with one argument of type Ant. It extracts the number of moves the Ant has left.

**TimeOfLastSighting:** *FunctionStatement* that extracts the time of last sighting from its argument of type PositionPredictionInfo.

### Double Statements

**AddD, SubD, MulD, DivD:** *FunctionStatements* that facilitate addition, subtraction, multiplication and division. They each have two arguments of type double and print "(arg1 OP arg2)" with OP being in order $+$,$-$,$*$ and $/$. Note that DivD is not protected and division by zero will be executed. This results in the value mandated by IEEE 754 floating point arithmetic rules (i.e. NaN or $\pm$ INF depending on the divisor).

**Sin, Cos:** *FunctionStatements* for trigonometric functions, each taking one argument of type double.

**Pow:** *FunctionStatement* with two arguments of type double. Returns the result of $\text{arg1}^{\text{arg2}}$.

**Log:** *FunctionStatement* with one argument of type double. Returns the natural logarithm of arg1.

**EpDouble($\mu$,$\sigma$):** an ephemeral constant with value $\mu$. This *Statement* uses a custom mutation operator, i.e. it does not use the methods outlined in section 4.7 on page 23. Instead, it uses an $N(0, \sigma)$ distributed random variable to offset its $\mu$.

**EpDoubleRange($\mu$,$\sigma$,min,max):** an ephemeral constant like EpDouble but with a value guaranteed to be $\in [\text{min}, \text{max}]$.

The following *Statements* are specific to the 2-AntWars context:

**PositionDistance:** *FunctionStatement* with two arguments of type Position. It returns the euclidean distance between them.

**FoodBeliefAtPos:** *FunctionStatement* with one argument of type PlayerState and one of Position. It returns the food belief at the passed Position on the playing field.

### Position Statements

**PosPlusDirection:** *FunctionStatement* with one argument of type Position and one argument of type Direction. It returns a Position moved in Direction. If the Position would leave the playing field it is not changed.

**PosPlusCoordinates:** *FunctionStatement* with one argument of type Position and two arguments of type int. It returns a Position with the int arguments added to the x- and y-coordinates of the Position argument. The coordinates are clamped to the borders of the playing field so the resulting Position is always valid.

**CurrentPrediction:** *FunctionStatement* with one argument of type PositionPredictionInfo. It extracts the current predicted position.

**LastSeenPosition:** Like CurrentPrediction, but extracts the last seen position from its argument.

**AntPosition:** *FunctionStatement* that extracts the position of its one argument of type Ant.

**EpPosition($\delta$):** an ephemeral constant of type Position. It is initialized with a random Position on the playing field. The custom mutation operator offsets the coordinates of the Position with a uniformly distributed random value in the range of $\pm\delta$ (while ensuring valid coordinate values).

**NearestFood:** *FunctionStatement* with four arguments, one of type PlayerState, one Position p, one double b and one integer n. This *Statement* sorts the Positions with a food belief not smaller than b by their distance in moves to p. If there is no such Position, p is returned. Otherwise the *Statement* returns the n-th nearest Position, with counting starting at zero. If n is smaller then zero, zero is assumed. If there is no n-th nearest Position, the Position with the largest distance is returned.

**NearestUnseenField:** *FunctionStatement* with three arguments, one of type PlayerState, one Position p and one integer n. It works like NearestFood, but Positions qualify if they have not been seen, not because of their assigned food belief.

### Direction Statement

2-AntWars has only one *Statement* that returns a type of Direction (disregarding constants). The Statement is called ToRandomDirection. It is a *FunctionStatement* that returns a random Direction from its argument of type Moves.

### Moves Statements

**MNone:** *TerminalStatement* returning a Moves value without any movement directions set.

**MAll:** *TerminalStatement* returning a Moves value with all movement directions set.

**MToward, MNeutral, MAway:** *FunctionStatements* with two arguments of type Position and calculating Moves that will decrease, not change or increase the distance (in number of moves) between its two arguments when applied to the first argument.

**MIntersection, MUnion, MDifference:** *FunctionStatements* with two arguments of type Moves and returning the result of the corresponding set operation.

**MNot:** *FunctionStatement* that inverts its argument of type Move. The return value will contain all Directions the argument does not.

**MAddDirection, MSubDirection:** *FunctionStatements* with one argument of type Moves and one of type Direction evaluating to a Moves value with the Direction argument added to (or removed from) the Moves argument.

**FoodEatMoves:** *FunctionStatement* with one argument of type PlayerState and one argument of type Position. It checks for which of the (at most) eight neighboring Positions of the Position argument the food belief is above zero and returns a Moves value containing the directions from the Position argument to those Positions.

**FoodEatMovesMaxBelief:** *FunctionStatement* like FoodEatMoves, but instead returns all Directions with the maximum belief.

**FoodEatMovesAboveBelief:** *FunctionStatement* like FoodEatMoves, but with one additional argument of type double. It will only add directions to positions to the return value if the position's food belief is not smaller than the double argument.

**MaxNewFieldsMoves:** *FunctionStatement* with one argument of type Position and one argument of type PlayerState. It returns all the movement directions from its Position argument that achieve the maximum of newly seen fields.

**NearestFoodMoves:** *FunctionStatement* with four arguments, one of type PlayerState, a Position p, a double b and an integer n. This *Statement* takes all Positions on the playing field

with a food belief not smaller than b and sorts them by distance (in number of moves) to p. If no Positions qualify, then an empty Moves value is returned. From all qualifying Positions, only those on the n-th distance level are regarded (counting starts at zero). If n is smaller than zero, the zero-th level is used. If n is larger than the total number of distance levels, the level with the highest distance is used. The *Statement* then returns a Moves value with all Directions from p to the Positions on the target distance level. This is the most powerful *Statement* available to 2-AntWars as it transforms the seen food (more precisely the food belief) into useful Moves without the need to evolve loops or data structures.

**NearestUnseenFieldMoves:** *FunctionStatement* with three arguments, one of type PlayerState, a Position p and an integer n. This *Statement* works like NearestFoodMoves, but Positions qualify if they have not been seen yet and not because of food belief.

### Special Statements

The *Statements* presented here either have return types depending on template arguments, are used to work around some issues with the type system or facilitate special functions of GPS (like function arguments and memory access).

**Constant<T>(V):** *TerminalStatement* that evaluates to a constant of type T with value V.

**DefaultReturnBlock<T>:** *FunctionStatement* that is used as root for every *Function* (instantiated with the return type of the *Function*). It has the return type T and two arguments, one of type void and one of type T. It prints "arg1;return arg2;". This makes sure that every generated program contains a valid return statement.

**Convert<S,T>:** This *FunctionStatement* template has a return type of T and one argument of type S. It is used to facilitate type conversions (which are not supported by GPS). It prints as "T(arg1)". For 2-AntWars it is most often utilized to use int values (like the width of the playing field) in floating point calculations.

**Argument(T,name):** This *TerminalStatement* has a return type of T and is used to make the arguments of a *Function* available to its *Statement* tree. It is automatically generated after the problem supplies the description of the *Functions* to GPS (argument types and their names, return type and *Function* name) and not by the problem. Note that the return type is specified at run-time because of this. When printed, this *Statement* simply outputs its name.

**LoadVar(T,name):** This *TerminalStatement* has a return type of T. It is used to access the value of a variable and prints its name. It is automatically generated by GPS based on the variable description of the problem (number and types of variables).

**StoreVar(T,n):** This *FunctionStatement* has a return type of void and one argument of type T (arg1). It is used to store the value of its argument in the variable of type T with name n and writes "n=arg1;".

**Assignment<L,R>:** This *FunctionStatement* template has a return type of void and is besides return<T> and StoreVar(T,n) the third *Statement* to facilitate the transition from program structure to actual program code. The *Statement* has two arguments, one of type L (arg1) and one of type R (arg2). L has to be a pointer type. When printed, this *Statement* results

```
double belief(Position p, bool haveSeenField, bool haveSeenFood, int timeSinceSighting, int
    elapsedTime, int foundFood, int seenFields, PositionPredictionInfo enemy1,
    PositionPredictionInfo enemy2);
```

Listing 5.2: The signature of the belief function.

```
Position predict(PositionPredictionInfo ppi, PlayerState ps);
```

Listing 5.3: The signature of the predict functions.

in "(*arg1)=arg2;". It has to be ensured that this assignment makes sense from the point of view of the C++ type system. For an usage example, see section 5.6 on page 35.

## 5.3 Belief Function

This section describes the signature of the belief *Function* and lists the available *Statements* for it. The signature is shown in listing 5.2 and the meaning of its arguments is described below:

**p:** Position for which the belief is being calculated
**haveSeenField:** true if the position has been seen
**haveSeenFood:** true if the position contained food when it was last seen
**timeSinceSighting:** number of moves since the position was seen
**elapsedTime:** number of moves since the start of the game
**foundFood:** number of food the player has already eaten
**seenFields:** number of fields that have been seen at least once
**enemy1, enemy2:** prediction information for the ants of the enemy

The following *Statements* are available for the belief *Function*:

**void:** NoOp, IfThenElse, Program, Return<double>
**bool:** True, Not, And, Or, SmallerEq<double>, LargerEq<double>, SeenMovable
**int:** Width, Height, TotalFood, PosGetX, PosGetY, TimeOfLastSighting, DistanceMoves
**double:** EpDouble(0.5,0.15), Sin, Cos, Log, Pow, AddD, SubD, MulD, DivD
**Position:** CurrentPrediction, LastSeenPosition
**Special:** DefaultReturnBlock<double>, Convert<int,double>, Constant<double>(0), Constant<double>(1)

The belief *Function* has no variables and is only to itself crossover compatible.

## 5.4 Predict Functions

This section describes the signature of the predict *Functions* and lists the available *Statements* for it. The signature is shown in listing 5.3 and its arguments are described below:

```
Moves movement(Ant myAnt, PlayerState myState, Ant otherAnt);
```

Listing 5.4: The signature of the movement functions.

**ppi:** current PositionPredictionInfo for the enemy ant that has to be predicted
**ps:** state of the player

The following *Statements* are available for the predict *Functions*:

**void:** NoOp, IfThenElse, Program, Return<Position>
**bool:** True, Not, And, Or, Smaller<int>, Eq<int>, SmallerEq<double>, LargerEq<double>, Eq<Position>, IsNorth, IsSouth, IsEast, IsWest, IsNE, IsNW, IsSE, IsSW, PositionWasSeen, SeenMovable
**int:** EpInt(-30,30,3), Width, Height, ElapsedTime, AddI, SubI, ModI, DistanceMoves, PosGetX, PosGetY, TimeOfLastSighting, SightingTime
**double:** EpDouble(0,0.15), AddD, SubD, MulD, DivD, Sin
**Position:** EpPosition(1), CurrentPrediction, LastSeenPosition, PosPlusDirection, PosPlusCoordinates
**Direction:** Constant<Direction>(ZERO), Constant<Direction>(NORTH), Constant<Direction>(SOUTH), Constant<Direction>(EAST), Constant<Direction>(WEST), Constant<Direction>(NE), Constant<Direction>(NW), Constant<Direction>(SE), Constant<Direction>(SW), ToRandomDirection
**Moves:** MAll, MNone, MToward, MNeutral MAway, MUnion, MIntersection, MDifference, MAddDirection, MSubDirection
**Special:** DefaultReturnBlock<Position>, Convert<int,double>

The predict *Functions* have no variables and are crossover compatible to each other.

## 5.5 Movement Functions

This section describes the signature of the movement *Functions* and lists the available *Statements* for it. The signature is shown in listing 5.4 and its arguments are described below:

**myAnt:** information about the ant of which the Moves have to be calculated
**myState:** state of the player
**otherAnt:** information about the other Ant

The following *Statements* are available for the movement *Functions*:

**void:** Program, IfThenElse, NoOp, Return<Moves>
**bool:** True, Not, And, Or, Smaller<int>, SmallerEq<int>, Eq<int>, LargerEq<int>, Larger<int>, Smaller<double>, SmallerEq<double>, Larger<double>, LargerEq<double>, Eq<Position>, AntIsMovable, AntIsPassive, AntInBattle, IsNorth, IsSouth, IsEast, IsWest, IsNE, IsNW, IsSE, IsSW

```
AntID decision(Moves* m1, Moves* m2, Ant a1, Ant a2, PositionPredictionInfo enemy1,
    PositionPredictionInfo enemy2, PlayerState ps, int battleEndTime);
```

Listing 5.5: The signature of the decision function.

**int:** EpInt(0,20,2), AddI, SubI, Width, Height, MovesPerAnt, TotalFood, BattleRounds, AntExtractX, AntExtractY, AntMovesLeft, PosGetX, PosGetY, ElapsedTime, FoundFood, DistanceMoves

**double:** EpDouble(0.5,0.15), FoodBeliefAtPos, PositionDistance

**Position:** EpPosition(1), AntPosition, NearestFood, NearestUnseenField

**Direction:** Constant<Direction>(NORTH), Constant<Direction>(SOUTH), Constant<Direction>(EAST), Constant<Direction>(WEST), Constant<Direction>(NE), Constant<Direction>(NW), Constant<Direction>(SE), Constant<Direction>(SW)

**Moves:** MAll, MNone, MToward, MNeutral, MAway, MUnion, MIntersection, MDifference, MNot, Constant<Moves>(NORTH), Constant<Moves>(SOUTH), Constant<Moves>(EAST), Constant<Moves>(WEST), Constant<Moves>(NE), Constant<Moves>(NW), Constant<Moves>(SE), Constant<Moves>(SW), FoodEatMoves, FoodEatMovesMaxBelief, FoodEatMovesAboveBelief, MaxNewFieldsMoves, NearestFoodMoves, NearestUnseenFieldMoves, MAddDirection, MSubDirection, PosPlusDirection

**Special:** DefaultReturnBlock<Moves>

The movement functions have access to two variables of type Position, Direction and Moves and are crossover compatible to each other.

## 5.6 Decision Function

This section describes the signature of the decision *Function* and lists the available *Statements* for it. The signature is shown in listing 5.5 and its arguments are described below:

**m1, m2:** Moves suggested by the movement functions. They are pointers because the decision function can change them.

**a1, a2:** Ants of the player.

**enemy1, enemy2:** PositionPredictionInfo about the enemy ants.

**ps:** State of the player.

**battleEndTime:** End time of a battle if such a battle exists. If none is being fought, the current time is passed as argument. If two battles are being fought simultaneously, the argument is the end time of one of those battles. Which one is unspecified because there are no movable ants in such a case.

The following *Statements* are available for the decision *Function*:

**void:** Program, IfThenElse, NoOp, Return<AntID>
**bool:** True, Not, And, Or, Smaller<int>, SmallerEq<int>, Eq<int>, LargerEq<int>, Larger<int>, Smaller<double>, Larger<double>, AntIsMovable, AntIsPassive, AntInBattle
**int:** EpInt(0,20,2), EpInt(0,160,2), AddI, SubI, ModI, AntExtractX, AntExtractY, AntMovesLeft, Width, Height, BattleRounds, TotalFood, ElapsedTime, FoundFood, DistanceMoves
**double:** EpDoubleRange(12,5,0,24), PositionDistance
**Position:** EpPosition(1), AntPosition
**Direction:** Constant<Direction>(NORTH), Constant<Direction>(SOUTH), Constant<Direction>(EAST), Constant<Direction>(WEST), Constant<Direction>(NE), Constant<Direction>(NW), Constant<Direction>(SE), Constant<Direction>(SW)
**Moves:** Constant<Moves>(NORTH), Constant<Moves>(SOUTH), Constant<Moves>(EAST), Constant<Moves>(WEST), Constant<Moves>(NE), Constant<Moves>(NW), Constant<Moves>(SE), Constant<Moves>(SW), MAll, MNone, MAway, MNeutral, MToward, MIntersection, MUnion, MDifference, MNot
**Special:** DefaultReturnBlock<AntID>, Convert<Moves&,Moves>, Assignment<Moves*, Moves>, Constant<AntID>(0), Constant<AntID>(1)

The decision *Function* has variables of type Moves and Direction (two of each) and is only to itself crossover compatible.

## 5.7 Settings

Table 5.1 on the facing page lists the standard settings that were used to create the results reported in part III on page 41. Deviating settings will be mentioned in the appropriate sections.

After some preliminary runs of GPS the population size and the number of iterations has been chosen to keep the run-time manageable but ensure enough opportunity for the population to generate optimized playing strategies. The number of individuals per source code file has been chosen to allow for parallel compilation with two threads, appropriate for the two threads that GPS uses to evaluate the *Population*. The -O0 optimization level has been chosen because the increased execution speed did not offset the increased compilation time with higher optimization levels. The selection deltas for the host and parasite parts of the population in combination with the population size means that one improved *Individual* can take over the population in 166 generations. The maximum sizes of the *Functions* of 2-AntWars were set to reflect that belief and predict *Functions* have a auxiliary role while the most work is done in the movement *Functions*, with the decision *Function* in the middle. As a result of the allowed sizes, the total *Individual* size cannot exceed 1100 *Statements*. The crossover and mutation probabilities were found to deliver a good trade-off between evolutionary speed and the probability of drastically reducing an *Individual*'s fitness by changing important parts of its *Statement* tree. The probabilities of the different mutation types were chosen so that only a quarter of *Statement* mutations will not

result in any change. It is important to note that the minimum depth of the *Statement* trees is the real minimum, *Statement* trees cannot be smaller. The root *Statement* is always a DefaultReturnBlock, its argument of type void can be filled by a Return *Statement* which in turn needs at least a *TerminalStatement* as argument, so the total depth of the tree is two.

Table 5.1: Default GPS settings for the 2-AntWars problem.

| Setting Name | Description | Value |
|---|---|---|
| Population size | Size of the host population | 1000 |
| Iterations | MaxGen from listing 4.1 on page 20 | 1000 |
| Individuals per file | Compiled with one gcc instance | 500 |
| Compile flags | For the population | -O0 -DNDEBUG |
| $\Delta_{\text{host}}$ | Delta for host selection | 3 |
| $\Delta_{\text{parasite}}$ | Delta for parasite selection | 3 |
| $\Delta_{\text{eval}}$ | Delta for asymmetric evaluation | 0 |
| Coevolution | No coevolution means evolutionary mode of evaluation | 0 |
| GamesPerMatchBest | Number of games per match when the best host battles the best parasite | 50 |
| MaxBeliefSize | Size limit for belief *Function* | 100 |
| MaxPredictSize | Size limit for predict *Functions* | 100 |
| MaxMovementSize | Size limit for movement *Functions* | 300 |
| MaxDecisionSize | Size limit for decision *Function* | 200 |
| $p_c$ | Crossover probability (per *Function*) | 0.6 |
| $p_m$ | Mutation probability (per *Statement*) | 0.001 |
| $m_g$ | Grow mutation probability | 0.3 |
| $m_s$ | Shrink mutation probability | 0.3 |
| $m_i$ | Inplace mutation probability | 0.3 |
| $m_r$ | Replace mutation probability | 0.3 |
| TreeDepthMin | Minimum depth of *Statement* trees during growth | 2 |
| TreeDepthMax | Maximum depth of *Statement* trees during growth | 6 |

# Part III

# Results

CHAPTER 6

# No Adversary

This chapter presents the results of letting one 2-AntWars player evolve while fixing the other to do nothing. This will show that the 2-AntWars player model is suited to play 2-AntWars and that GPS is able to evolve successful strategies. The data outlined here will also serve as base-line for comparison in the following chapters. The results presented from here on out are based on one GPS run with the settings from table 5.1 on page 37. Computing results based on multiple runs was not possible because one run took approximately one day to complete and the evolutionary dynamic that emerges can only be discussed in detail based on a single run anyway. One experiment that involved multiple runs can be found in section 10.2 on page 107.

## 6.1 Fitness development

Figure 6.1 on the next page shows the fitness development of the run with no adversary. It can be seen that GPS learns to play 2-AntWars relatively fast. The best player finds 134 pieces of food per match in generation six. There is continuous improvement until generation 55 where the amount of found food per game jumps up to 155 per match. To find the reason for this sudden increase in performance, the games played by the best *Individuals* of generations 25 and 100 were studied. It was seen that both *Individuals* first move ant 2 until it runs out of moves and then ant 1. However, the *Individual* of generation 25 sometimes moves to an empty position even though a position containing food is also reachable with one move. Furthermore, the food seen by ant 2 seems to have a bigger influence on the movement direction of ant 1 than the food ant 1 sees itself. In one game where ant 2 ran out of moves in the upper right corner of the playing field, ant 1 immediately moved to that position to eat the food ant 2 was seeing while ignoring food on the way. Such wasteful behaviour was not observed with the ants of the *Individual* of generation 100. At generation 90 the first *Individual* achieves the maximum of 160 pieces of food. Starting with generation 150, the best *Individual* never finds less than 158 pieces. These results might suggest that GPS is able to evolve perfect 2-AntWars players. However, this is not the case. As seen in figure 3.2 on page 13, the random placement of food can introduce a significant bias. In this case, a positive bias (i.e. suitable food placement) was

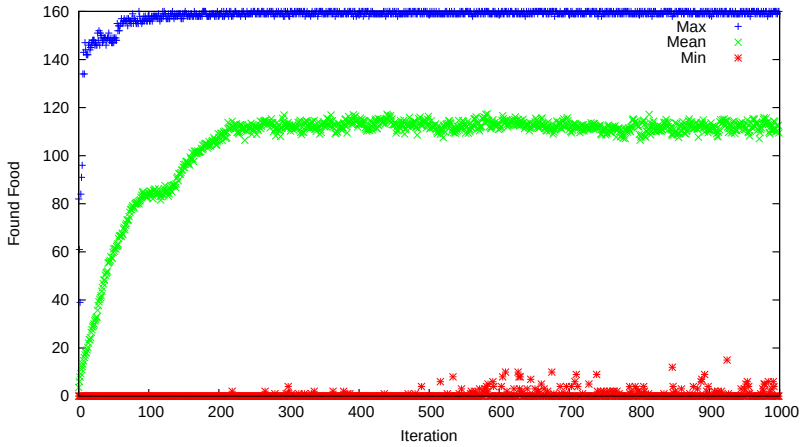



Figure 6.1: Fitness development of the 2-AntWars population without opponent.

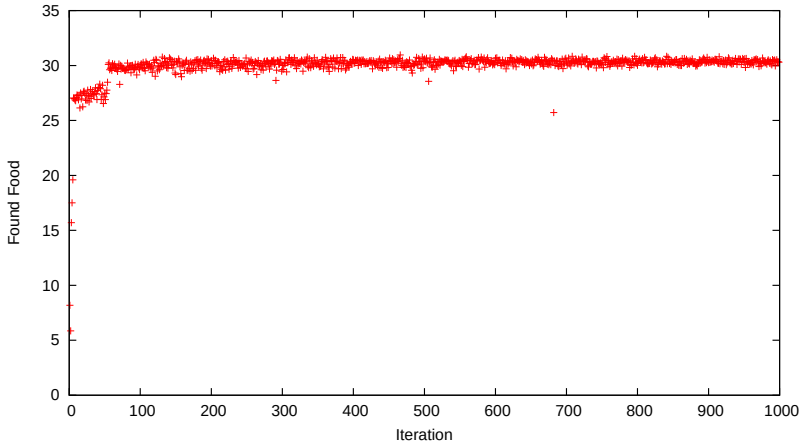



Figure 6.2: Average number of found pieces of food of the best *Individual* per game.

needed to achieve the perfect score. The study of the games of the best *Individuals* of generation 25 and 100 showed that the ants simply ran out of movements at the end of the game and could not continue to collect. The result can be seen in figure 6.2 which depicts the average found food of the best *Individual* when it has to play a match with 50 games. The figure shows that the best *Individuals* only find about 30 pieces of food on average per game instead of the maximum of 32. In the last 300 generations this average changes by at most 1.06 pieces of food.

Figure 6.3 on the facing page gives another view on the fitness progression of this run. Its z-values (number of *Individuals* collecting a specific amount of food) are clipped at 100. The maximum is 400, but this is reached in the lower left corner, which means it is the number of *Individuals* in the first generation not finding any food. As can be seen, the fitness distribution splits into four distinctive clusters at 150, 90, 60 and 30 pieces of food after 250 generations.

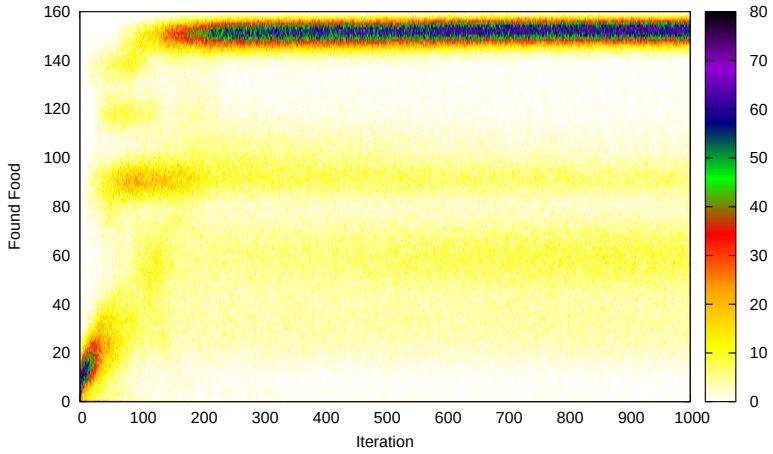
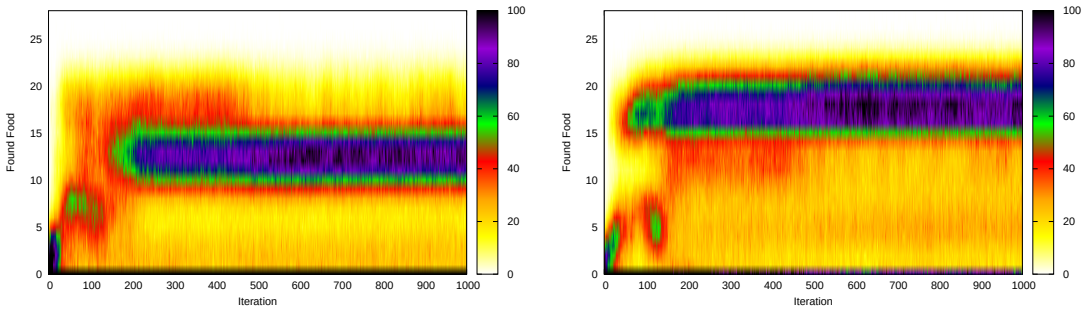



Figure 6.3: Number of *Individuals* finding a particular number of pieces of food.




(a) Ant 1

(b) Ant 2

Figure 6.4: Distribution of found pieces of food per ant and game.

The highest level is what was expected from figure 6.2 on the preceding page, since 150 pieces of food in a standard match equals to 30 pieces of food per game. The other levels can be explained by examining figure 6.4. Both ants collect food, ant 1 12 pieces per game (which is the source of the cluster at 60 pieces of food per match) and ant 2 18 pieces (which is the source of the cluster at 90 pieces of food per match). That means that the second and third cluster are the result of a destructive crossover or mutation that damaged one movement *Function* or the decision *Function* so that only one ant collects food. The fourth cluster at 30 pieces of food seems to be mostly a result of a destructive change in ant 2 but this is not as clear as it is with the other clusters. It is not surprising that ant 2 collects more food than ant 1 because it moves first.

These subtleties uncovered with figures 6.3 and 6.4 are completely missed by figures like figure 6.1 on the preceding page so they will not be shown for future runs. Instead, figures like figure 6.5 on the following page will be used to give an overview of the fitness development of the population.

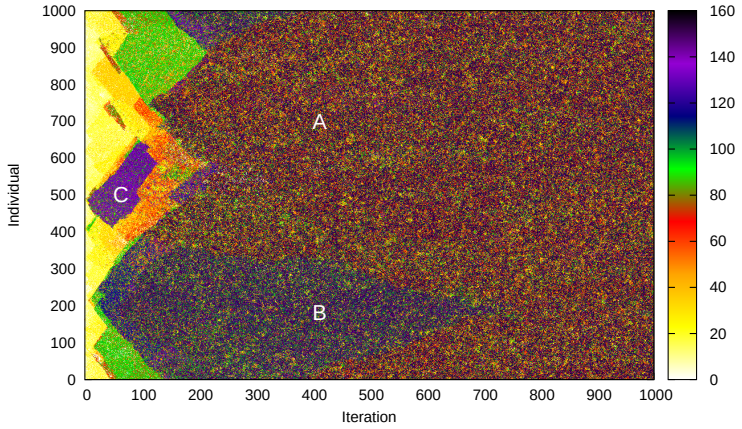
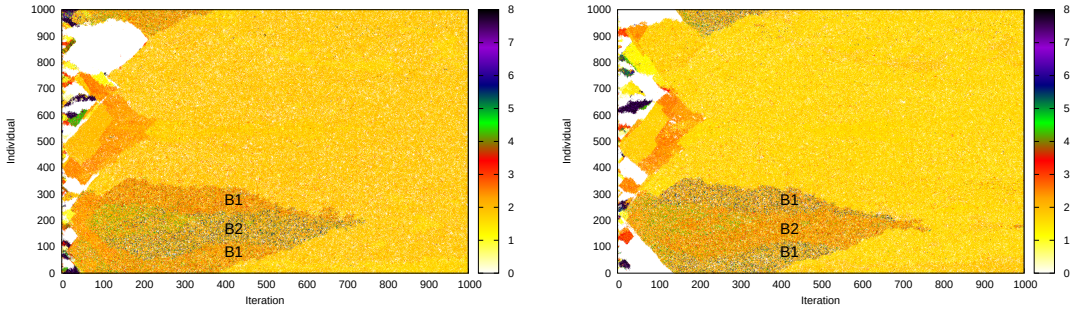



Figure 6.5: Fitness development of the 2-AntWars population without opponent.

A lot of change in the population can be seen in the early generations when improved *Individuals* spread. As hinted by the previous graphs, the situation stabilizes at generation 200. But instead of converging, the population still contains two species, labeled A and B. From here on out a set of *Individuals* sharing a characteristic will be called species or a subspecies if it is embedded in a species. Species A slowly replaces species B until B vanishes around generation 700. This shows that, due to the stochastic nature of 2-AntWars, replacement can happen at a much slower rate than with the maximum determined by the $\Delta_{\text{host}}$ and $\Delta_{\text{parasite}}$ values but the process is not stopped completely. The standard settings set them to three which means that in a population containing two species one can replace the other with a speed of six *Individuals* per generation. In the situation depicted in figure 6.5, species B has as size of 450 *Individuals* at generation 225 and becomes extinct at generation 700 which means it only loses 0.95 *Individuals* per generation. Species C, that becomes extinct even though it is superior to its neighboring species, will be discussed later.




(a) Ant 1

(b) Ant 2

Figure 6.6: Average number of move directions to chose randomly from after the decision *Function* determined the ant that will be moved.

When analyzing other aspects of the population development it becomes clear that B is not one species, but two that are not distinguishable by their fitness. This can be seen best in figure 6.6 on the facing page. It shows how many move directions on average the Moves variable of the ant that is moved contains. One of the directions is chosen randomly (see listing 5.1 on page 27). One ant in species B moves more randomly than the other one. The distinguishing feature of the two subspecies of B (B1 and B2) is which one of the ants is the more randomly moving one.

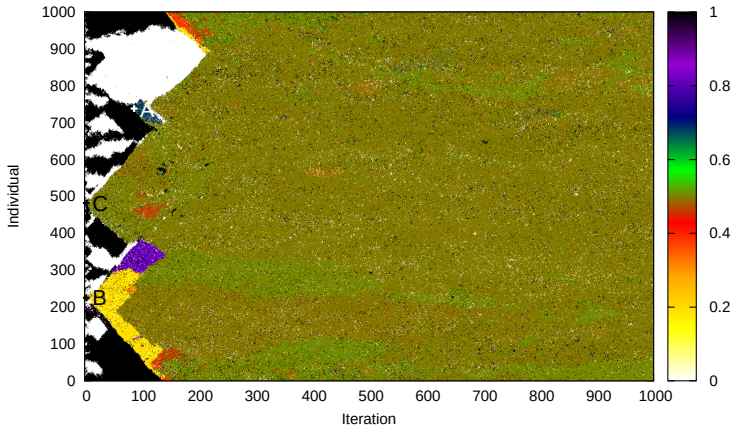



Figure 6.7: Probability that the decision *Function* returns the AntID of ant 1.

Figure 6.7 shows the probability of the decision *Function* returning the AntID of Ant 1. It has to be interpreted in the following way: During a game, every player can move at most 80 times. Every ant of this player can move 40 times, so the efficient thing to do is to use the moves of both ants. However, there is nothing that forces the developed players to do so. What happens most often is that the decision *Function* does not decide at all but returns a constant AntID. This ant will then be moved 80 times but of course after the 40 available moves of the ant are spent it will stay at its position. This behaviour can be seen in figure 6.7. In the beginning, a big fraction of the population consists of decision *Functions* that constantly return the same AntID, so the probability of the decision *Function* returning the AntID of ant 1 is either zero or one. At the points A, B and C emerge decision *Functions* that do not return constant values. From here on out, a point where a distinguishing characteristic is first observed will be called spawn point. The spawn point A is the source of species A. Spawn point B is a predecessor of species B. The decision *Function* emerging at B favours moving ant 2. However, using both ants is favorable to using only one ant, even if they are not both used to their full potential, so the movement *FunctionGroup* spreads (as the decision *Function* is a member of this *FunctionGroup* and selection is based on *FunctionGroup* scores). It is interesting to note that while the decision *Function* originating at B spreads, it changes "sides" at one point and prefers ant 1. This variant continues to spread in one direction but cannot penetrate the positions already occupied by the original decision *Function*. About 50 generations after the first occurrence of

the decision *Function* at spawn point B it changes so that it moves both ants equally and so species B is born.

The strategy that is used to move the ants is still unknown though, B could first exhaust the possible moves of one ant and then use the other ant, or use one ant for one move and than the other. Figure 6.8 shows the average elapsed time in a game until both ants have left the starting position. This is an indicator for the used movement strategy, if the average time is 40 then the moves of one ant were exhausted before the other ant was used. If both ants leave their starting positions earlier, then some intermittent ant switching has occurred. The graph shows that beginning with generation 400, the whole population moves one ant after the other. It also shows an aspect of the development of species B that was previously not recognizable. In the beginning, it did not exhaust one ant before it used the other. Only later did it switch to that mode, while the original method held on for 250 generations before it went extinct.

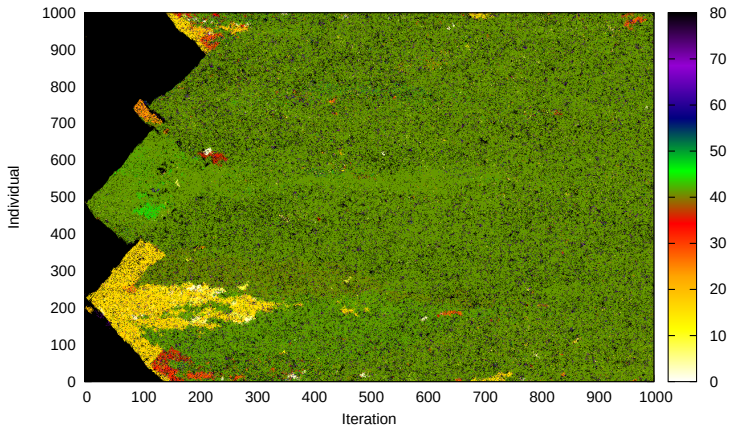



Figure 6.8: Average time until both ants have left their starting positions.

## 6.2 Belief

This section presents the development of the belief *Functions*. Figure 6.9 on the next page shows its fitness during the run. One should keep in mind that the score of the *Function* is the deviation of the food belief from the real (unobserved) state, so lower values are better. Since the enemy ants do not move, the belief *Function* achieves perfect scores rapidly. It is noticeable that the species inside the population with a medium score (around 1000) resist the destructive effects of mutation and crossover better than later species. When the perfect score is reached, a lot of *Individuals* show an extremely bad performance. Figure 6.10 on the facing page, depicting the size of the belief *Functions*, provides an explanation. The *Functions* are extremely small and so a lot of changes will be destructive. This is the downside of using the size of the *FunctionGroups* as decision criteria, when a lot of *Functions* reach the perfect score, then the small ones win, even though they are brittle. In the following runs the second player will also move and excessive *Function* shrinking will be of no concern. The graphs also show the reason why species C went

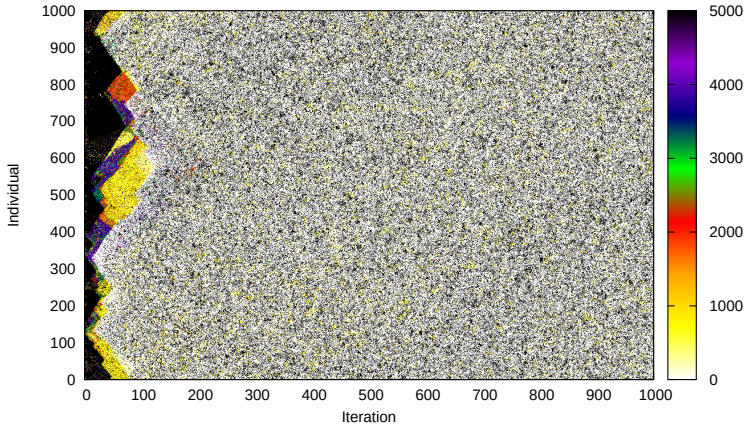



Figure 6.9: Development of the deviation of the belief *Function*.

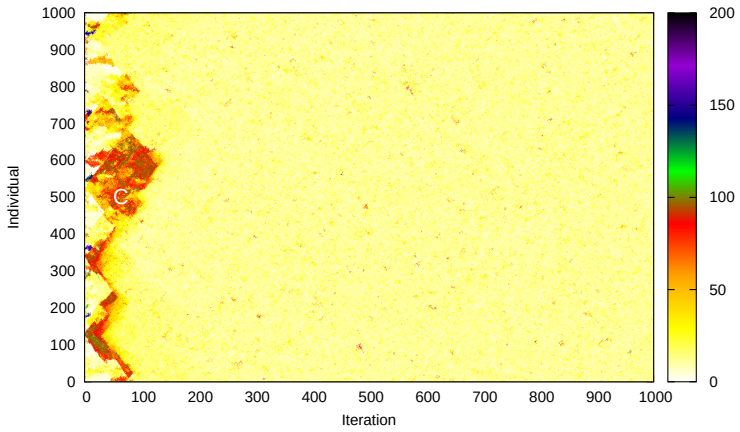
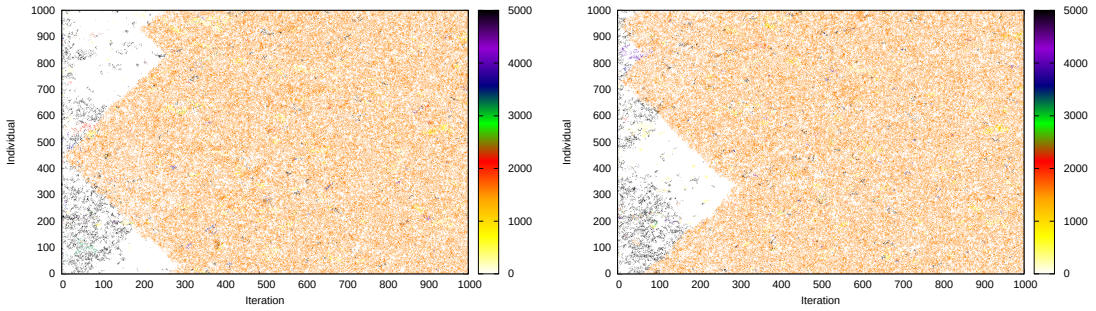



Figure 6.10: Size of the belief *Functions*.

extinct even though it was clearly superior to its neighbours. It depended on its belief *Function* to behave in a non-optimal way and when those were replaced by "better" performing ones species C could not survive.

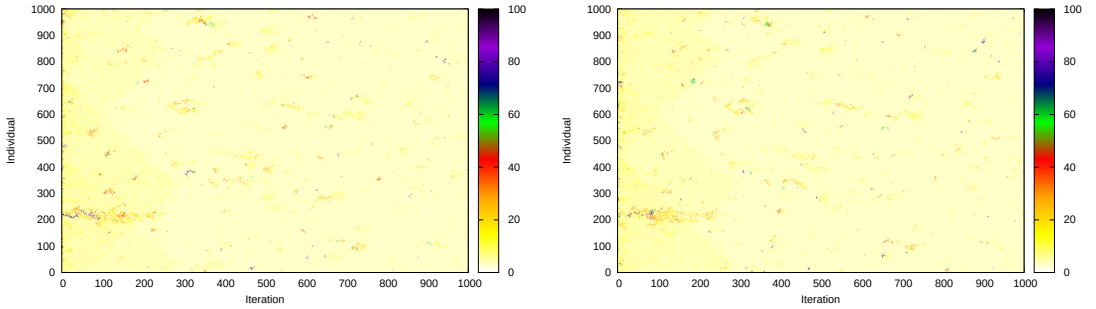
## 6.3 Prediction

The predict *Functions* did not have anything to do in this run as the opponent ants stayed at their starting positions. In the spirit of establishing a base line with which to compare further results, figure 6.11 on the next page shows the development of the score and figure 6.12 on the following page the development of the size of the predict *Functions*. Like the belief *Function*, the predict *Functions* suffer from the problem of brittleness due to a very small size. It is also interesting to note that a slightly smaller but far more susceptible to destroying changes species took over the




(a) predict 1 (b) predict 2

Figure 6.11: Fitness development of the predict *Functions*.




(a) predict 1 (b) predict 2

Figure 6.12: Size development of the predict *Functions*.

population. Another effect worth mentioning is that size anomalies (i.e. suddenly bigger than the surroundings) occur in the populations of the *Functions* at the same positions due to the allowed crossover between them.

## 6.4 General Performance Observations

Figure 6.13 on the next page shows the development of the total population size during the run. As is expected with genetic programming, the size of the population increased steadily from 100000 *Statements* in the beginning to 400000 in the end. Note, however, that the increase is not monotonic. There are phases where the population is shrinking (in terms of number of *Statements*). Figure 6.14 on the facing page shows how the size of the population is distributed among the *Individuals*. A correlation between *Individual* size and fitness can only be observed during the early generations; the spawn points B and C from figure 6.7 on page 45 can be recognised. After that, size is independent from fitness, i.e. code bloat can be observed. It is interesting to note that species B barely grows, which is in stark contrast to species A. Due to the exploding population size, one would expect the time needed to evaluate it would also increase.

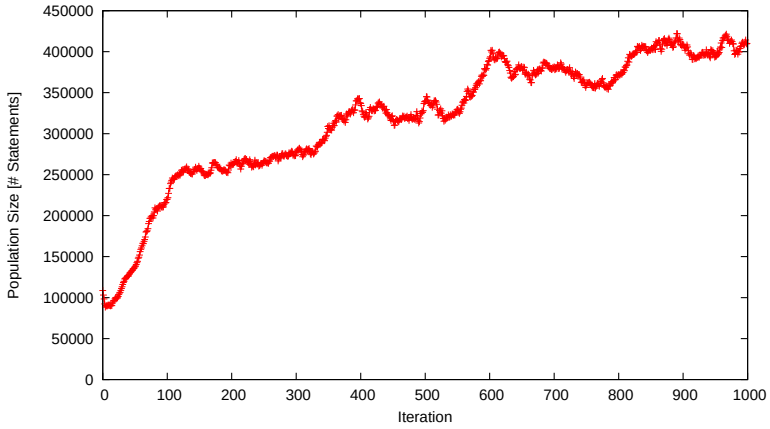



Figure 6.13: Size of the population in number of *Statements*.

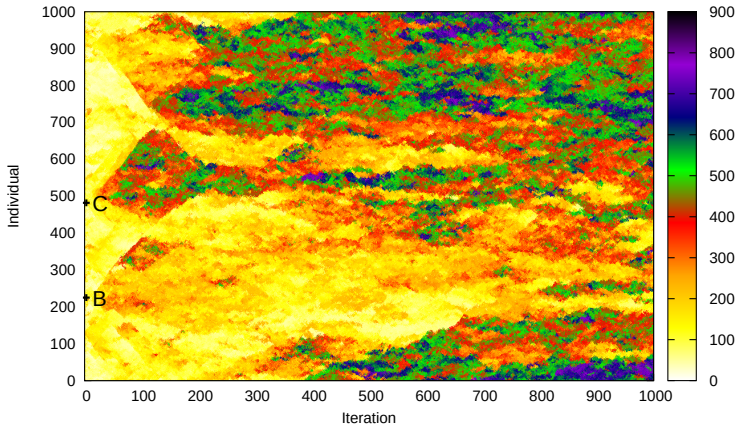



Figure 6.14: Development of *Individual* size.

However, as can be seen in figure 6.15 on the following page, this is not the case. The time to evaluate a generation even decreases during the first 100 generations and then stays more or less constant (but still with a downward tendency). Keep in mind that the values presented are the CPU time of the GPS process. This does not include the time spent waiting for processes GPS started (compressing the logging information after every generation and, more importantly, compiling) and the values are, as usual for run-time measurements on PCs, imprecise.

When using genetic programming it is interesting to analyze the influence of the mutation and crossover operators on the size of the *Individuals*.

To that effect figure 6.16 on the next page shows the size change of an *Individual* after applying the mutation operator. More than 5000 mutation events per generation generate no size change at all, in the graph the count was clamped at 30 to make other values visible. A growth of two, three or four *Statements* seems to be a common result of a mutation. This is most

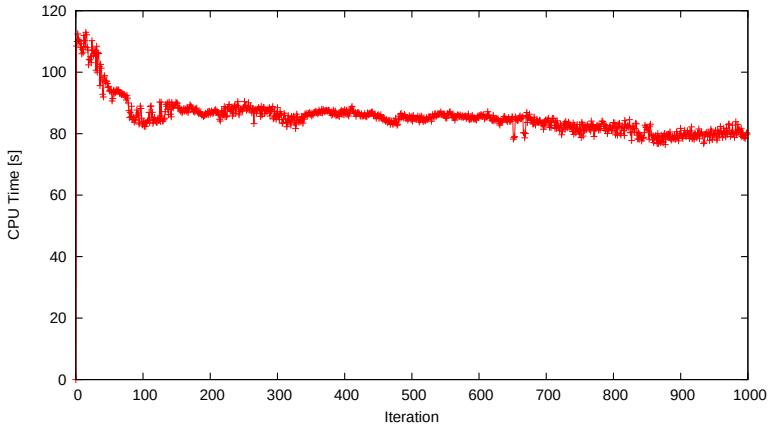



Figure 6.15: CPU time needed to calculate one generation.

likely due to the replace mutation that uses the same mechanism as the population initialisation to grow the replacement *Statement* trees, which have a minimum depth. This seems to indicate that the mutation operator has a strong growth bias; however, when looking at table 6.1 on the facing page it can be seen that the bias is not strong at all, but indeed present. The values are different for each *Function* because they are allowed a different amount of *Statements* and have a different set of *Statements* available which has an influence on the success rates of mutations.

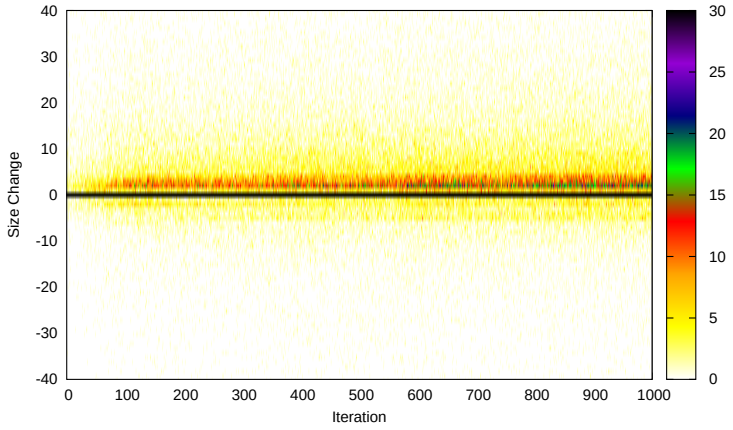



Figure 6.16: Number of mutation events creating a particular size change per generation.

Figure 6.17 on the next page shows the size change of an *Individual* after applying the crossover operator. The counts were clamped at 200 to make infrequent size changes more visible. A change of zero occurred more than 4500 times per generation, $\pm 1$ occurred about 200 times per generation. There is no obvious bias visible. Table 6.2 on page 52 shows that there is a bias, but in both directions. The decision, movement1 and movement2 *Functions* even shrink on average. This of course begs the question why the *Individuals* still grow even though

Table 6.1: Size change ($\Delta_S$) of *Individuals* due to mutation.

| Function | $\overline{\Delta_S}$ | $\sigma_{\Delta_S}$ | $\Delta_{Smin}$ | $\Delta_{Smax}$ |
|---|---|---|---|---|
| belief | 0.056 | 1.533 | -79 | 94 |
| decision | 0.028 | 3.556 | -197 | 137 |
| movement1 | 0.073 | 3.913 | -283 | 147 |
| movement2 | 0.070 | 3.901 | -287 | 162 |
| predict1 | 0.010 | 0.646 | -48 | 95 |
| predict2 | 0.012 | 0.721 | -21 | 97 |

mutation and crossover do not create bigger *Functions* on average. The answer is that the shrunk *Individuals* are less likely to be selected in the next iteration than others because some important parts of the *Statement* tree were destroyed and the fitness will suffer. Changes that cause a growth in size may also destroy important parts but with a smaller probability, so the smaller *Individuals* get weeded out by selection and the population grows. From tables 6.1 and 6.2 on the following page it can be seen that the crossover operator can create more growth than the mutation operator, because it can take a big subtree from the donor *Function* and insert it near to the leaves in the receiving *Function*. Mutation has to create new *Statement* trees, and their maximum depth (and so the amount of *Statements* they can contain) is limited.

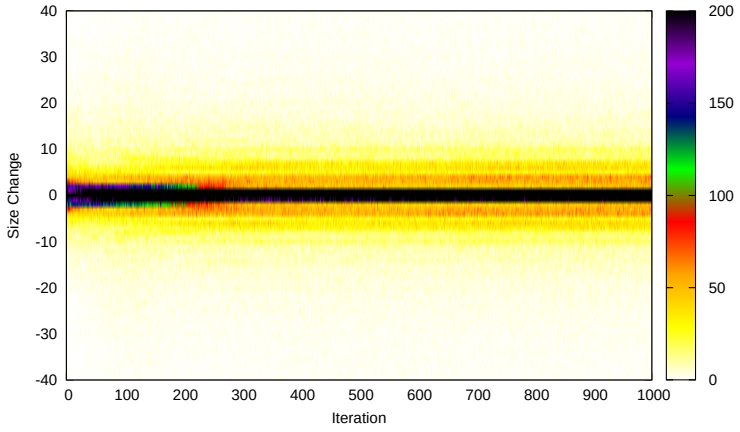



Figure 6.17: Number of crossover events creating a particular size change per generation.

## 6.5 Best Individuals

This section analyzes the properties of the best *Individuals* per generation. Figure 6.18 on the next page shows the positions of the best *Individual* per generation. The spawn points A,B and C from figure 6.7 on page 45 can be easily recognised. There is even an additional spawn point (labeled D in figure 6.18 on the next page) that was not obvious from other figures. The fitness

Table 6.2: Size change ($\Delta_S$) of *Individuals* due to crossover.

| Function | $\overline{\Delta_S}$ | $\sigma_{\Delta_S}$ | $\Delta_{Smin}$ | $\Delta_{Smax}$ |
|---|---|---|---|---|
| belief | 0.023 | 3.818 | -160 | 94 |
| decision | -0.774 | 14.317 | -197 | 195 |
| movement1 | -0.120 | 13.867 | -291 | 285 |
| movement2 | -0.120 | 13.894 | -292 | 287 |
| predict1 | 0.062 | 1.407 | -82 | 96 |
| predict2 | 0.058 | 1.392 | -80 | 96 |

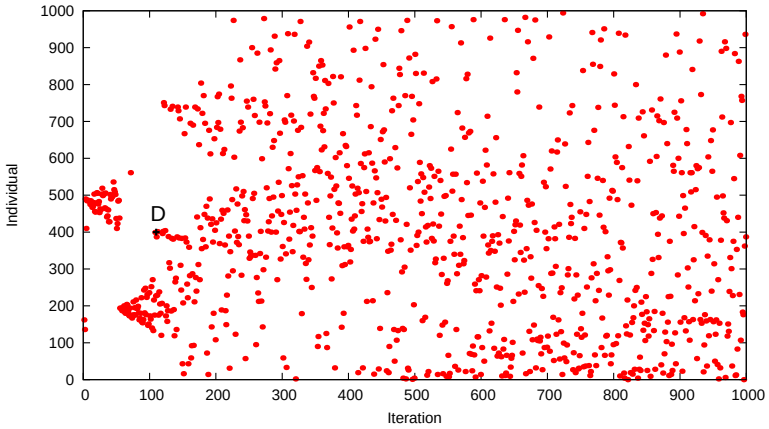



Figure 6.18: Positions of the best *Individuals* per generation.

development of the best *Individuals* was already shown in figure 6.2 on page 42. The size of the best *Individuals* fluctuates wildly between 100 and 800, showing that relatively small *Individuals* are sufficient to successfully play 2-AntWars when the opponent does not move.

## 6.6 Conclusion

The analysis of the run showed that the 2-AntWars model is suitable to create players that are proficient at finding food on the playing field and use both ants effectively. Interestingly, the developed players use the ants one after another instead of alternating the moved ant which is what a human player would most likely do. It was also seen that the population model was conductive to creating different species inside the population without an explicit method to do so, like partitioning the population to create dedicated space for different species.

CHAPTER 7

# Strategies Version 1

This chapter presents the results of trying to evolve players capable of beating the first implementation of the three discussed strategies in section 3.3 on page 14. During this and the following chapters, player 1 designates the player that is developed by GPS, player 2 is the player moving according to a fixed strategy.

## 7.1 Greedy

This section presents the results of a run against the Greedy strategy. First the implementation of the strategy is discussed before the run is analyzed in detail.

### Implementation

Movement1 and movement2 of the greedy player are the same. During the first four moves, it returns twice north-west and once north for ant 1 and twice south-east and once south for ant 2 to separate the ants enough so that they chase different pieces of food. Later, the *Function* returns immediately scoring moves if such moves exist. Otherwise the immediately scoring moves from the positions north-east, north-west, south-east and south-west to the current position of the ant are compared and the direction with the best scoring potential is returned (if it has at least one scoring move). Then the moves uncovering the most unseen fields are calculated and returned if there are moves that uncover unseen fields. In the case that still no moves have been returned, the nearest food position and nearest unseen field position and their distances to the position of the ant are calculated. If the distance to the position of the food is not larger than the distance to the nearest unseen field then the moves towards the food, otherwise the moves towards the unseen field are returned.

The decision *Function* for the greedy player alternates between moving ant 1 and 2. It also prioritizes moves suggested by the movement *Functions* in the following order: for ant 1 north, south, north-west, south-west; for ant 2 south, north, south-west, north-west. This is done to improve collection efficiency; before moving away from the starting positions to reach food,

more local positions are tried. Ant 1 prioritises north movements, ant 2 south movements to keep them distant from each other and better explore the playing field (diagonal moves generally uncover more positions). If the movement *Function* of the ant that is about to be moved did not suggest any moves, then north-west and south-west moves are added to continue moving across the playing field towards the opponent's half.

The belief *Function* for the greedy mover behaves in the perfect way if the opponent does not move. It returns one for every unseen field and every seen field that contained food and zero otherwise.

The prediction *Function* for the greedy mover just returns the current prediction because the prediction information is not used in any way.

## Results

Figure 7.1 shows that GPS was able to develop players that beat the implementation of the greedy strategy. It took six generations until the developed player beat the greedy strategy for the first time, by collecting two pieces of food more over 50 games than the opponent (out of a total of 1600 pieces of food available). The following generations were won by the greedy player but GPS continued to improve the player (and the frequency of winning) so that in the last generation player 1 was able to collect 100 pieces of food more than player 2.

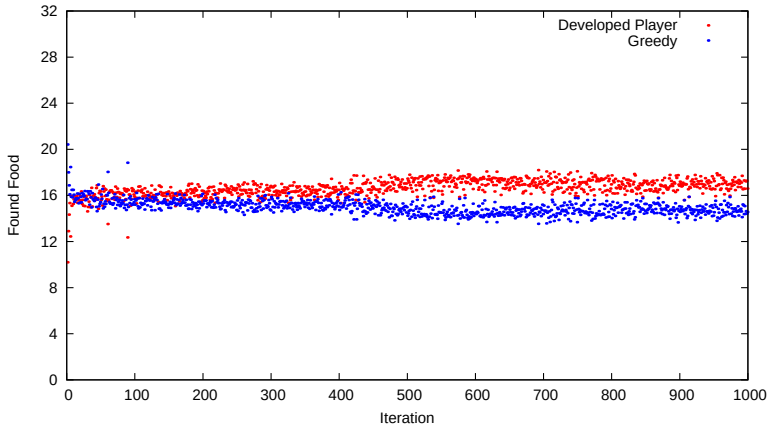



Figure 7.1: Average number of found pieces of food of the best *Individual* per game.

Figure 7.2 on the next page gives an overview of the fitness development of the population. It can be seen that one species takes over the population after 400 generations have passed, labeled as A. There are two subspecies (B, C). B is superior to its surroundings and still dies out. Species C emerges inside of A and is superior to its parent species. This is barely visible in figure 7.2 on the facing page, but figure 7.3 on the next page shows this clearly.

When analyzing how the collected food is distributed among the ants, the situation depicted in figure 7.4 on page 56 emerges. Player 1 mostly uses his second ant to collect food while the second player's ants are equally utilized (as expected because they were programmed that way). It is surprising to see that ant 1 of player 1 mainly finds only one piece of food. Figure 7.5 shows

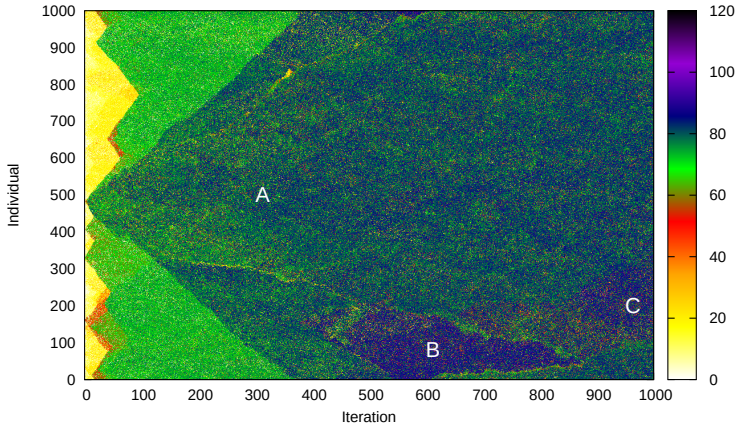



Figure 7.2: Found food of the *Individuals* playing 2-AntWars against a Greedy player.

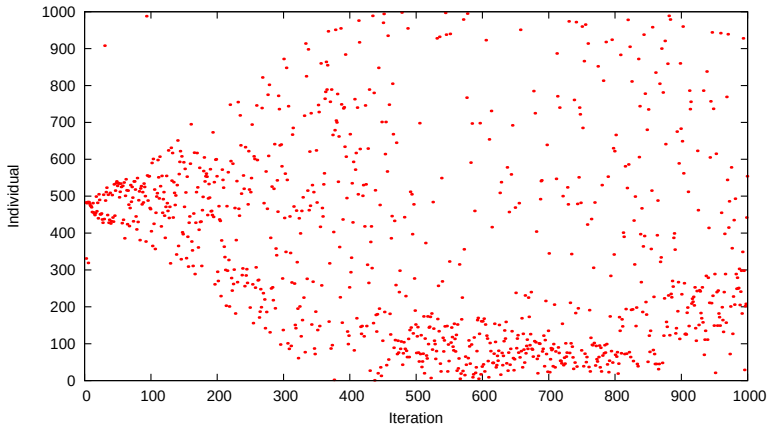
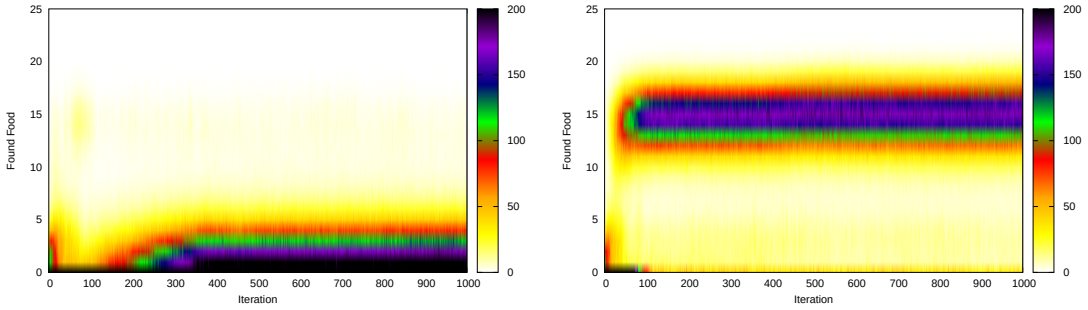



Figure 7.3: Location of the best *Individual* inside the population of a particular generation.

that Player 1 uses one ant until it cannot move anymore before the other one is moved. Combined with the fact that ant 2 collects more food than ant 1 this means that player 1 uses ant 2 first and then ant 1. Not every *Individual* of the population uses this method though, figure 7.5 on the next page shows at least one distinct subspecies (labeled D) that moves both ants earlier.

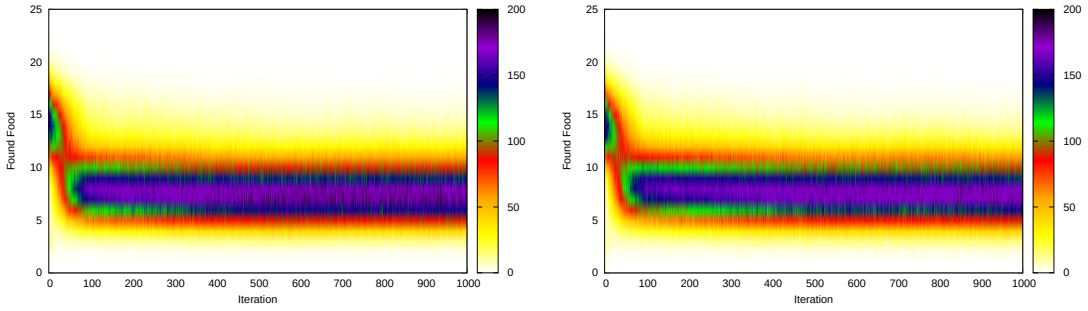
It was shown before that there is a species inside the population that is superior to its neighbors but still dies out (species B, figure 7.2). Figure 7.6 on page 57 shows the reason: The belief *Function* that B uses gets replaced by a slightly better (but a lot brittler) *Function*. Now that player 2 actually moves, the population does not converge to extremely small and extremely brittle *Functions* as seen in the run without adversary, but still the better belief *Functions* seem to be more susceptible to catastrophic change.

Figure 7.7 on page 57 shows the fitness of the belief *Function* that was used by the best *Individual* in each generation. Three distinct levels of fitness can be observed, the first corre-




(a) Player 1 Ant 1

(b) Player 1 Ant 2




(c) Player 2 Ant 1

(d) Player 2 Ant 2

Figure 7.4: Distribution of found food per ant.

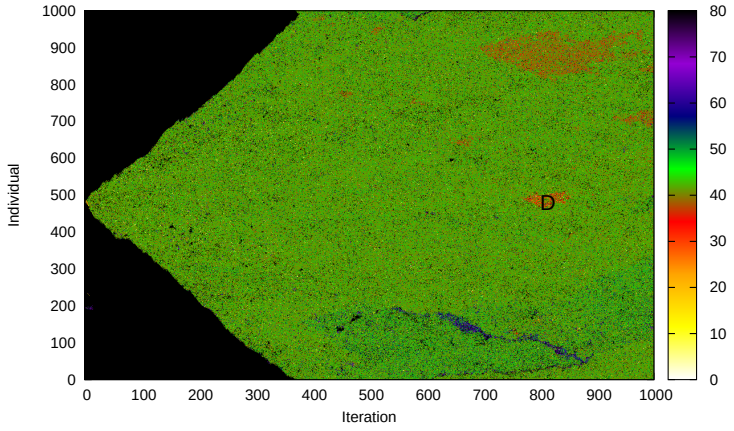



Figure 7.5: Average time when both ants of the developed player leave the starting position.

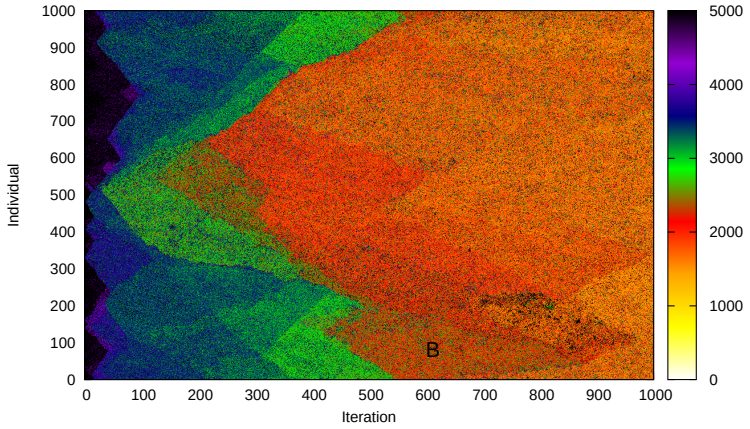



Figure 7.6: Development of the belief deviation.

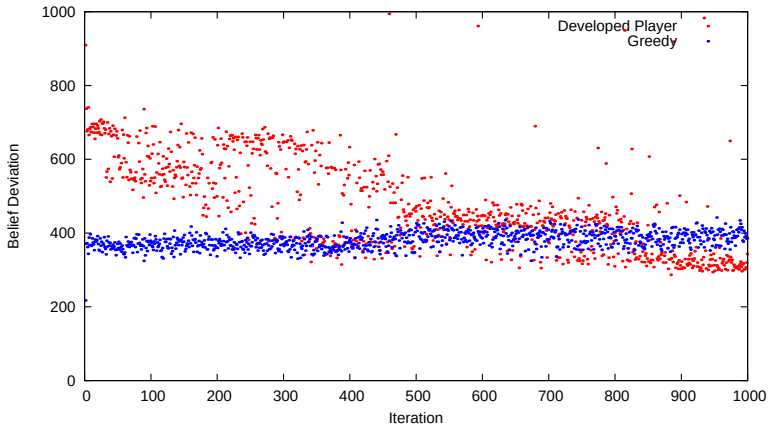



Figure 7.7: Average belief deviation of the best *Individual* per game.

sponding to species A, the second to species B and the third to species C. Only the third level is better than the belief *Function* of player 2, which shows that it is hard for GPS to find a suitable belief *Function*. The belief *Function* in the last generation has an average deviation of $343.4$ per game. It reaches that deviation by beginning with a belief of one at every unseen position and slowly reducing it until it reaches about $-0.07$ when the game ends (the actual value depends on the length of the game). The belief depends on the time that has passed but not on the predicted positions of the enemy's ants (or any position at all). The belief in food that was seen but is currently not visible is $0.8$.

The fitness of the other auxiliary *Functions* (predict1 and predict2) of the best *Individual* per generation is shown in figure 7.8 on the next page. It can be seen that the performance of player 1's predict *Functions* is relatively stable compared to player 2's *Functions*. Player 2 is very bad at predicting the position of ant 2 of player 1 because as determined earlier, this ant is

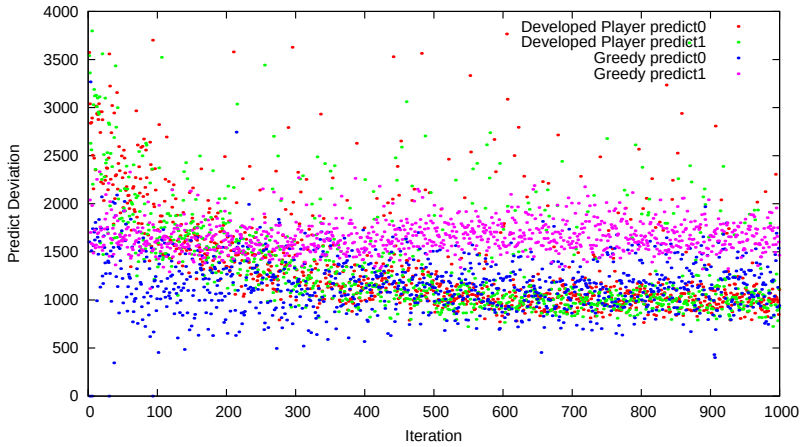



Figure 7.8: Average predict deviation of the best *Individual* and its opponent per game.

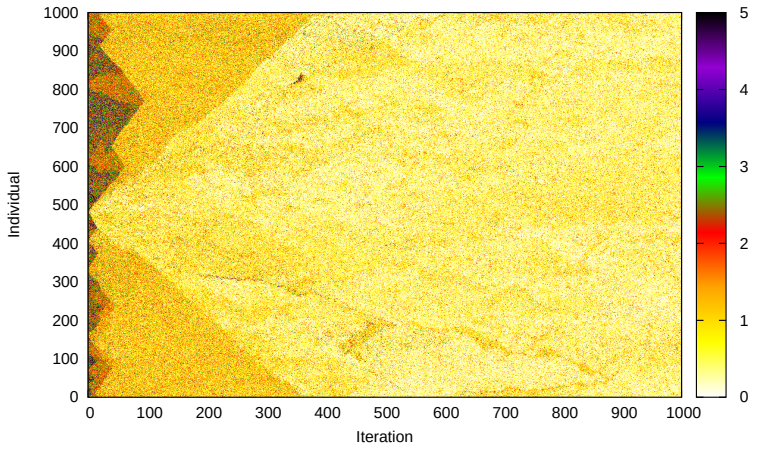
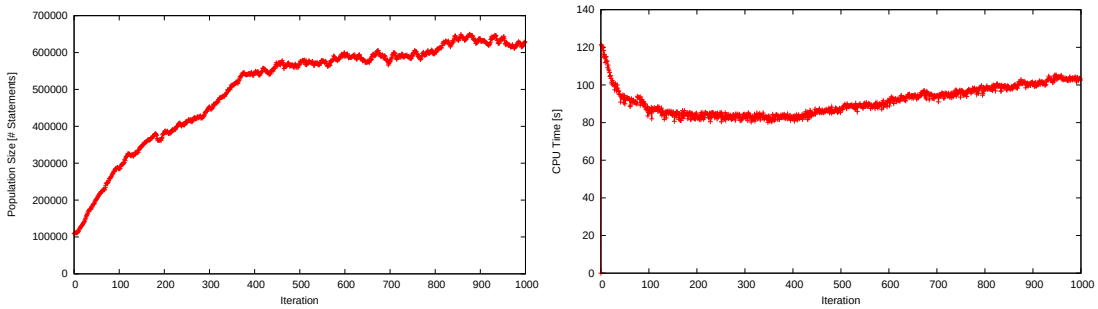



Figure 7.9: Average food remaining on the playing field after a game.

moved first so that predicting it at the starting position is wrong during the whole game. The prediction of ant 1 of player 2 is not so bad and comparable to the performance of the prediction *Functions* of player 1. It is, however, less stable. Player 1 reaches his prediction performance by moving the predicted positions one field to the west of the current prediction every third move to approximate the positions of the enemy's ants.

The average food remaining on the playing field after a game is presented in figure 7.9. It starts with a maximum of 12, but as the *Individuals* in the population increase their fitness, less and less food remains on the playing field so that in the last generation only $0.52$ pieces of food remain uneaten per game. This shows that when two players are active nearly all pieces of food can be collected without needing a suitable positioning bias as was the case when only one player was moving.

The total size of the population and the time needed to evaluate it are depicted in figure 7.10. As was already seen in the run without adversary, the time needed to evaluate a population is not influenced by its size in *Statements*. The fitness of the players is more influential as fitter players find all of the available food faster which shortens the games and reduces the computational burden. From generation 300 onward, the size of the best *Individual* is above 300. This indicates that playing successfully against moving opponents requires more complex programs than playing against stationary opponents.




(a) (b)

Figure 7.10: (a) Total number of *Statements* contained in the population and (b) the time needed to evaluate them.

## 7.2 Scorched Earth

This section discusses the run against the Scorched Earth strategy. As before, the implementation will be presented before the result is analyzed.

### Implementation

The decision *Function* lets every ant move once before the other ant is moved.

The movement *Functions* move the ants in the following way: When the game begins, movement1 moves the ant to position $(9, 2)$, movement2 to position $(9, 10)$, which brings them one position into the playing field half of player 1. Food is ignored on the way. Then the ants are moved north and south until all food on positions with the same y-coordinate as theirs is eaten or all positions have been seen. After that, the ants are moved west once and the process is repeated. Figure 7.11 shows the movement path of the ants with this implementation of the Scorched Earth strategy.

The belief and predict *Functions* are the same as the ones the Greedy strategy used, i.e. belief as if the other player did not exist and predict always the last prediction.

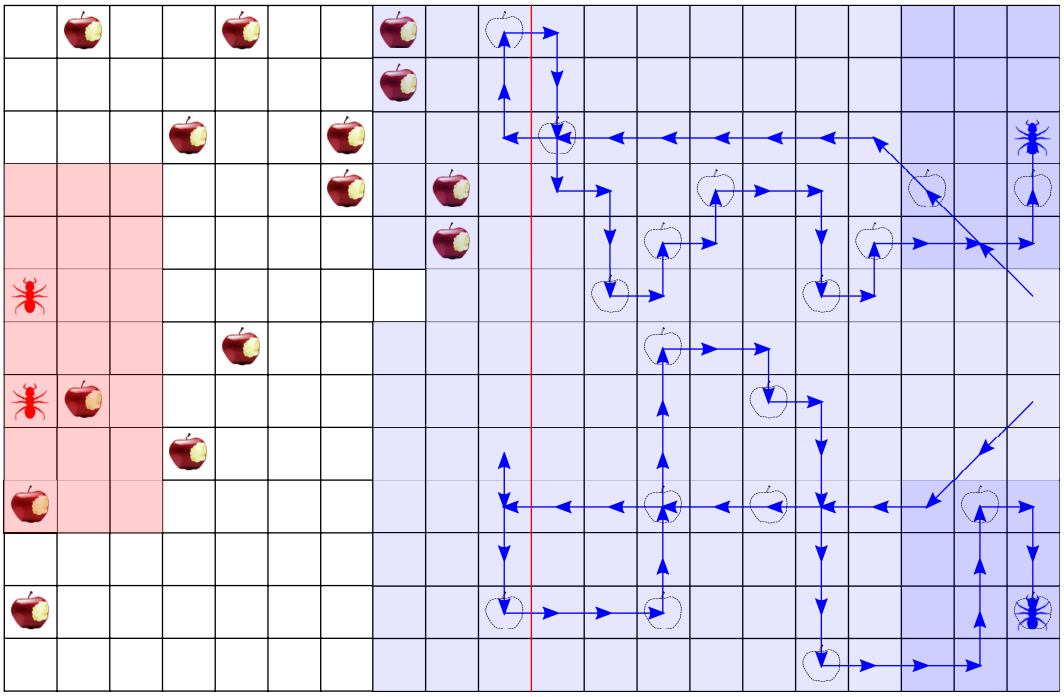


Figure 7.11: Player 2 using the first implementation of the Scorched Earth strategy.

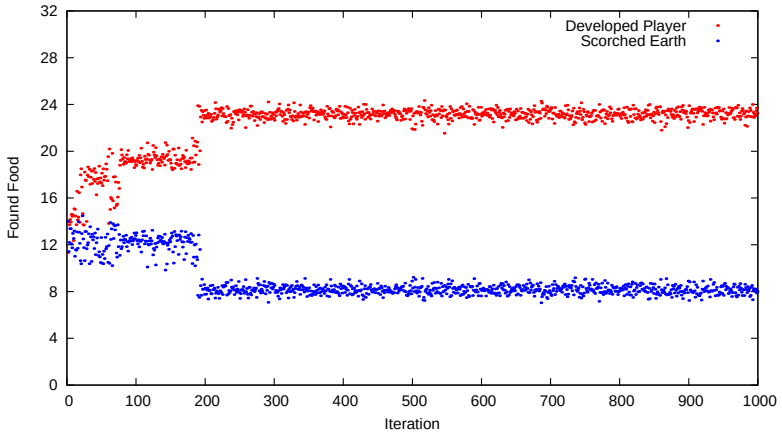



Figure 7.12: Average number of found pieces of food of the best *Individual* per game.

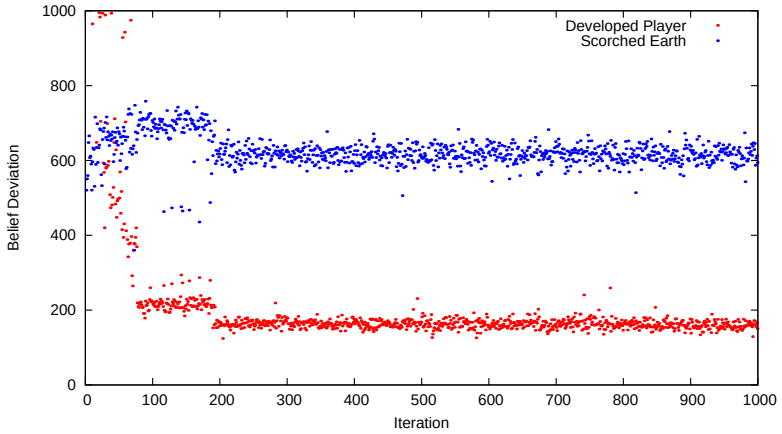



Figure 7.13: Average belief deviation of the best *Individual* and its opponent per game.

## Results

Figure 7.12 shows that GPS had no problem developing players that are capable of beating the first implementation of the Scorched Earth strategy. Beginning with the second generation, the best *Individual* always performed better than its opponent (with the exception of generation 9). During the first 100 generations, the fitness of the *Individuals* improves steadily, then it stagnates for 100 generations before it suddenly jumps to its final value. The reason for this last improvement was found by watching the games of the best *Individuals* of generation 150 and 200. In both games the players exploited the rigidity of the Scorched Earth player and sent one ant into his half while his ants were still moving towards the center of the playing field. There they collected some food, ensuring that the Scorched Earth player cannot win. One might say that Scorched Earth was beaten by a more efficient form of Scorched Earth. The innovation of

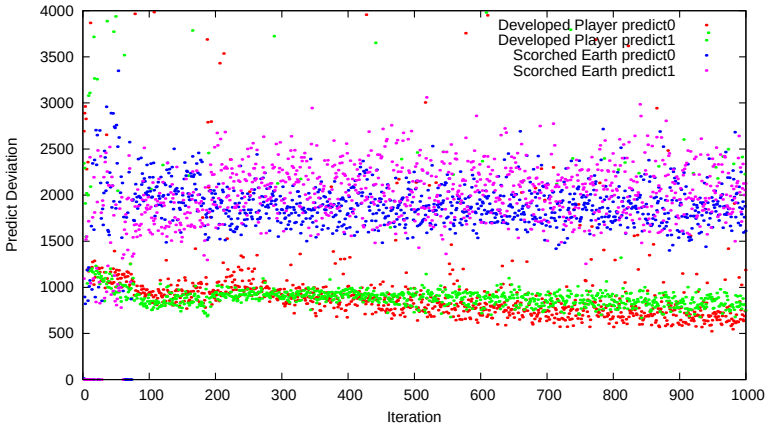



Figure 7.14: Average predict deviation of the best *Individual* and its opponent per game.

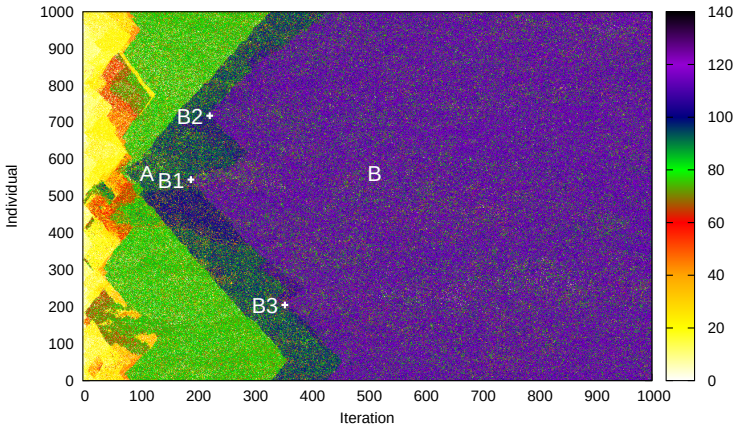
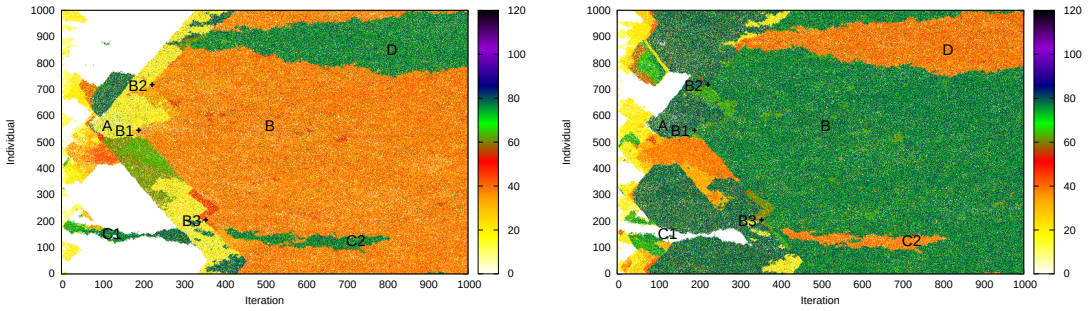



Figure 7.15: Found food of the *Individuals* playing 2-AntWars against a Scorched Earth player.

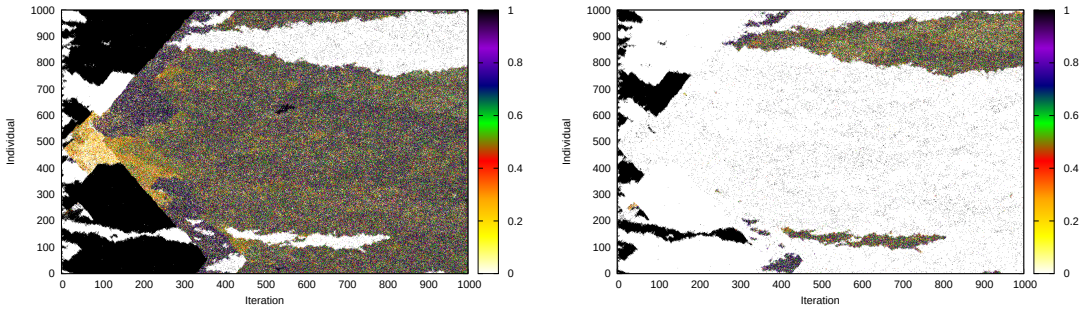
the player at generation 200 was that the ants did not always act greedily while moving towards the half of player 2. There were situations where the ant could have moved one field backwards to eat one piece of food but did not do it. Instead, the ant moved towards the enemy's half of the playing field (not ignoring the food on the way), was there earlier than the player of generation 150 and collected more food there.

The average belief deviation of the best *Individual* per generation depicted in figure 7.13 on the preceding page shows the same development as the amount of eaten food. First it improves steadily and then it stagnates before jumping to its final value. The average deviation of the Scorched Earth player is significantly higher than the deviation of the Greedy player (see figure 7.7 on page 57), even though they use the same *Function*. The reason for that is that the Scorched Earth player believes that the food he passes on his way to the center of the playing field stays there, but in reality it is eaten by the developed player .




(a) Ant 1 (b) Ant 2

Figure 7.16: Found food of the ants of player 1 against the Scorched Earth strategy.




(a) Ant 1 (b) Ant 2

Figure 7.17: Probability that an ant of player 1 is still movable after the game has ended.

Figure 7.14 on the facing page shows that the predict *Functions* reached the same level of precision as in the run against the Greedy player, but the method by which it was reached was different. The predict1 *Function* constantly returned $(14, 3)$ as prediction, predict2 returned $(14, 9)$. Instead of moving the prediction around, it was placed at the center of the path the ants move on.

The overview of the fitness development of the run against the Scorched Earth player is given in figure 7.15 on the preceding page. After 100 generations, species A formed and took over the population. At spawn points B1, B2 and B3 improvements to A emerged that formed species B which was the only species present in the population by generation 450. Figure 7.16 gives a more detailed insight into the development of the population by separating the performances of ant 1 and ant 2 of player 1. It shows that from this point of view, species A is not a coherent species at all but rather a conglomerate of at least three species. The change to species B was mostly achieved by improving the performance of ant 1 while the one of ant 2 stays the same. Species B is also not as homogenous as it seemed. The predominant behaviour is that ant 2 moves first until it has no moves left and then ant 1 starts to move (see also figure 7.17) but there are two species hidden inside it (C2 and D) that have switched the role of ant 1 and ant 2.

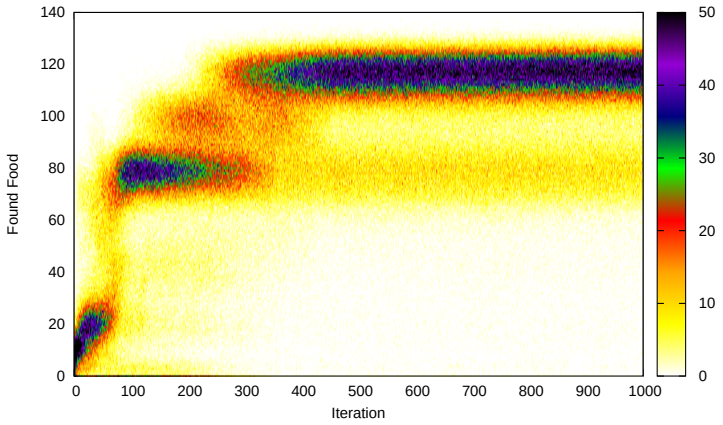
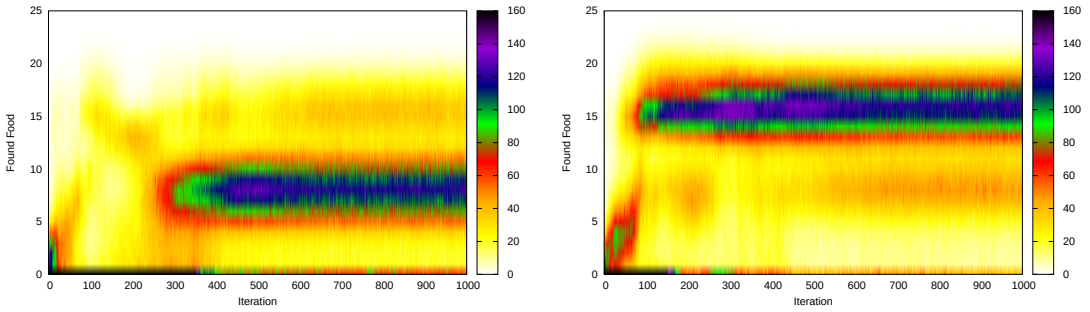



Figure 7.18: Distribution of found food of the *Individuals* playing 2-AntWars against a Scorched Earth player.

C2 died out over the course of the run but D remained stable. D is also interesting because it originated at the beginning of species A (at *Individual* 870) and not B. C2 even seems to have roots going back to the first generations, represented by C1. C1 only used ant 1 to gather food, but was successful enough to survive until species A emerged. It then resurfaced inside of B in an improved fashion, also using ant 2 to gather food. Beside the fact that C1 and C2 both dominantly use ant 1 and their location is the same, there is no further evidence that they are actually related, so these similarities could be coincidence.

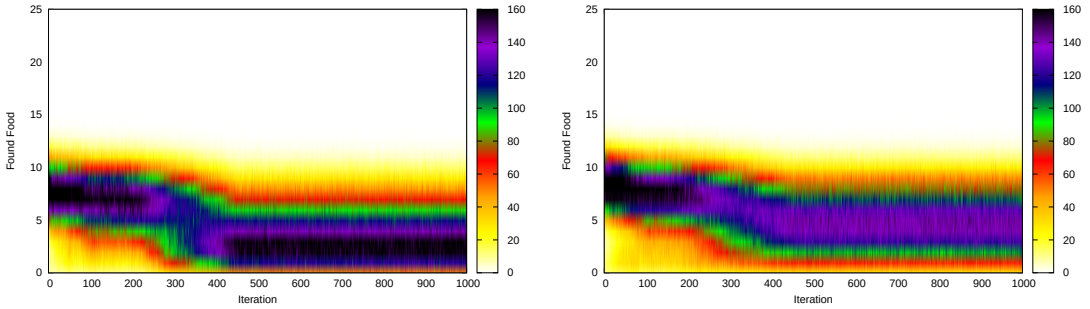
The fitness histogram depicted in figure 7.18 shows the three phases of the development of player 1. In the first phase, achieving about 80 food per match, the use of ant 2 was optimized. In the second phase both ants are used effectively and species A is born, achieving about 100 pieces of food per match. The third phase optimized the use of ant 1, resulting in players that score about 120 food per match. The optimizations increased the average number of positions seen per game which means that ant 1 had a stronger instinct to explore. Figure 7.19 on the next page corroborates this interpretation of the fitness histogram. It also shows (in figures 7.19c on the facing page and 7.19d on the next page) the abysmal performance of the Scorched Earth player.

Figure 7.20 on the facing page shows whether the predict2 *Function* of the developed player was changed during the course of evolution, be it by means of crossover or mutation. A large portion of the predict2 *Functions* inside the population was never changed. This can only happen if the result of a change is larger than the set maximum size (100 for predict *Functions*) and the original *Function* is kept. Normally it is possible for big *Functions* to shrink as a result of changes which improves the chances for successful changes even if they result in *Function* growth because they now have space to grow. This mechanism is not working for the unchanging predict2 *Function* because it starts off much bigger than the maximum size. This is possible because during the construction of *Individuals* only the tree depth matters, not the total amount of *Statements*, so very large *Functions* are possible. Normally they die out quickly because initial *Individuals* are generally unfit, but this one survived a long time. This could be an indication




(a) Player 1 Ant 1

(b) Player 1 Ant 2




(c) Player 2 Ant 1

(d) Player 2 Ant 2

Figure 7.19: Distribution of average found food per ant and game.

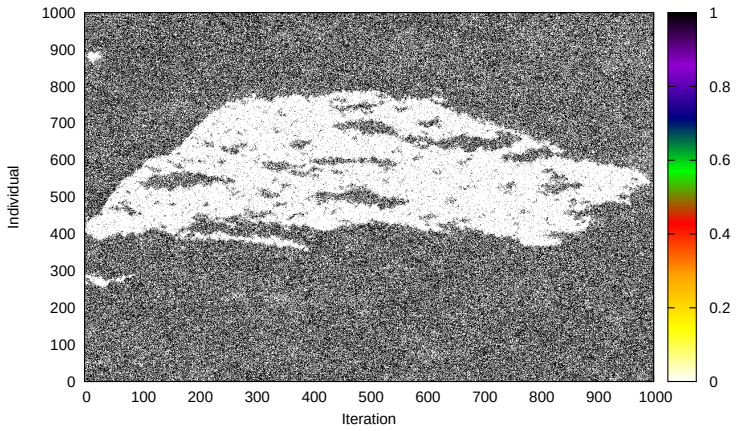
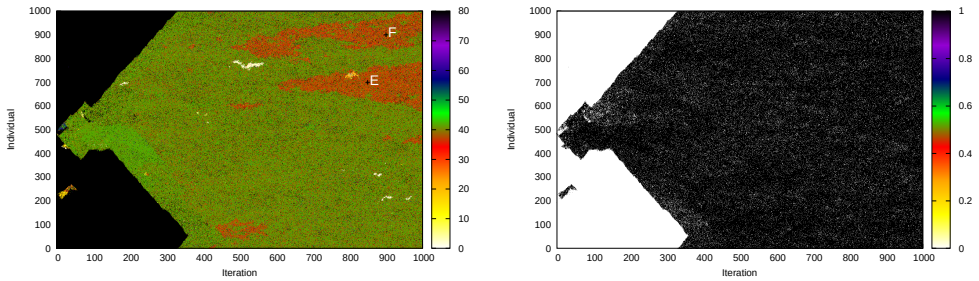
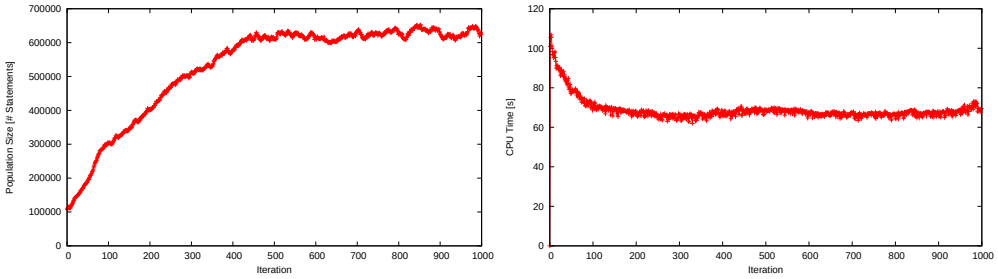



Figure 7.20: Changes to the predict2 *Function* of player 1. A changed *Function* is marked with the value 1, an unchanged *Function* with the value 0.




(a) First time both moved (b) Both collecting food

Figure 7.21: Effective use of both ants.




(a) (b)

Figure 7.22: (a) Total number of *Statements* contained in the population and (b) the time needed to evaluate them.

that the size limit of 100 is too small because a static large *Function* survived a long time against evolving and improving smaller *Functions*.

The effective use of both ants is analyzed in figure 7.21. Figure 7.21a shows the average of the first time per game when both ants have left their starting positions. The use of both ants was present in the population from the beginning, but only effective when species A emerged. There are also two additional species visible (E,F) that move both ants earlier. Note that F is a part of D. The population also contains some very small species that move both ants nearly at the beginning of the game. As figure 7.21b indicates, those small species use both ants effectively and are not simply defective *Individuals*. This is the behaviour one would expect from a human player (moving both ants "simultaneously" and collecting food with both), so GPS is indeed able to develop it; however, it does not survive against the more common strategy of moving one ant until the moves run out and then the other.

As always, the end of the discussion of a run is a look at the development of the size of the population and the time needed to evaluate it, provided by figure 7.22. The size of the population rises fast towards 600000 *Statements*, which is the same value that was seen in the discussion of the Greedy strategy. Again, the time needed to evaluate the population is independent of the actual size (in *Statements*) of the population.

## 7.3 Hunter

This section discusses the run against the Hunter strategy.

### Implementation

The Hunter implementation assigns two roles to its ants. Ant 1 is the hunter and therefore doing all the hunting, and Ant 2 is the gatherer, doing no hunting but tasked with collecting as much food as possible to make the implementation more productive. Movement1 does not return any moves because the move for the hunter ant is calculated by the decision *Function*, which has access to the position predictions of the enemy's ants. Movement2 uses the movement *Function* of the Greedy strategy, see section 7.1 on page 53 for its implementation. The decision *Function* decides to use the hunter ant as long as it is movable and the gatherer ant otherwise. If the hunter ant is chosen, its move is calculated in the following way: If the hunter ant is one move away from one of the ants, it attacks immediately. Otherwise the moves towards the enemy's ants (predicted) positions (if they were seen movable) and towards food that is reachable with one move are calculated. The intersections of the following moves are tried in that order and returned if they are not empty: all three moves, move towards nearer ant and food, move towards nearer ant. If this does not result in any move (for example if both ants were already neutralized) the movement *Function* of the Greedy strategy is used to determine the move of the hunting ant. The belief *Function* uses the same method to calculate the food belief as the Greedy and Scorched Earth implementations. The implementation of the predict *Functions* was a bit tricky, because it is not really possible to predict how the enemy's ants move before the run, so the prediction was simply moved randomly either one position north-east, east or south-east from the current prediction every second move. If the predicted position is visible and the enemy's ant is not on it then the nearest unseen position to the predicted position is chosen as new prediction.

### Results

Figure 7.23 on the following page shows that GPS was able to evolve players that beat the Hunter strategy, but it took 700 generations to do so. Interestingly, the developed belief *Function* was not able to beat the belief *Function* of the Hunter (figure 7.24a on the next page), unlike the two previous runs. The next figure (7.24b on the following page) shows that the predict *Functions* for the Hunter are not better than the "do not predict" *Functions* of the Greedy and Scorched Earth strategies and that the developed player uses prediction *Functions* that beat them in terms of precision. Also note that the prediction accuracy of predict1 is better than for predict2, which is no surprise because the hunting ant behaves more predictably than the gathering ant. It is also helpful that the hunting ant tries to be close to player 1's ants so that it is seen and therefore accurately predicted. The study of the games of the best *Individuals* showed that predict 1 of the developed player is very close to the actual position of the hunter ant on its way to the starting position of player 1.

The fitness development of the whole population (figure 7.25 on page 69) shows species A spread throughout the population to be replaced by the successors B and C. As figure 7.26 on page 69 shows, A spreads because it uses both of its ants to gather food. However, this

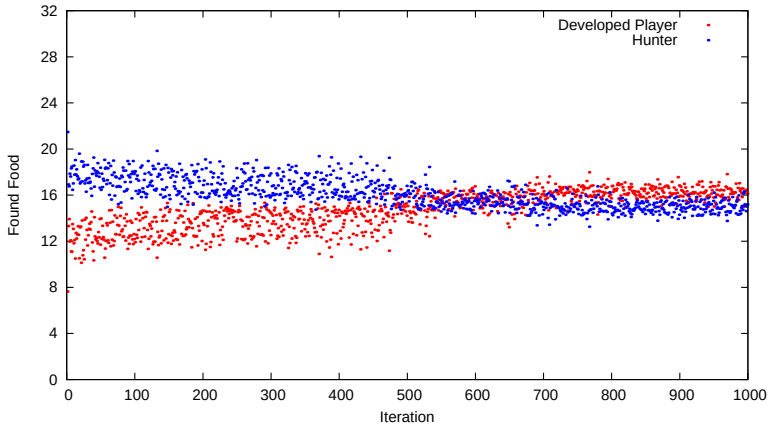
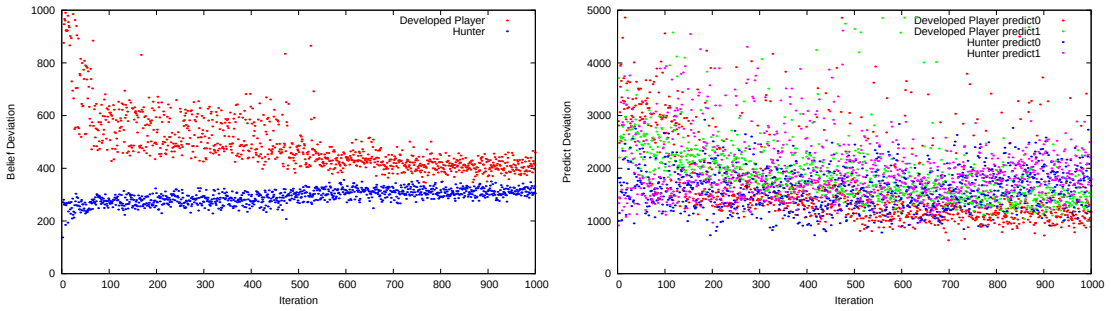



Figure 7.23: Average number of found pieces of food of the best *Individual* per game.




(a) Belief deviation

(b) Predict deviation

Figure 7.24: Average belief and predict deviations of the best *Individual* and its opponent per game.

feature does not spread as fast as in the previous runs. An explanation for this could be that the developed player uses his ants sequentially again (figure 7.31b on page 72) and when the hunter ant of player 2 reaches the starting position of player 1, it can neutralize the waiting ant at the starting position and using both ants to collect food is not possible any more. Figure 7.26 on the facing page also shows an interesting development concerning species D. The *Individuals* at D consist of a movement2 *Function* that emerged before A spread and was enhanced by decision and movement1 *Functions* that enabled the effective use of both ants. Also, species C changed tactics and actually moves ant 1 before ant 2. A part of species B also changed, as did three more species hidden in A. Seeing as this behavior is able to spread through the population it seems to have an advantage over moving ant 2 first. It may well be the deciding factor that allowed the developed player to beat the Hunter strategy. When watching the games that the best *Individual* played in the last generation, it becomes clear that ant 1 is used as bait to lure the hunting ant away. The hunting ant is also ant 1 which means it is more likely to find ant 1 of player 1 when

moving across the playing field. It then follows ant 1 (the lure) which is busy collecting food and moving away from the starting position (and from ant 2). When the hunter finally catches up with the bait, it has already eaten some food and is quite a distance away from ant 2, so ant 2 can begin to collect food unhindered. Furthermore, the hunter already spent a lot of moves on hunting and will not reach ant 2 in time. The gatherer ant does not hunt the remaining ant of player 1, so player 1 is able to win.

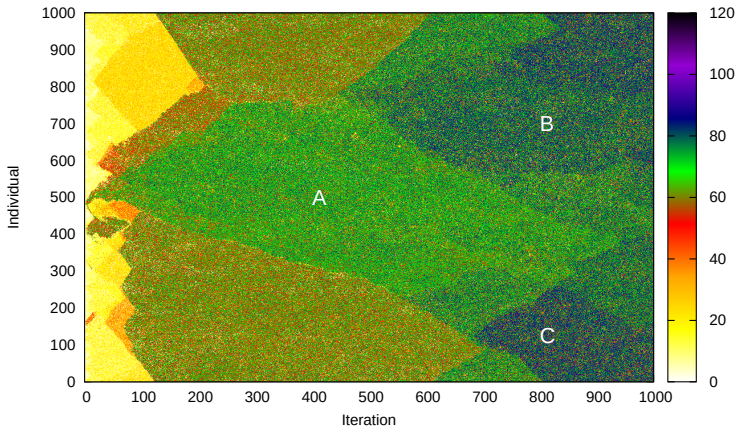
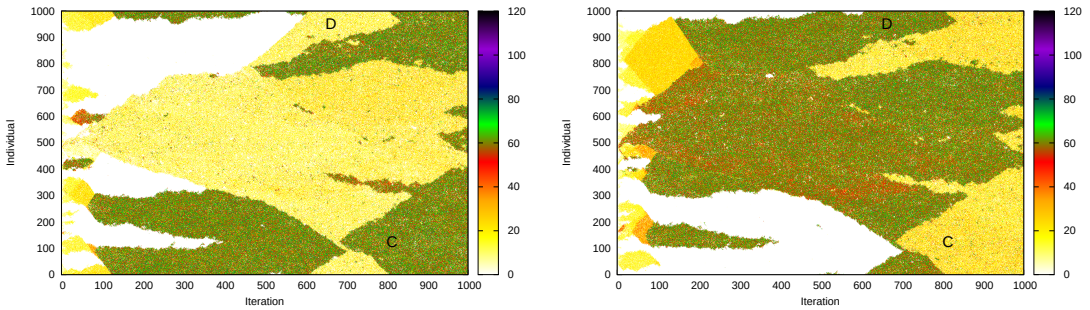



Figure 7.25: Found food of the *Individuals* playing 2-AntWars against a Hunter player.




(a) Ant 1

(b) Ant 2

Figure 7.26: Found food of the ants of player 1 against the Hunter strategy.

The histogram of the fitness development (figure 7.27 on the next page) shows that the population as a whole fails to reach the 80 pieces of food mark and that it is not as clearly split up into different performance levels as the populations in the previous runs were. The reason for this is seen in figure 7.28 on the following page. The performance of both ants of player 1 is distributed among the same two levels which are the result of choosing one ant or the other to move first. Unsurprisingly, the performance of ant 2 of player 2 is better than that of ant 1 because it is the gatherer ant's job to be good at collecting food.

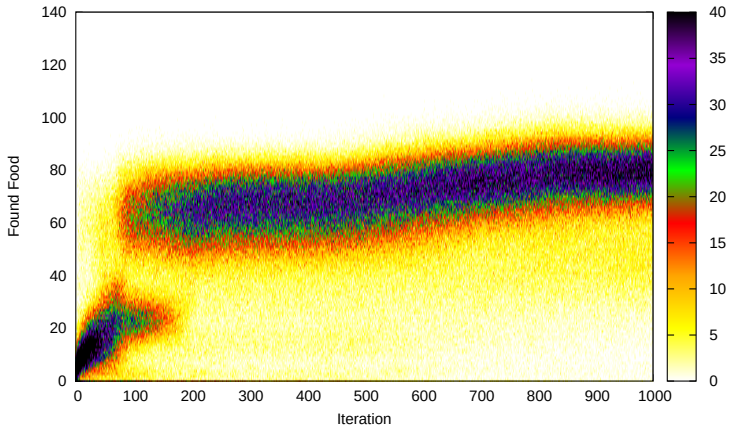
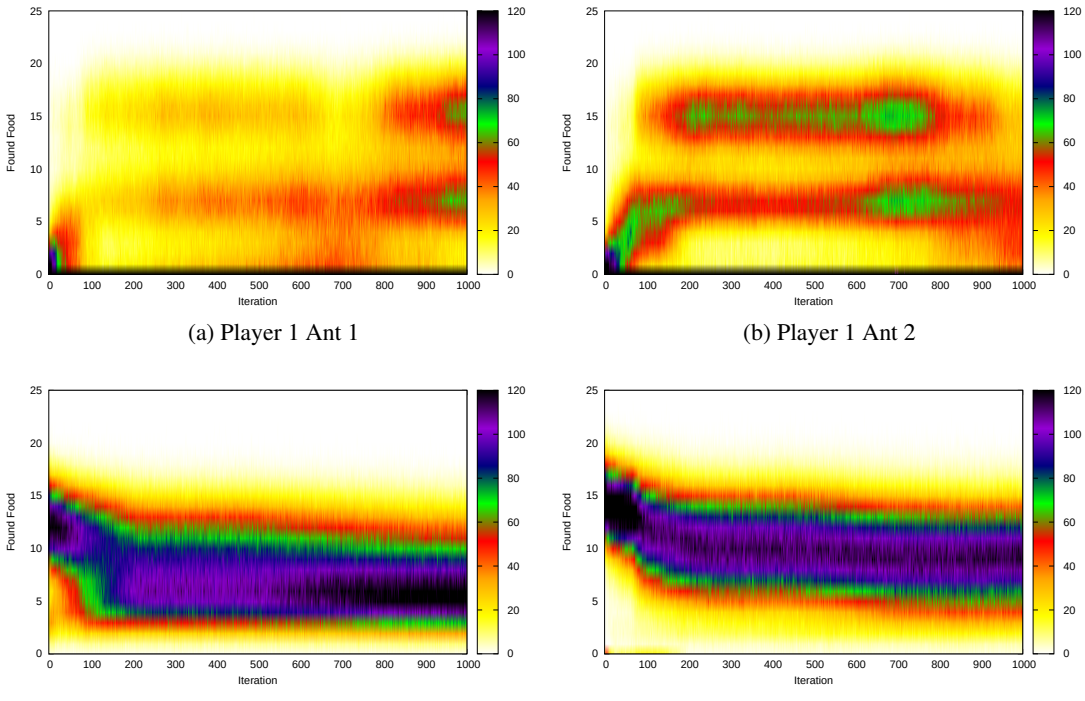



Figure 7.27: Distribution of found food of the *Individuals* playing 2-AntWars against a Hunter player.




(a) Player 1 Ant 1

(b) Player 1 Ant 2

(c) Player 2 Ant 1

(d) Player 2 Ant 2

Figure 7.28: Distribution of average found food per ant and game.

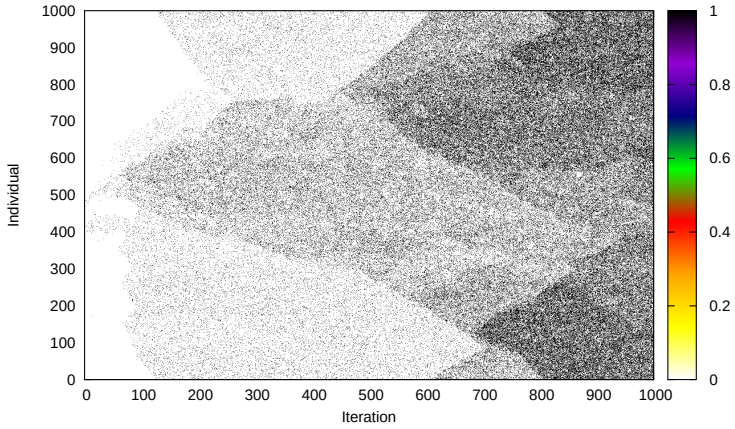
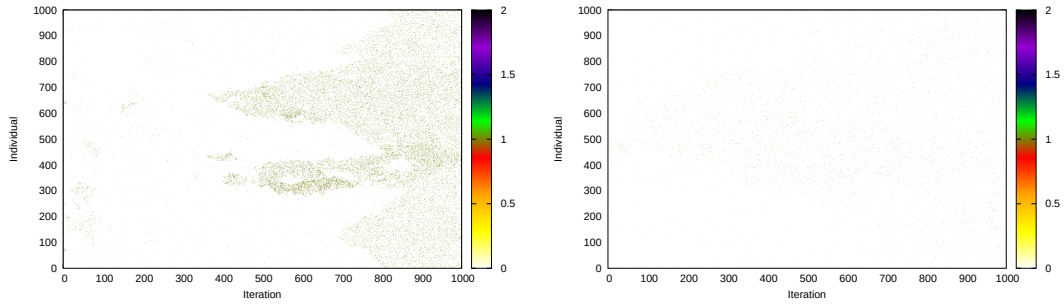



Figure 7.29: The developed ants winning against the Hunter strategy.




(a) Player 1

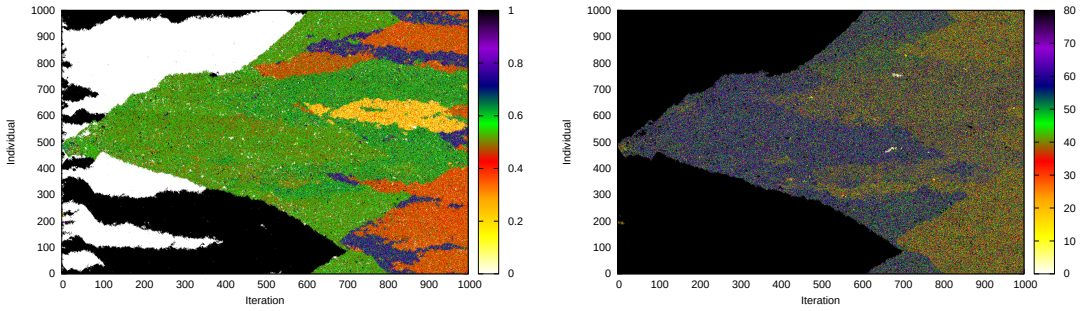
(b) Player 2

Figure 7.30: Number of times a player supported a battle with his second ant during a match.

Figure 7.29 gives an interesting view on the result of the matches played by the population. Even though it took 700 generations for the best *Individual* to start consistently beating the hunter strategy, single matches were won much earlier.

The most surprising result of this run is shown in figure 7.30a. There are actually multiple species contained in A that use the battle support rule to win fights against player 2. It does not happen often (once of a maximum of ten during a match), but it does happen and it is not only coincidence, because figure 7.30b shows how often player 2 supported his battles even though he was not programmed to do so; it rarely happened.

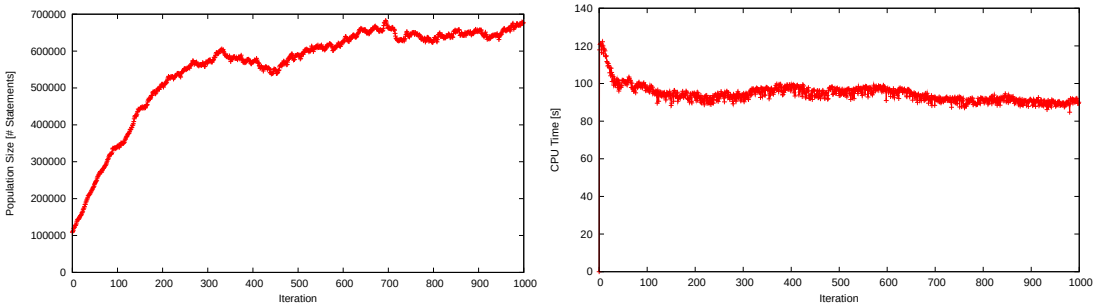
Figures 7.31a on the following page and 7.31b on the next page show that the population is still very diverse from generations 700 to 1000 regarding the movement probabilities of the ants and when both ants are used for the first time. Figure 7.31b on the following page indicates that moving both ants earlier than after 40 moves is spreading inside the population. This could be an effect of the lure strategy, where the luring ant does not use its full 40 moves before it gets caught.




(a) Probability of choosing ant 1 for movement

(b) Average time until both ants leave their starting positions

Figure 7.31: Indicators of the decision method used by player 1 against the Hunter strategy.




(a) (b)

Figure 7.32: (a) Total number of *Statements* contained in the population and (b) the time needed to evaluate them.

The size development of the population (figure 7.32a) and the time needed to evaluate on generation (figure 7.32b) offer no surprises. The total number of *Statements* contained in the population rises fast towards 600000 but does not effect the execution time. The execution time is on a higher level than in the run against the Scorched Earth strategy, at least partly because the Hunter strategy is computationally more expensive.

CHAPTER 8

# Strategies Version 2

For this chapter, the implementation of the strategies from the previous chapter was improved. Section 8.1 contains the documentation of the run against the second implementation of the Greedy strategy, section 8.2 on page 79 presents the improved Scorched Earth strategy and section 8.3 on page 84 the Hunter strategy.

## 8.1 Greedy

### Implementation

The movement *Function* (used for both movement1 and movement2) of the second implementation of the Greedy strategy was, compared to the first implementation, greatly simplified. It only uses the functionality provided by the NearestFood *Statement*. When it is executed, it calculates the Moves towards the nearest, second nearest and third nearest food positions and returns the intersection of those Moves if it is not empty, or the intersection of the nearest and second nearest (if not empty) or just the Moves towards the nearest food positions.

The decision *Function* was modelled after the results of the runs against the first implementations. In all cases, the developed players used the ants sequentially which seemed to be advantageous, so the decision *Function* uses ant 1 as long as it is movable and ant 2 otherwise.

The predict *Functions* are still of no concern to the Greedy strategy and simply return the previous prediction.

The belief *Function* behaves as it did in the original implementations, i.e. believe that food is on positions that were not previously seen and that food seen previously stays there (and is not eaten by enemy ants). The belief in food at unseen positions is also necessary to keep exploring the playing field, because otherwise the movement *Functions* would not return Moves if the ants have not seen food, as the implementation of the NearestFood *Statement* is based on food belief.

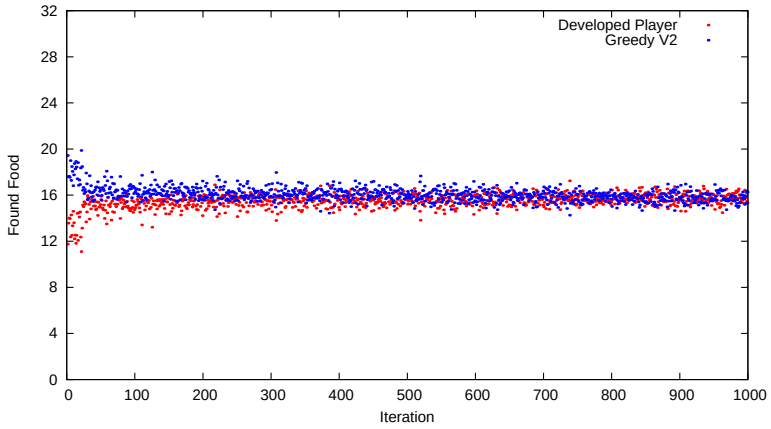
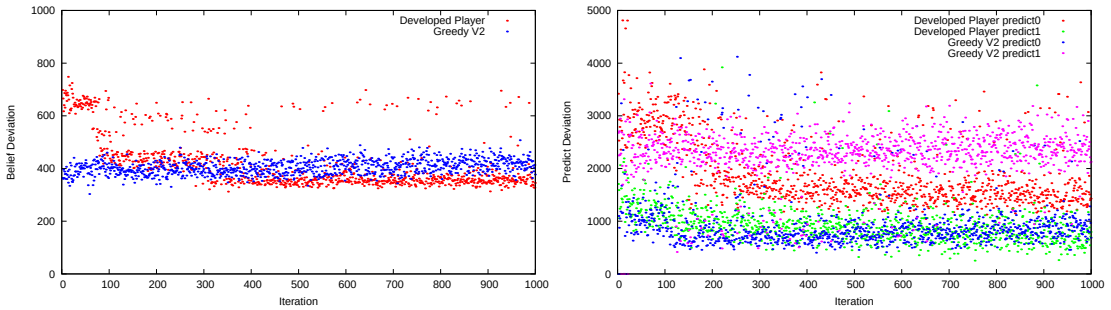



Figure 8.1: Average number of found pieces of food of the best *Individual* per game.




(a) Belief deviation

(b) Predict deviation

Figure 8.2: Average belief and predict deviations of the best *Individual* and its opponent per game.

## Results

Figure 8.1 shows that GPS was able to develop players that are on par with the Greedy strategy, but it was not able to beat it with both averaging to about 16 eaten food pieces per game.

The developed belief *Function* did beat the belief *Function* of the Greedy player as can be seen in figure 8.2a. It behaves in the following way: Belief in food at unseen fields and in seen fields with food is 0.985 and does not change over time, with the exception that the belief in food at unseen positions jumps to zero after 24 moves. Note that the best *Individuals* per generation sometimes use a bad belief *Function*, which indicates that in this case the quality of the belief *Function* is not all that important for a high performing 2-AntWars player.

The predict deviation of the Greedy player (figure 8.2b) shows that the developed players again first use up the moves of ant 2 before ant 1 is moved, so always predicting the last seen position is a bad choice for the prediction of ant 2. The developed player does better with the

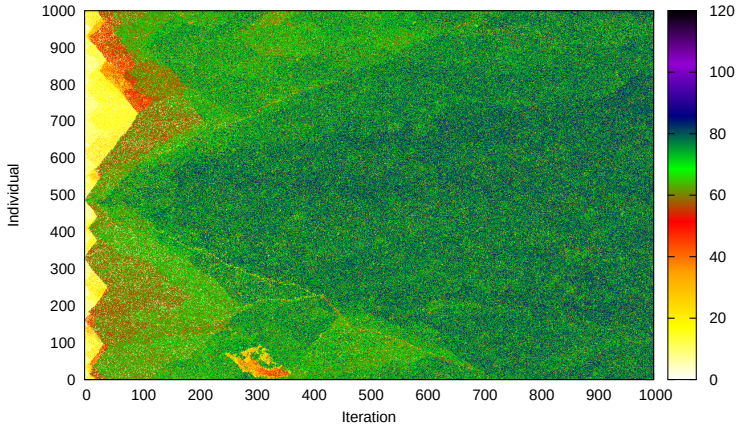



Figure 8.3: Found food of the *Individuals* playing 2-AntWars against a Greedy player.

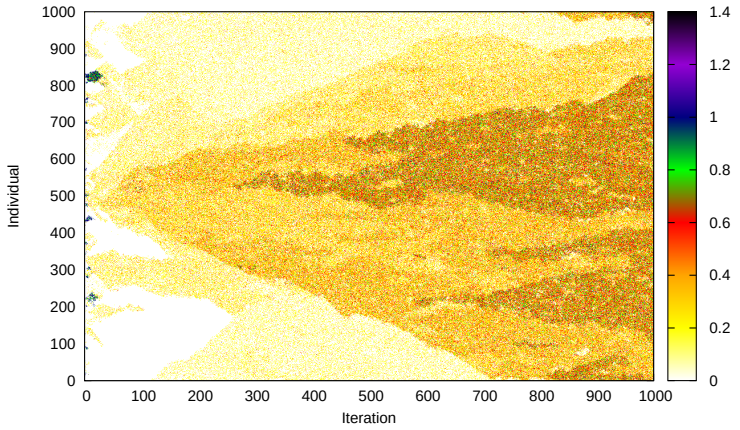



Figure 8.4: Average number of begun battles per game.

prediction of ant 1 of the Greedy player by moving the prediction to $(12, 5)$. The developed player keeps the prediction of ant 2 on the starting position.

Figure 8.3 gives the overview of the fitness development of the population. This time around it only shows one species slowly taking over the whole population. Other graphs are needed to better distinguish between species inside the population.

One graph helpful in that respect is figure 8.4. It shows the average number of begun battles per game of ant 2 of the developed player and one can see that the feature of aggressive use of ant 2 spreads through the population, every increase in aggressiveness giving an evolutionary advantage. At the end of the run the *Individuals* with the most aggressive use of ant 2 take up more than half of the population.

Figure 8.5 on the next page shows two principal decision strategies present in the population. The more aggressive species moves ant 2 more often than ant 1 while the less aggressive species

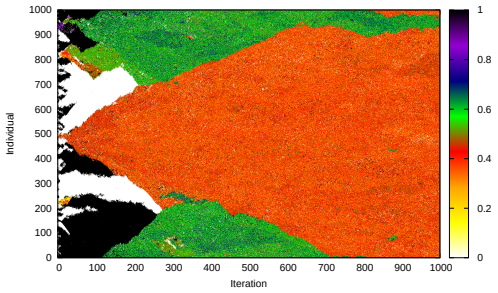



Figure 8.5: Probability that player 1 chooses ant 1 for movement.

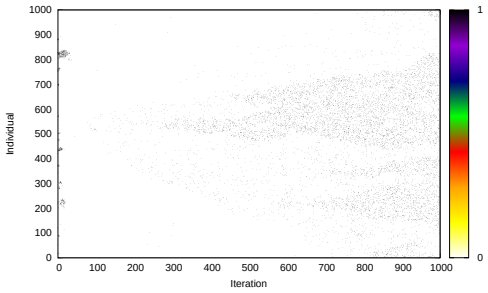
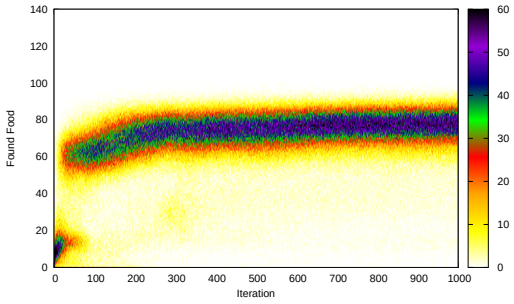


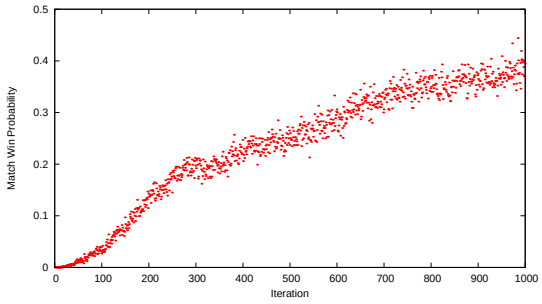

Figure 8.6: Player 2 not finding food with both ants.

uses the ants with equal probability. Furthermore, the usage of both ants originated from four players in the original population, where one died out, two merged into the less aggressive species and one was the ancestor of the more aggressive species which slowly drove the less aggressive species to near extinction.

Figure 8.6 shows which Greedy players were able to use both ants to find food. Normally, every Greedy player should use both ants to collect food, but some of them could not. The reason for this is that the developed players neutralized ant 2 of the greedy player before it could begin collecting food. The Greedy player fails to use both ants in the early generations because players that move one ant constantly east are better than the most randomly generated players and player 1 moving ant 2 constantly east will neutralize the waiting ant 2 of the greedy player.



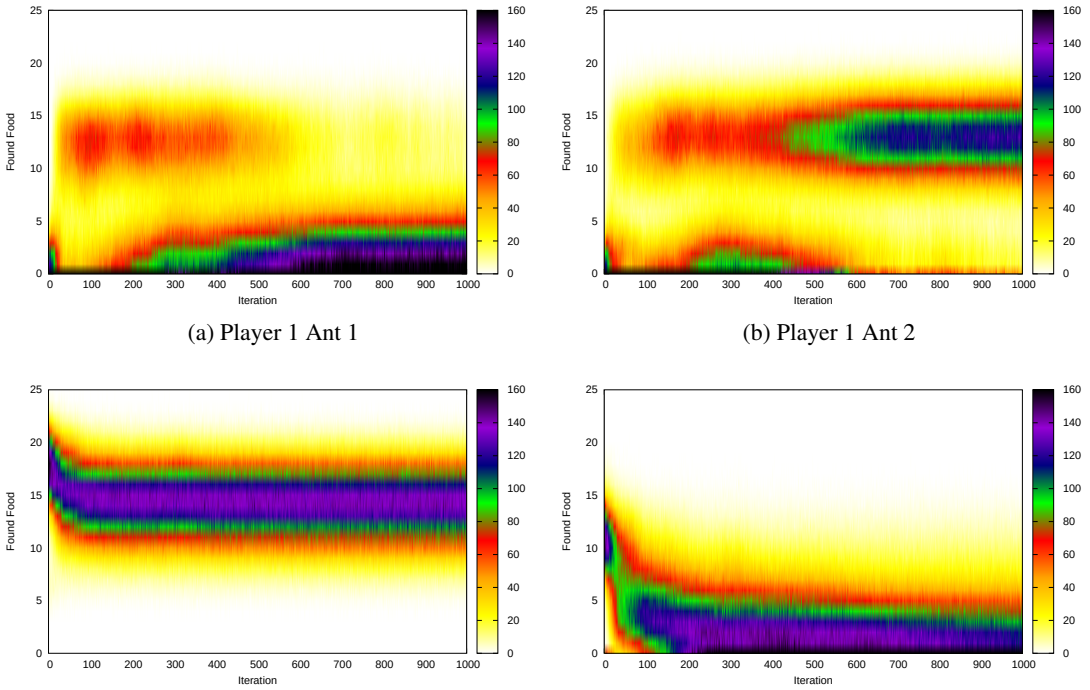

(a) Distribution of found food




(b) Probability of winning a match

Figure 8.7: (a) Distribution of found food and (b) probability of winning a match of an *Individual* in the population against a Greedy player.

The distribution of the fitness values of the *Individuals* in the population (figure 8.7a) shows the population slowly increasing its fitness without the formation of secondary fitness levels as has been observed before. Figure 8.7b shows the corresponding probability of a random *Individual* in the population winning against a Greedy player which at the end of the run was $0.4$. This was also hinted at by the fitness distribution, as the center of the distribution did not




(a) Player 1 Ant 1

(b) Player 1 Ant 2

(c) Player 2 Ant 1

(d) Player 2 Ant 2

Figure 8.8: Distribution of average found food per ant and game against a Greedy player.

reach 80 pieces of food per match (and obviously more *Individuals* lie below the center of the distribution than above).

The histogram of the performance of each individual ant per game confirms previously made assertions about the behaviour of each ant. Player 1 predominantly uses ant 2 to collect food but is not as efficient as ant 1 of player 2. Ant 2 of the greedy player gets more and more suppressed until it hardly finds any food at all.

The time to evaluate each generation showed the same behaviour as was already seen before, the early generations took 140 seconds which decreased to 100 seconds after 200 generations. The size development of the population also offered no surprises, it rose fast from 100000 *Statements* in the beginning to 600000 after 400 generations.

Due to the unsatisfying performance of the developed player, a run with 2000 generations and 2000 *Individuals* was attempted, the results, however, were worse than the first run, as is depicted in figure 8.9 on the next page. It took GPS the whole 2000 generations to generate players on par with the Greedy strategy. Interestingly, the same development of aggressiveness could not be observed.

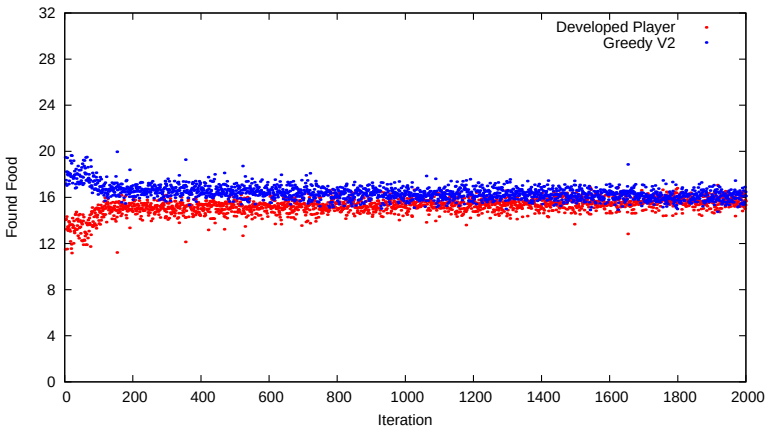



Figure 8.9: Average number of found pieces of food of the best *Individual* per game.

## 8.2 Scorched Earth

### Implementation

The Scorched Earth strategy had the most need for change since the first implementation was beaten by a wide margin. The reason for this was the rigidly defined movement of the ants which was subsequently exploited by the evolved players. The first change was to make the movement *Function* to use one ant as long as it is movable and then use the second ant. The movement *Function* for ant 2 is the same as the movement *Function* of the Greedy implementation in the previous section, so the idea of Scorched Earth is incorporated solely into the movement *Function* of ant 1. Since rigidity proved to be the downfall of the first implementation, ant 1's movement direction is not strictly predetermined in this implementation but rather guided by waypoints. During the first 12 moves, ant 1 moves towards $(10, 1)$ which brings it to the middle of the playing field. The next 12 moves are towards $(10, 11)$, crossing the playing field southward and after that for the next 12 moves ant 1 moves towards $(18, 11)$, finishing the half-circle of scorched earth. During those movements, the exact move is computed by calculating the moves towards the current waypoint, the moves towards the nearest food pieces and returning the intersection of them if it is not empty or the moves towards the waypoint otherwise. When the earth is sufficiently scorched (after 36 moves), ant 1 spends its remaining moves by collecting food in a greedy fashion, like the second Greedy implementation.

The decision *Function* moves ant 1 until it cannot move any more and then ant 2. It is crucial for the scorched earth strategy to keep the enemy's ants out of the own half of the playing field before the food has been collected, so the decision *Function* overrides the movements suggested by the movement *Functions* if one of the ants can attack an ant of the enemy in one move.

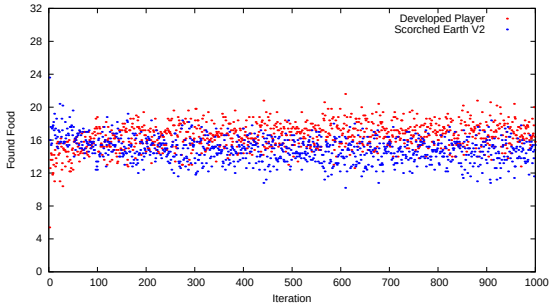
The belief *Function* is the same that the second Greedy implementation used, and the predict *Functions* are the same as the first implementation of the Hunter strategy.

### Results

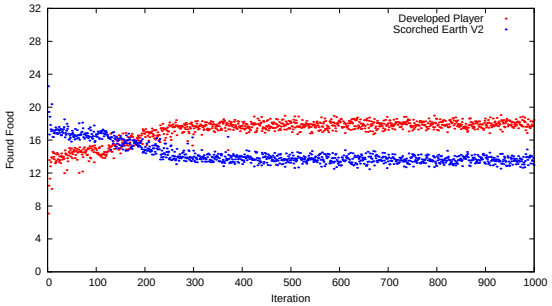
Due to a configuration error, the run against the second implementation of the Scorched Earth strategy used the value of 5 for the GamesPerMatchBest setting instead of 50. It was repeated with the correct settings yielding essentially the same result, but the run with the erroneous setting will be presented here because it showed the interaction between the *Functions* of the 2-AntWars player in a clarity not seen previously.

Figure 8.10 on the following page shows the fitness development of the best *Individual* per generation for both configurations. In both runs, GPS was able to develop players capable of beating the Scorched Earth strategy (but with less margin than the first implementation showing that the second implementation indeed was an improvement). Figure 8.10a on the next page also serves the purpose of showing with what kinds of random fitness fluctuations GPS has to deal when trying to evolve capable 2-AntWars players.

Because of the configuration error, the data on the belief and predict deviation shown in figure 8.11 on the following page is fuzzier than usual which hampers the interpretation process. The belief deviation shows that the belief *Functions* of the evolved players beat the belief *Func-*

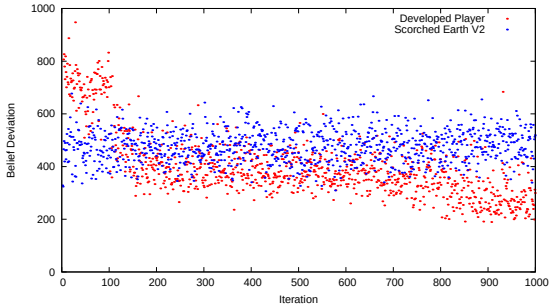



(a) Erroneous 5 GamesPerMatchBest

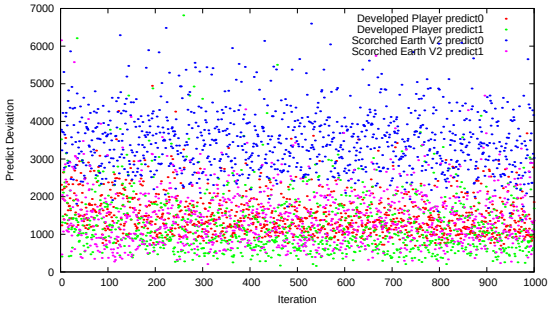



(b) Correct 50 GamesPerMatchBest

Figure 8.10: Average number of found pieces of food of the best *Individual* per game, (a) with only five games between the best *Individual* and the Scorched Earth strategy and (b) the correct amount of games between them.




(a) Belief deviation




(b) Predict deviation

Figure 8.11: Average belief and predict deviations of the best *Individual* and its opponent per game.

*tion* of the Scorched Earth player. Note the improvement of the evolved belief *Function* around generation 800, this will come into play later.

The belief *Function* that the best *Individual* of the last generation uses showed the most complex behaviour of all the belief *Functions* discussed until now. It believes in seen food constantly with a value of one. Initially, it also believes in unseen food with the same value. After 20 moves, the belief in food at unseen positions only remains in the seven rows on the top of the playing field and it shrinks to two rows after 40 moves. This is a result of the fact that ant 1 of the Scorched Earth player misses more food at the begin of his arc than at the end (because at the end it greedily collects food in the area at the bottom of the playing field). After ant 1 has exhausted its moves, ant 2 collects food around its starting position in the center area of the playing field, leaving only the top most rows unharvested (with a certain probability, not always). The downside of this is that food in the south-western corner of the playing field goes

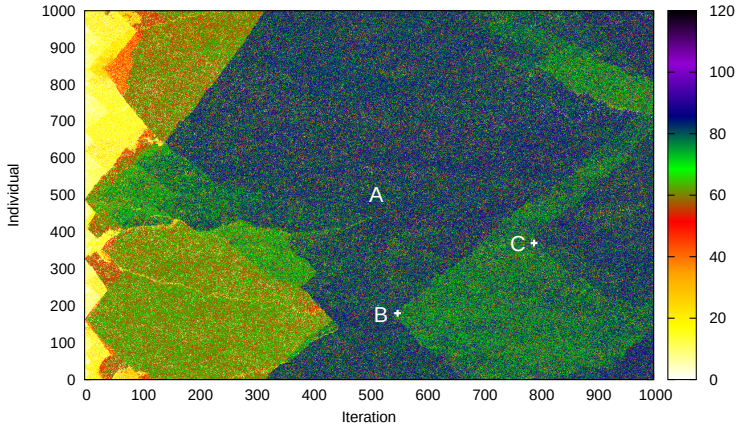



Figure 8.12: Found food of the *Individuals* playing 2-AntWars against a Scorched Earth player.

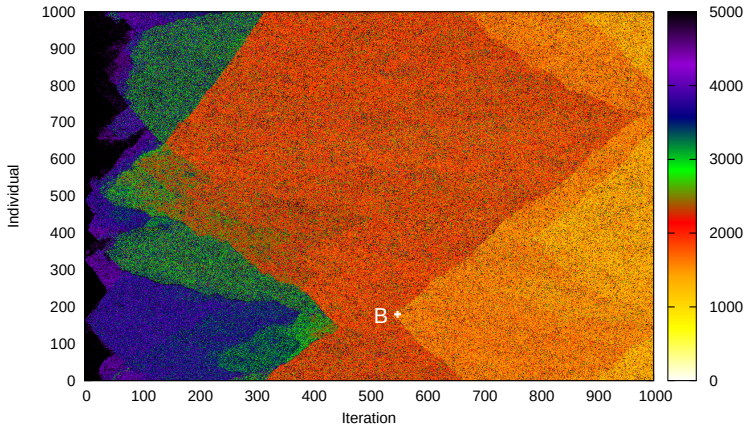
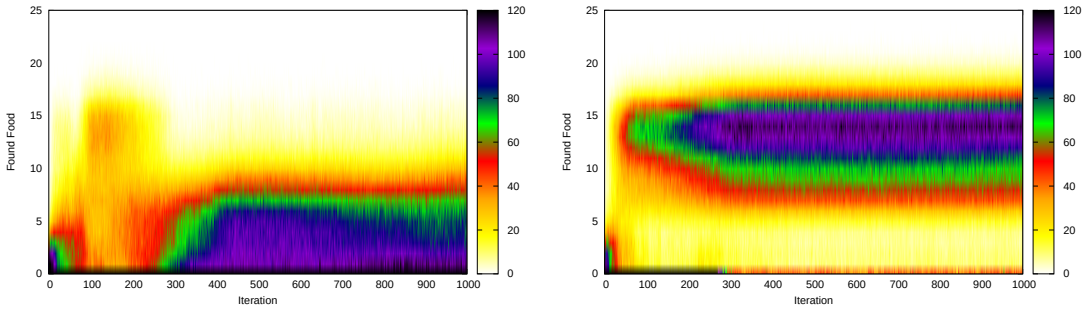



Figure 8.13: Belief deviation of the *Individuals* playing 2-AntWars against a Scorched Earth player.

uneaten by the evolved players because they evolved in a way that lets them be guided by the food belief but not by unseen fields and it is too far away for ant 2 of the Scorched Earth player.

The predict *Functions* did not evolve this kind of behavioural complexity. The prediction of ant 1 moves to $(12, 6)$ and then jumps around in a one move neighborhood around it which is a good approximation of the later positions of ant 1. The prediction of ant 2 stays at the position one move to the west of ant 2's starting position.

The fitness development of the population depicted in figure 8.12 shows fairly standard behaviour during the first 500 generations, with species A taking over the population. A obviously consists of multiple subspecies but this is not the focus of concern right now. What's important is spawn point B, where it seems that a species with inferior fitness emerged that spread throughout the whole population. Only some time later, another species reestablished the fitness




(a) Ant 1 (b) Ant 2

Figure 8.14: Distribution of average found food per ant and game against a Scorched Earth player.

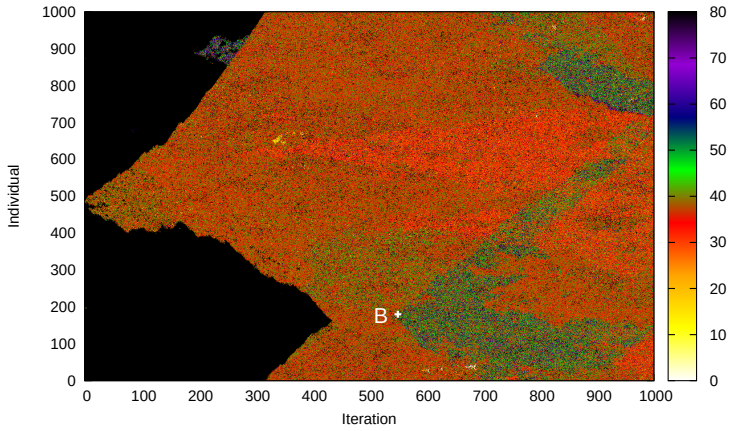
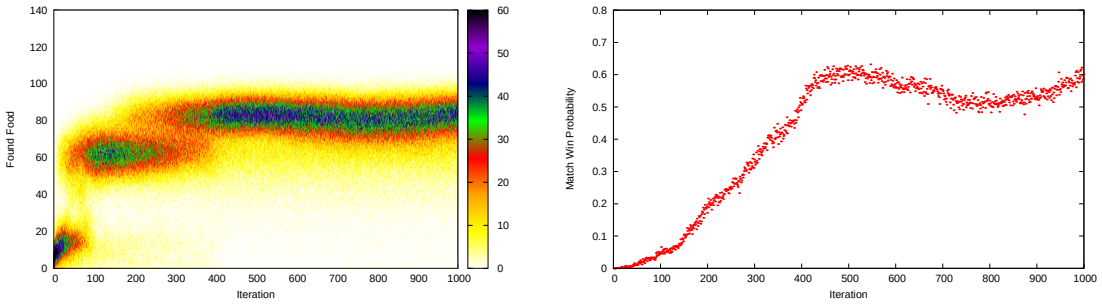



Figure 8.15: Average time when both ants of the *Individuals* playing 2-AntWars leave their starting positions against a Scorched Earth player.

level of A at spawn point C. Here figure 8.11a on page 80 comes to mind as it showed that the best *Individuals* of the population began to use improved belief *Functions* around generation 800 which is also the same time C emerges. This indicates that an improvement in the belief *Function* might be responsible for the inferior species B.

Figure 8.13 on the previous page shows the development of the belief deviation of the population and indeed, spawnpoint B marks the rise of a new and improved belief *Function*. It behaved differently than its predecessor and damaged the performance of the evolved 2-AntWars players, but what exactly changed? The performance of ant 1 (figure 8.14a) was influenced, while the one of ant 2 (figure 8.14b) stayed constant. Figure 8.15 shows the average time when both ants of the developed player have left the starting positions. Note that B is visible, but not as usual for this graph where new species show up by moving both ants before the player moved 40 times. Instead, the ants of B take more than 40 moves on average to leave their starting positions.




(a) Distribution of found food

(b) Probability of winning a match

Figure 8.16: (a) Distribution of found food and (b) probability of winning a match against a Scorched Earth player.

This can happen in two ways. The first way is that ant 1 (which is the ant that moves second) might wait some moves after ant 2 ran out of moves before it starts moving, which seems unlikely. The second possibility is that in some cases ant 1 moves immediately after ant 2 ran out of moves, but in some cases does not move at all. This can happen when the belief *Function* does not believe in food at unseen positions, the player has not seen food that he has not eaten and does not explore unseen fields. That seems to be the reason for the less fit individuals.

Note also how this figure shows the existence of additional species that move both of their ants earlier, confirming the previous assertion that A consists of multiple species.

The fitness distribution of the *Individuals* in the population (figure 8.16a) shows the damage done by the emergence of the improved belief *Function* at B. It is even more visible in figure 8.16b, with the probability of winning a match dipping from $0.6$ to $0.5$. The run with the correct settings did not have this dip and the probability of winning converged to $0.7$. Also note that the whole population reaches a win probability of $0.5$ after 400 generations while the best *Individuals* start beating the Scorched Earth player after 100 generations.

In this run, the population size rose only to 500000 *Statements* instead of the 600000 *Statements* seen previously. The time needed to evaluate a generation also diverged from the standard behaviour, it decreased from 120 to 100 seconds during the first 500 generations but then increased, reaching 120 again at the end of the run.

## 8.3 Hunter

### Implementation

It was hard for GPS to evolve players that beat the first implementation of the Hunter strategy, so the improved implementation of the Hunter kept the separation of concern between hunting ant and gathering ant and was made even more aggressive and focused on one target.

The base of movement, the implementation of the movement *Functions*, reused the movement *Function* of the second implementation of the Greedy player so it would benefit from its improvements.

The decision *Function* was still the centerpiece of the hunting behaviour, because it has access to the position predictions of the enemy's ants. The first step of the decision *Function* is checking if one of the ants is one move away from one of the ants of the enemy, if so, the ant will attack. Otherwise the Moves towards both of the enemy's ants are calculated. If the intersection is not empty, the intersection is used as the base for the further decision process. In the case that the intersection is empty, the Moves towards the nearest (or the union of the Moves if the distance is the same) is used as the base. If the intersection of the base with the Moves-suggestion of movement1 is not empty, the result is used to override the Moves-suggestion for ant 1, otherwise the base overrides the Moves-suggestion and the ID of ant 1 is returned. This calculation is only done if ant 1 is actually movable, in the case that it is not, the ID of ant 2 is returned.

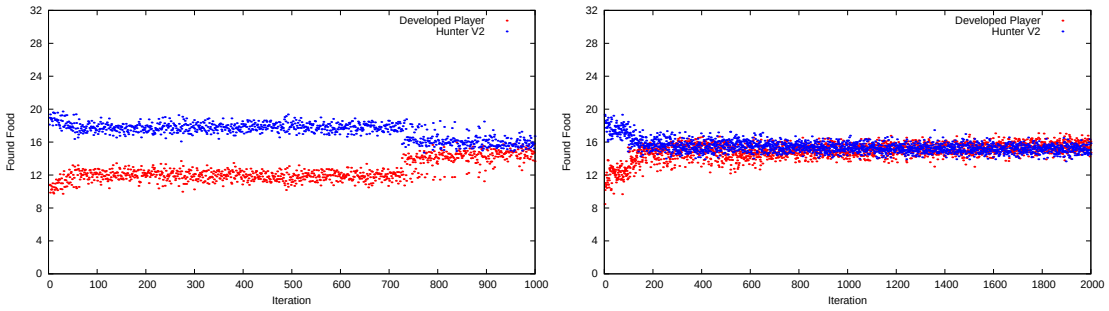
The belief and predict *Functions* work exactly like they did in the first implementation of the Hunter strategy.

### Results

The run against the second implementation of the Hunter strategy was not successful as is shown by figure 8.17a on the next page. Around generation 720 the evolved players were finally able to catch up but not competent enough to actually beat the Hunter strategy. The problem was that the evolved players did not use both ants effectively until this jump in fitness around generation 720, so the run was repeated with a population size of 2000 and 2000 generations. The results of this run are presented here.

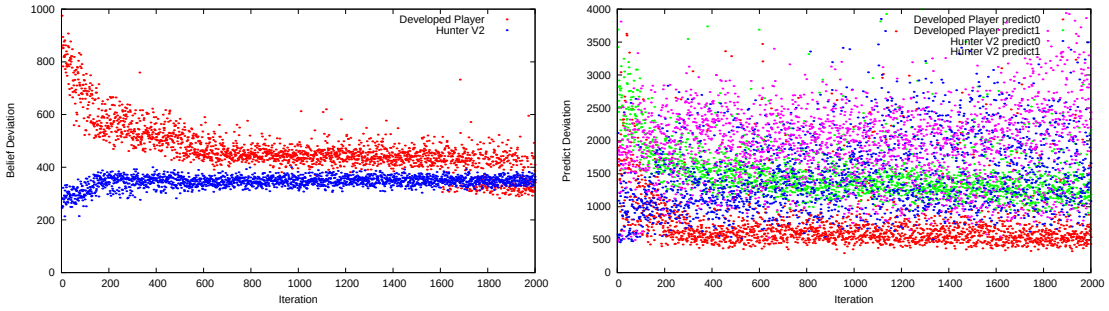
Figure 8.17b on the facing page shows the result of the run with changed settings against the Hunter strategy. In this run, the best *Individuals* were on par with the Hunter player after 1000, and able to beat it after 2000 generations.

The belief *Functions* of the developed player were not able to beat the belief *Function* of the Hunter player until generation 1600 (figure 8.18a on the next page). Beginning with that generation a transition to a better performing belief *Function* can be witnessed, so even though the run had already lasted for 1600 generations, there were still improvements found. The belief *Function* of the best *Individual* of the last generation always believes in food that was once seen and in food at unseen positions until the 31-st move, when it only believes in food at unseen positions at the two northern-most rows of the playing field. This is a good approximation of the behaviour of the Hunter, because the hunting ant will go most likely straight west and the gathering ant starts in the southern half of the playing field.




(a) Normal settings (b) Large population

Figure 8.17: Average number of found pieces of food of the best *Individual* per game, (a) with the normal settings and (b) with 2000 iterations and 2000 *Individuals*.




(a) Belief deviation (b) Predict deviation

Figure 8.18: Average belief and predict deviations of the best *Individual* and its opponent per game.

The predict *Functions* (figure 8.18b) showed much more prediction precision than the ones of the Hunter and was doing better with the prediction of the hunting ant because its general direction is constant. The prediction *Function* for ant 1 of the best *Individual* in the last generation moved the prediction one step west with each move and stopped at $(5, 5)$. The prediction *Function* for ant 2 kept the prediction at the starting position until the 34-th move, when it was set to $(14, 8)$. Note that the prediction *Function* of the Hunter worked worse than in the run with the previous implementation, even though they are identical which indicates that the evolved players move their ants in a way very contrary to the expectations of the predict *Functions* of the Hunter. This seems logical as the hunting ant bases its movement decisions on these predictions and the evolved player's ants do not want to be found and attacked.

The development of the found food per match depicted in figure 8.19 on the next page is rather unremarkable. After 700 generations the species in the population seem to have converged, some variation in fitness level can be seen but not enough to separate different species.

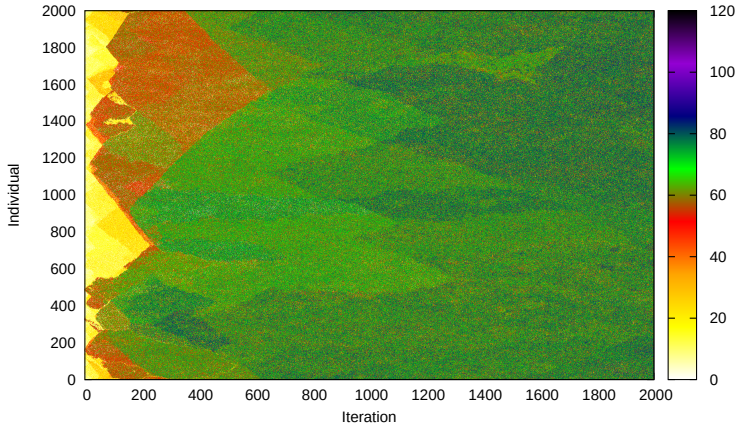
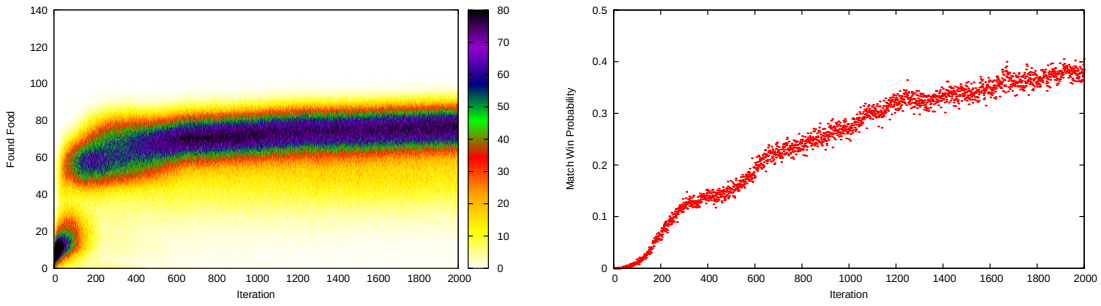



Figure 8.19: Found food of the *Individuals* playing 2-AntWars against a Hunter player.




(a) Distribution of found food

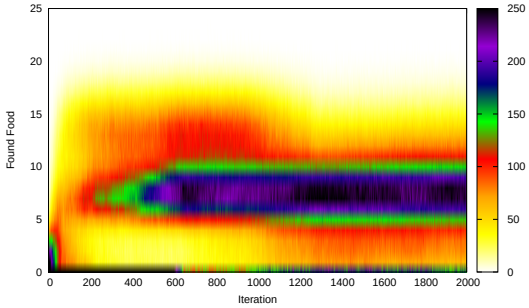
(b) Probability of winning a match

Figure 8.20: (a) Distribution of found food and (b) probability of winning a match against a Hunter player.

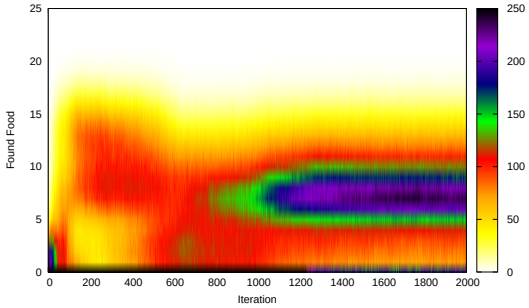
Another insight into the fitness development can be gained by studying figures 8.20a and 8.20b. There is only one dominant level of fitness present in the population as it slowly improves. The probability of winning a game improves by 10 percent between generations 1000 and 2000, a fact which is completely invisible in figure 8.19.

Figure 8.21 on the next page shows how the ants of the players contribute to the final fitness. It can be seen immediately that the division of labour between the ants of player 1 (the developed player) is completely different when compared to the previous runs. In this run, both ants contributed equally to the total amount of found food while in previous runs one ant was dominant, moved first and collected most of the food while the second ant searched for the missed pieces of food. Also interesting is the fact that the hunting ant (ant 1 of player 2) collects more food than the gathering ant, simply because it moves first and has more opportunity to do so.

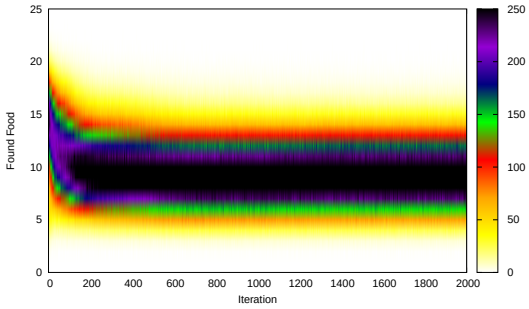
Now the obvious question is what the developed player was doing with his ants to reach this equal distribution of found food between the ants. The next figures shed light on this subject.



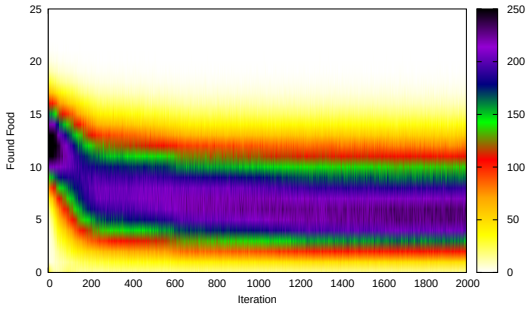

(a) Player 1 Ant 1




(b) Player 1 Ant 2




(c) Player 2 Ant 1




(d) Player 2 Ant 2

Figure 8.21: Distribution of average found food per ant and game against a Hunter player.

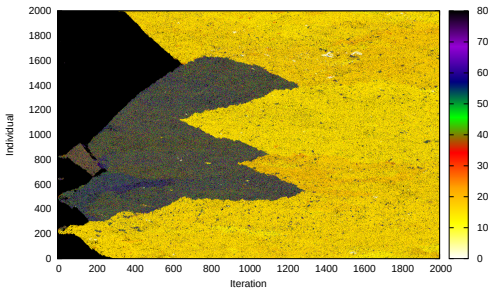



Figure 8.22: Average time when both ants of the *Individuals* playing 2-AntWars leave their starting positions against a Hunter player.

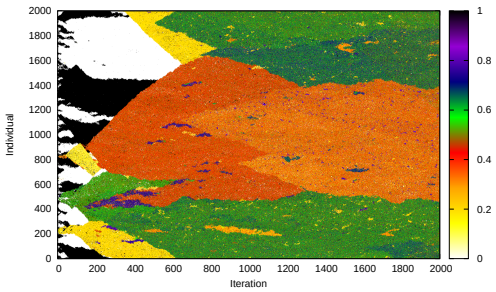



Figure 8.23: Probability of choosing ant 1 for movement.

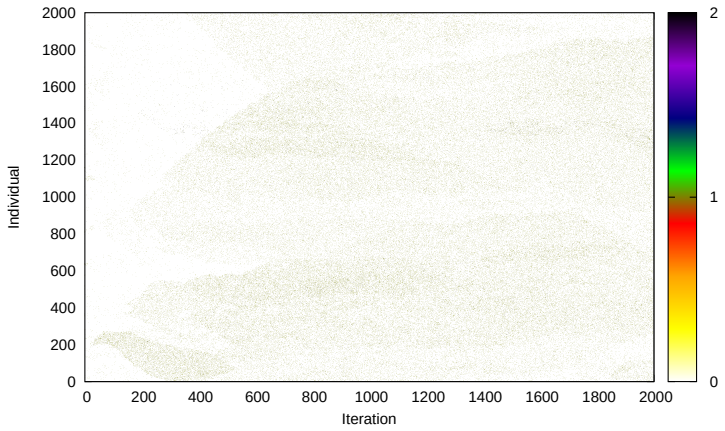
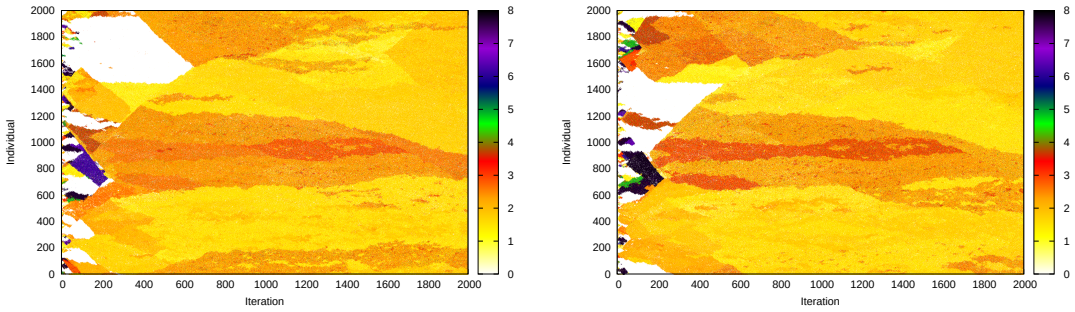



Figure 8.24: Number of times a player supported a battle with his second ant during a match.

Figure 8.22 on the preceding page shows that the ability to move both ants emerged at four positions in the population, one dying out almost immediately. Two of them used the traditional movement method of using up the moves of one ant before moving the other while one moved both ants earlier. A lot of times in the previous runs, the dominant species at the end of the run contained subspecies that moved both ants earlier. In this case, this property spread through the whole population. Even the two species originally moving one ant after another developed it. However, the population did not converge as figure 8.23 on the previous page shows. There are still two species present, one moving predominantly ant 1 and one predominantly moving ant 2. This does still not fully explain what the developed player does to keep his ants from being neutralized, so the games of the best *Individuals* were analyzed. It was seen that the evolved players first move one ant and collect some food. Then, when the hunting ant is about five to seven fields away from the starting positions of the ants, the other ant is moved so that the hunting ant does not find it. This ant then continues to search for food and if it runs out of moves, the ant that was moved first continues. With this procedure, the evolved players can keep searching food long enough to win games and matches.

The developed players also had a battle support component, as figure 8.24 shows. Figure 8.25 on the next page is proof that the population still contains a lot of diversity regarding the movement *Functions*. It is noticeable that the movement *Functions* of the ants mirror each other as a result of the crossover operation.

In this run, the population size rose to 1200000 *Statements* in the first 600 generations. The time needed to evaluate a generation became shorter, from 250 seconds in the beginning of the run to 170 seconds in the end.




(a) Ant 1

(b) Ant 2

Figure 8.25: Number of moves to choose randomly from after the decision *Function* of the developed player was called.

CHAPTER 9

# Coevolutionary Runs

This chapter details four coevolutionary runs, that means that both player 1 and player 2 were subject to evolution. Section 9.1 reports on a coevolutionary run with the standard settings, while section 9.2 on page 97 presents the results of a run using asymmetric evaluation with a $\Delta_{\text{eval}}$ of 1. As a result, the *Individuals* in the host population are evaluated by playing three matches against *Individuals* from the parasite population while only the result of one match is used to assign a score to the parasite *Individuals*.

The following two sections document runs where the aim was to explore the long term behaviour of coevolution. Both runs used a population of 500 *Individuals* and more generous size limits for the *Functions* of the 2-AntWars players (belief 200, predict 200, decision 300 and movement 400 *Statements*). $\Delta_{\text{host}}$ and $\Delta_{\text{parasite}}$ were set to 2 instead of 3 because of the smaller population. For every other setting (except the number of generations) the standard values were used. Section 9.3 on page 101 presents a run with these settings over the course of 10000 generations. The run that delivered the content of section 9.4 on page 104 used asymmetric evaluation with a $\Delta_{\text{eval}}$ of 1. Originally it was planned to let GPS evolve players for 10000 generations but due to the exorbitant time requirements the run was aborted after two weeks and 6500 generations.

## 9.1 Run with Standard Settings

Figure 9.1 on the following page shows the average number of pieces of food the best *Individual* of the host population (playing as player 1) and the best *Individual* of the parasite population (playing as player 2) found on average during a game. It took player 2 200 generations to reach the level of player 1 and from that generation on the performance of the players was about the same. It seems, however, that player 1 has a slight advantage as there are more outliers with better performance than worse performance.

It has to be mentioned that this type of graph is only of limited significance in coevolutionary runs because the best player 1 is most likely at a different position in the population than the best player 2 so when each of them adapts to their opponents they do not adapt to each other. It is

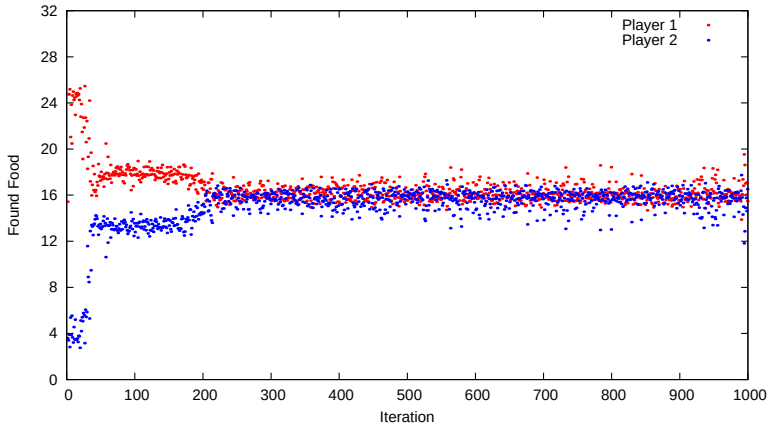
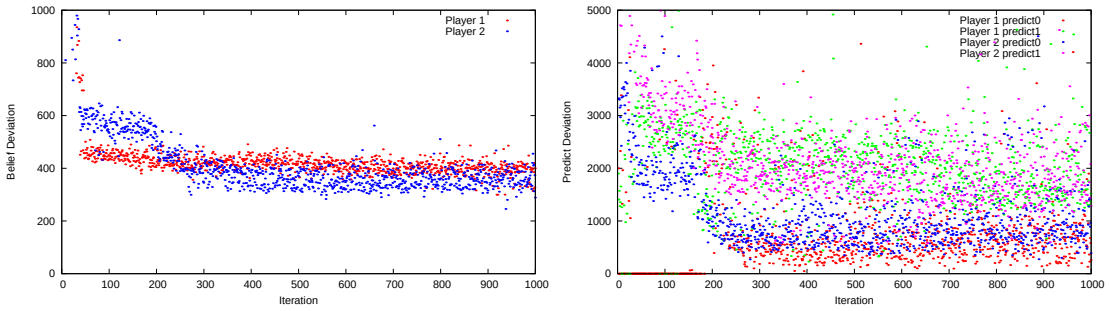



Figure 9.1: Average number of found pieces of food of the best *Individuals* per game.




(a) Belief deviation

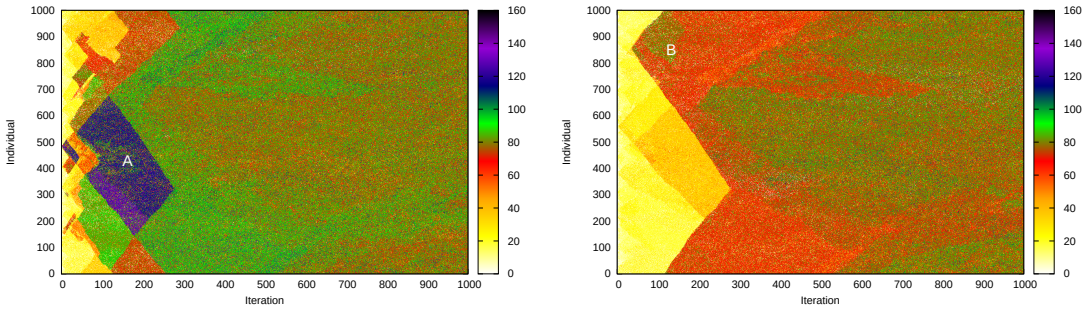
(b) Predict deviation

Figure 9.2: Average belief and predict deviations of the best *Individuals* per game.

possible that the best player 1 beats the best player 2 but player 2 is far superior to its local opponent and has a better fitness than player 1 against his opponent.

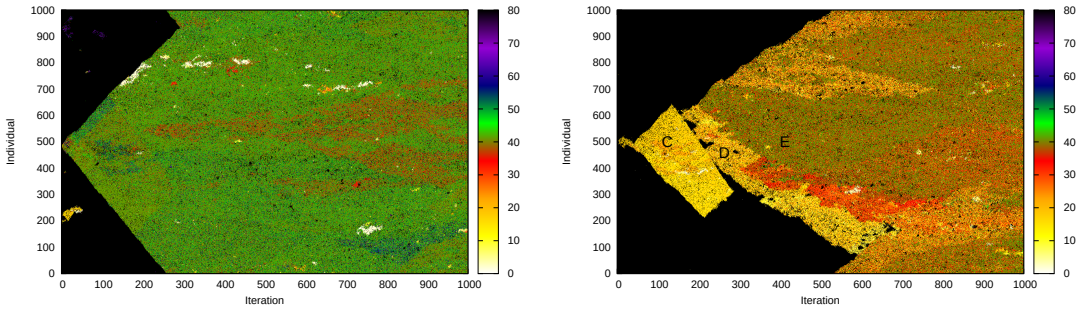
The belief *Function* of the best player 2 (figure 9.2a) is worse than the belief *Function* of the best player 1 during the first 200 generations. Then the belief *Function* of player 2 starts to improve until it is better than that of player 1.

In the last generation, the belief *Function* of the best player 1 showed the same behaviour as the belief *Function* used for the implementation of the strategies in chapters 7 on page 53 and 8 on page 73, i.e. always believe in food at unseen positions and always believe in food that was once seen. The belief *Function* of player 2, however, demonstrated a more sophisticated behaviour. It also believes in food that was once seen and in food at unseen positions, but the latter only until the 41-st move, then the food belief at the six west-most columns of the playing field goes to zero. This may have been instrumental for the victory of the best *Individual* of the parasite population against the best individual of the host population in the last generation, because ant 1 (which is moved after ant 2 runs out of moves) does not waste moves by searching




(a) Player 1 (b) Player 2

Figure 9.3: Found food of the *Individuals* playing 2-AntWars.



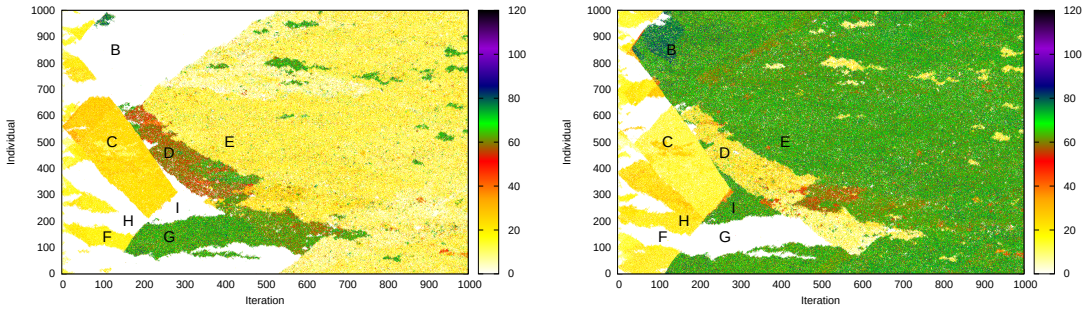

(a) Player 1 (b) Player 2

Figure 9.4: Average time when both ants of the *Individuals* playing 2-AntWars leave their starting positions.

food near the starting position of player 1 but stays near to its own starting position and searches for food that ant 2 might have missed there.

The development of the predict deviation (figure 9.2b on the facing page) shows that ant 1 of both players is easier to predict than ant 2. This indicates that at least the best of both players use ant 2 first and then ant 1. That the one of the predict *Functions* of the best player 1 had a perfect score for the first 180 generations shows that the best player 2 only used one ant during that time. The best players of the last generation showed the following prediction behaviour: Player 1 predicted ant 1 (of player 2) at its starting position, as it did move after ant 2. The predicted position of this ant was moved to $(12, 7)$. Player 2 predicted ant 1 (of player 1) to be at the position one field east to its starting position. Ant 2 was predicted to move to the positions between $(7, 3)$ and $(7, 7)$; the y-coordinate had a random component.

The fitness development depicted in figure 9.3 surprises with one feature that is not visible: local species that clearly beat their opponents. Instead, the whole population seems to follow the trends already indicated by the fitness of the best *Individuals*. At first there is a species in the host population that beats its opponents by a wide margin (labeled A in figure 9.3a) until a fit




(a) Ant 1

(b) Ant 2

Figure 9.5: Found food per ant of player 2.

species in the parasite population emerges that achieves the same score the host population does (labeled B in figure 9.3b on the previous page). Nothing out of the ordinary seems to happen, until one compares the spawn point of B to the location where the use of both ants emerged (labeled C in figure 9.4b on the preceding page). They do not coincide, in contrast to every run that was presented until now. There are three points at which B and C met and combined to create a superior species D that uses both ants. Later the species switched to the standard movement mode of moving one ant until it cannot move any more, labeled E.

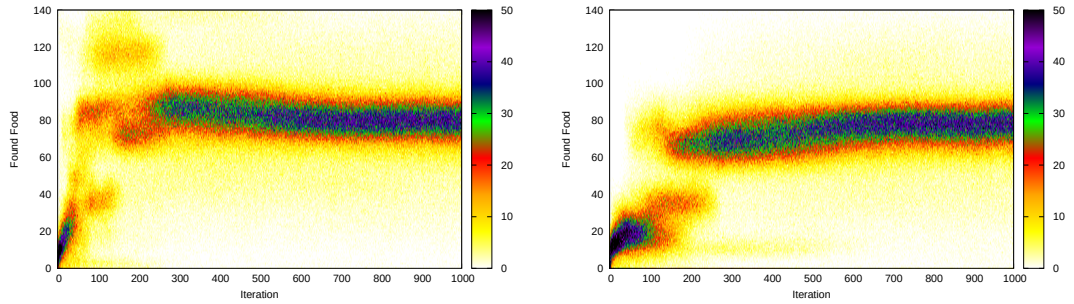
Figure 9.4a on the previous page shows that for player 1 the use of both ants emerged at least at three positions but only at one position it offered enough of a fitness advantage to survive.

Figure 9.5 demonstrates how movement capabilities transfer seamlessly from one ant to another. In this run it is visible in two places.

The first place is where B and C collide to form a new species. In species B, ant 2 did all the food collecting and ant 1 did not move. But when B and C merged, two things happened: the performance of the player improved and ant 1 started to find more food than ant 2. Obviously the capability to move both ants came from C, but C's ants did not find a lot of food and a sudden increase of ant 1's capability at three separate positions and at the same time another species comes into play is very unlikely, so the performance increase of ant 1 has to have come from species B by transferring the important parts of the movement *Function* of ant 2 to the movement *Function* of ant 1 of species C. Ironically, the decision *Function* from C changed soon after the creation of D and ant 2 was again the dominant ant, which also was better than the ants of C.

The second place is where the spread of B wraps around to the lower positions (generation 110). There it meets with a species F that uses only ant 1 and that not very efficiently. Without any delay species F incorporates the important parts of the movement *Function* of B's ant 2, creating species G which enjoys a pronounced increase in fitness. This modification spread through the original F and again the improved movement *Function* switched ants, but this time with some delay. This delay nearly drove H to extinction before it created I.

The development of the fitness of the two players from the histogram perspective (figure 9.6 on the next page) shows the initial supremacy of the host population and the increase in fitness of the parasite population until it cannot be distinguished visually from the host population. In




(a) Player 1 (b) Player 2

Figure 9.6: Fitness distribution.

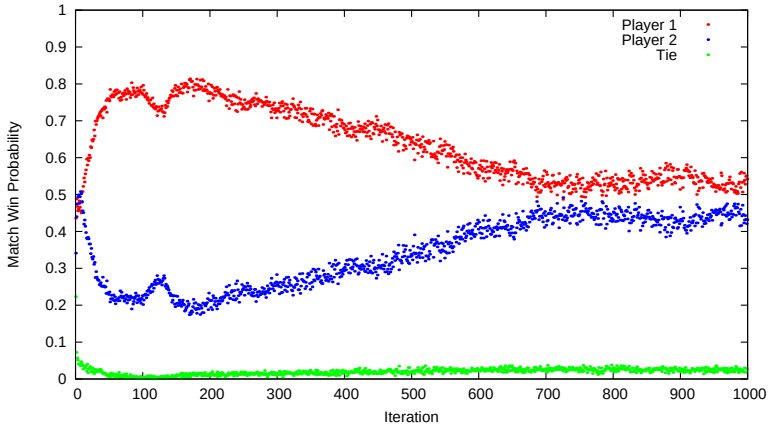


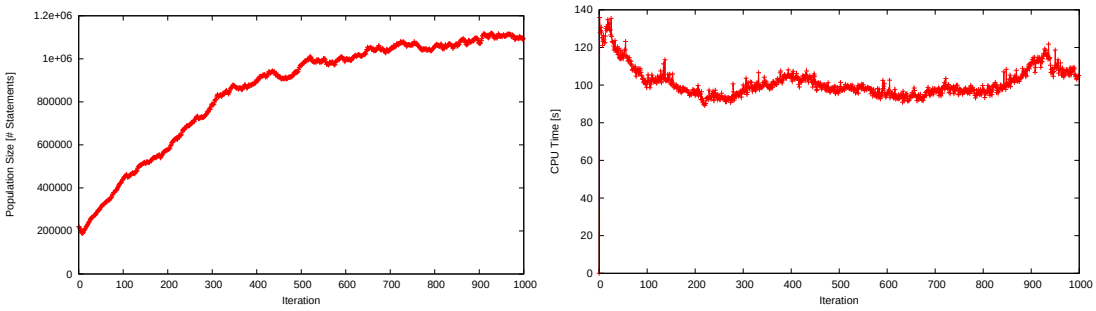

Figure 9.7: Probability of winning a match.

the last 20 generations, every *Individual* of the host population found $79.44$ pieces of food, while the *Individuals* of the parasite population found $76.28$ pieces on average.

Figure 9.7 shows the probability of a random player of the host and parasite population winning a match. The first short-lived increase in win probability of player 2 coincides with the emergence of species B. It clearly can be seen that the probability of winning the match for player 2 does not reach the probability for player 1. Note that as the winning probabilities draw closer, the probability of a tie increases, which was $2\%$ at the end of the run. The increase of winning probability for player 1 around generation 900 was not visible in any of the other graphs so it is not clear what caused it. It is, however, noteworthy that if the host population increased its fitness by changing its behaviour, the parasite population was able to adapt fast to reach the previous fitness levels. If the change was caused by some sort of defect in the parasite population, it was corrected fast. Whatever the cause, it does suggest stable populations and performance levels.

No species in both parts of the population evolved any kind of aggressiveness, there were of course some battles but they were more like accidents when the ants crossed paths than deliberate choices to fight. Similarly no support in battle evolved.

The fact that this and the following runs used coevolution had no influence on the qualitative characteristics of the population size and execution time. The population size doubled because the number of *Individuals* doubled. The execution time did not double because even with normal evolution some player 2 had to be executed every game. Again the execution time showed no correlation to the size of the population.




(a) (b)

Figure 9.8: (a) Total number of *Statements* contained in the population and (b) the time needed to evaluate them.

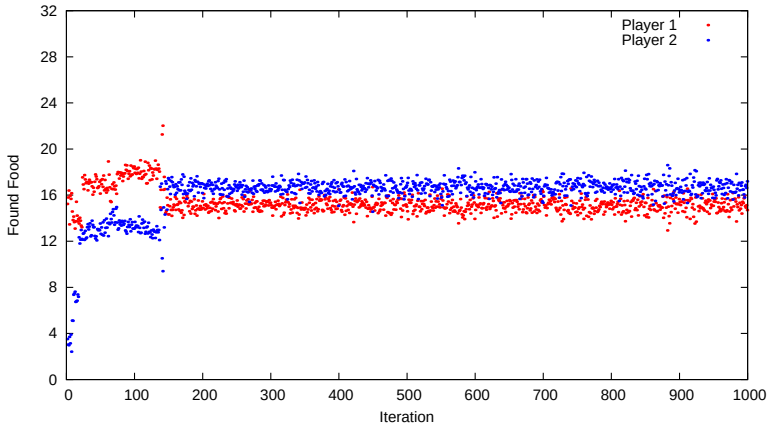
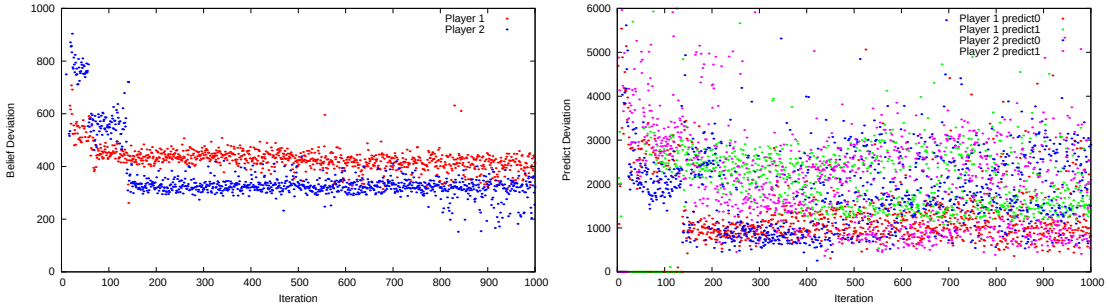



Figure 9.9: Average number of found pieces of food of the best *Individual* per game.




(a) Belief deviation

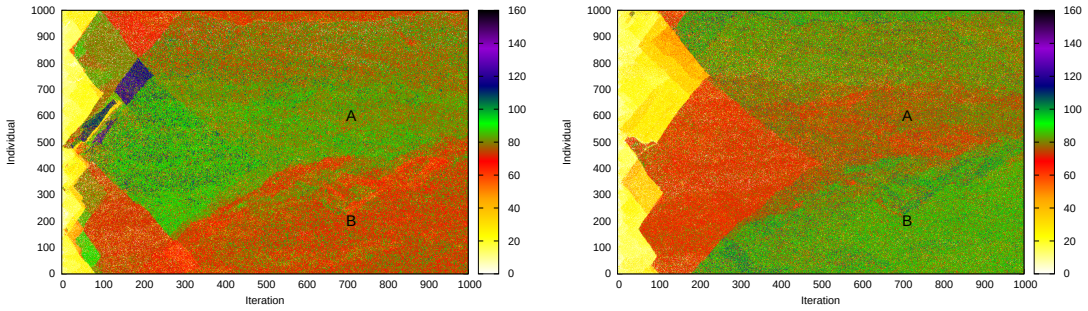
(b) Predict deviation

Figure 9.10: Average belief and predict deviations of the best *Individual* and its opponent per game.

## 9.2 Run with Asymmetric Evaluation

The expectation for the run with asymmetric evaluation was that player 1 would beat player 2 because he gets three times more evaluation information and GPS can better select the fitter *Individuals*. The run, however, did not turn out that way. Figure 9.9 shows the results of the battle between the best player 1 and the best player 2. Player 2 consistently beats player 1 by a convincing margin. Player 2 was able to do that after it developed the efficient use of both ants.

The deviation of the belief *Functions* in figure 9.10a shows that the best *Individuals* of the parasite population start using an improved version beginning with the 800-th generation. This improved belief *Function* initially believes in food at unseen positions and that food once seen is never eaten by the enemy's ants. After some moves, however, the belief *Function* reduces the belief at the west-most column of the playing field to zero. Every two moves on average the west-most column that still believes in food at unseen positions has its belief reduced to zero.




(a) Player 1

(b) Player 2

Figure 9.11: Found food of the *Individuals* playing 2-AntWars.

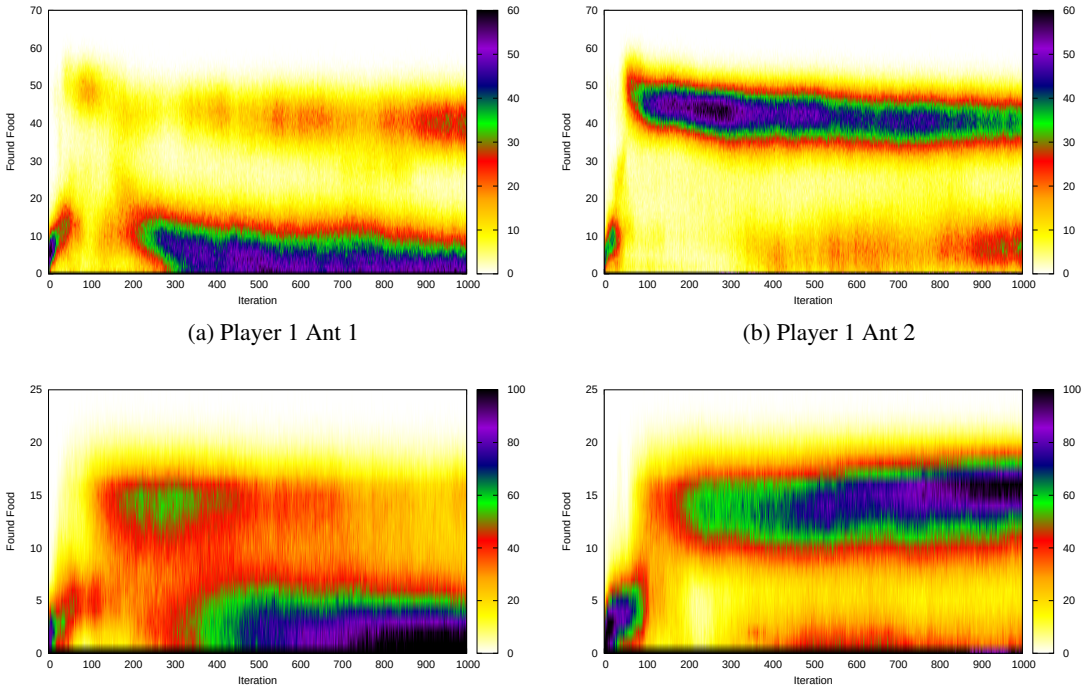
This method takes the movements of the enemy's ants into account, in areas where they collect food the belief is reduced to zero after some time.

The prediction deviations visible in figure 9.10b on the previous page show that in the first generations of the run the best *Individuals* of both players only use one ant to gather food. After 25 generations the best *Individuals* of player 1 started using both ants, the best *Individuals* of player 2 began using both ants at generation 140. The change of prediction accuracy of player 2's predict 1 *Function* around generation 400 means that the best *Individuals* of the host population started moving ant 1 first instead of ant 2. During the match of the best *Individuals* in the last generation, player 2 predicted player 1's ant 1 to stay at its starting position but it started to move and moved the prediction of ant 2 to $(8, 5)$ even though that ant did not move during most of the game. Player 1 predicts the enemy's ant 1 at its starting position and ant 2 at $(13, 5)$ which corresponds to the actual behaviour of the ants.

Figure 9.11 shows the found food per match of the *Individuals* in the population and even though player 1 does not beat player 2 as was expected, another expected result emerges: there is an area where player 1 beats player 2 (labeled A) and an area where player 2 beats player 1. This is only hinted at in this figure and will become clearer later.

The performance distribution of the ants depicted in figure 9.12 on the next page shows player 1 consisting of *Individuals* that use ant 1 predominantly and *Individuals* that instead use ant 2. Player 2 follows the behaviour most often observed until now by relying on ant 2 to find most of the food and cleaning up with ant 1 afterwards. Note that the found food of the ants of player 1 is three times as high compared to player 2 because player 1 got evaluated three times.

In the runs against a fixed strategy the amount of positions that a player uncovered was correlated with the fitness of the player because exploring more means finding more food in general. Figure 9.13 on the facing page, however, shows that for this run exploring less does not automatically mean lower fitness. For instance, in the parasite population (figure 9.13b on the next page) species C emerges that explores a big portion of the playing field. It is, however, replaced by D and E in quick succession, each exploring less of the playing field.



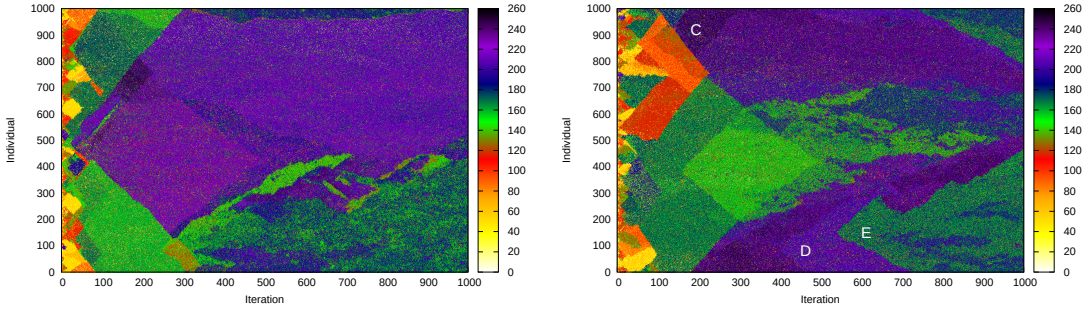

(a) Player 1 Ant 1

(b) Player 1 Ant 2

(c) Player 2 Ant 1

(d) Player 2 Ant 2

Figure 9.12: Average found food per ant and game.

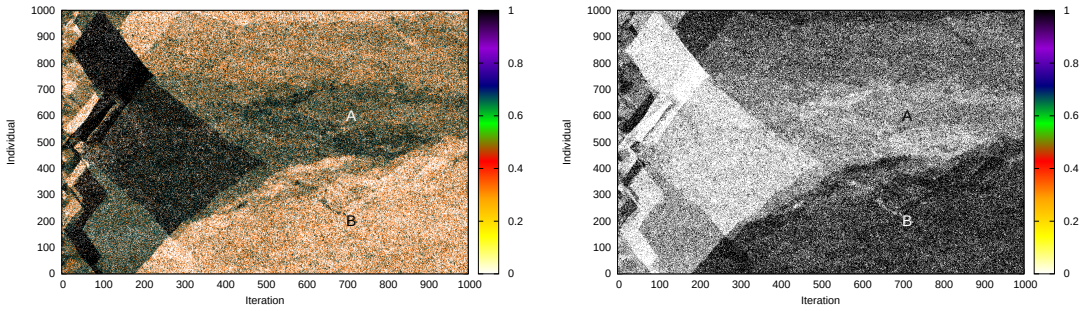



(a) Player 1

(b) Player 2

Figure 9.13: Average seen positions per game.

Figure 9.14 on the following page shows with more clarity what the fitness development of the population already suggested: two sections in the population, in one section player 1 is dominant, in one section player 2 is dominant. The sections are not stable, B slowly takes over the population and A does not find a way to beat it. Note that the data of player 1 is based on three matches but the data of player 2 on one match because of the asymmetric evaluation.




(a) Player 1 (b) Player 2

Figure 9.14: Probability of *Individuals* winning their matches.

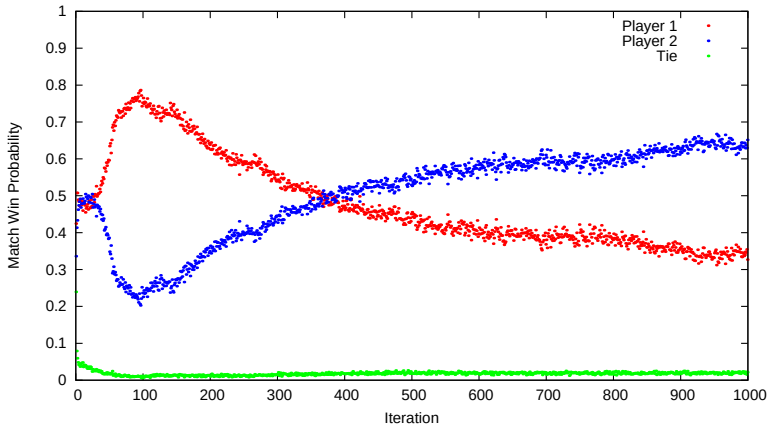



Figure 9.15: Probability of a random *Individual* winning an 2-AntWars match.

An overview of the development of winning probabilities is given by figure 9.15. It shows how the initially superior player 1 gets beaten more and more frequently, at the end of the run a random *Individual* playing player 1 has only a 35% chance of winning a match, while the chance of player 2 is 63%.

## 9.3 Long Run

The run presented in this section lasted 10000 generations. The aim was to study the long term coevolutionary behaviour of the 2-AntWars players and the data presented will focus on that.

The best overview of the long term evolution of the 2-AntWars players in this particular run is given by the probability of a random *Individual* winning his match, which is depicted in figure 9.16. Where shorter runs showed mostly static behaviour with one player constantly beating the other one in later generations, the winning probabilities of this run paint a completely different picture. For 4000 generations, player 1 won his matches very convincingly, then player 2 for 2000 generations. This was followed by player 1 winning for 1000 generations, no clear winner for 2000 generations and player 2 winning for the last 1000 generations. Player 1 had such a high probability of winning during the first 4000 generations because player 2 did not use both ants to collect food. The reasons for the superiority of one player over the other one in the following generations proved to be very difficult to extract. After both players learned to use both ants they reached a basic level of competence and only details in their behaviour varied. For instance, from generation 4000 to 6000 player 2 supports his ants in battle, around generation 6600 player 2 does not move with either ant if no position has food belief. In generation 9500 player 1 does not move with his free ant if the other one is in battle, player 2 does that. These differences might have been the important advantage for winning matches, but are not completely satisfactory explanations.

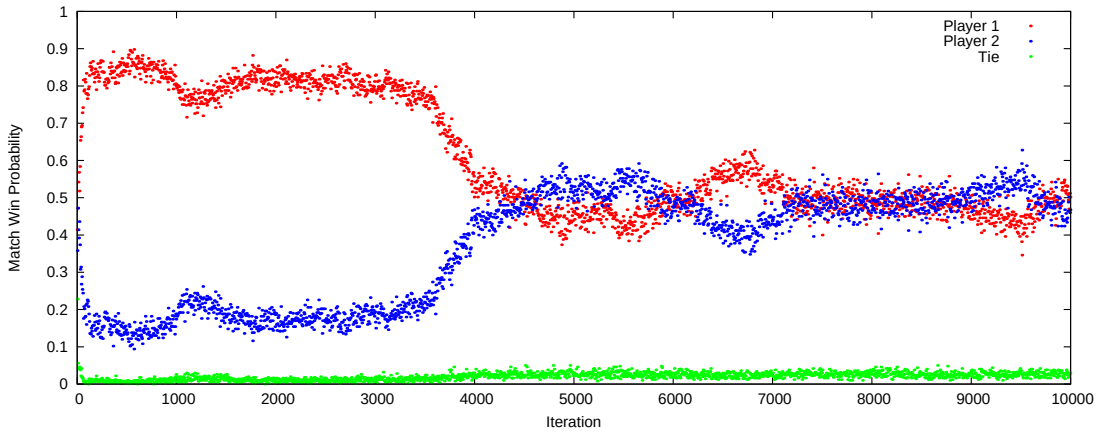
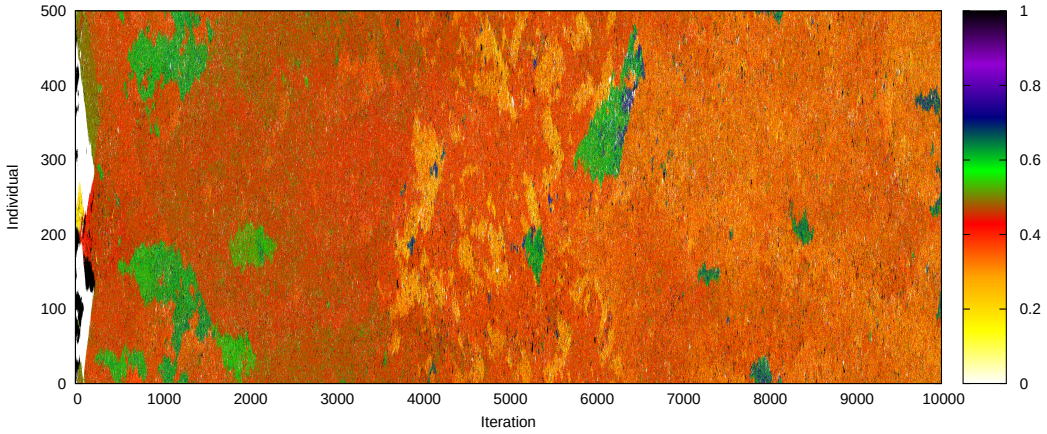
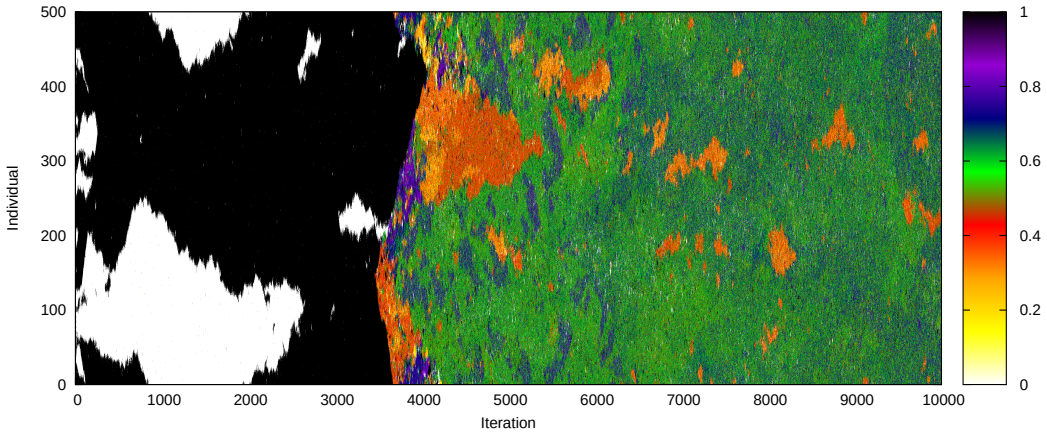



Figure 9.16: Probability of a random *Individual* winning a match of every fifth generation.

From the probability of choosing ant 1 for a move in figure 9.17 on the next page we can see exactly how long it took the parasite population to use both ants effectively and that player 1 favoured ant 2 and player 2 favoured ant 1. Player 1 seems to use ant 2 even more intensively as player 2 develops the use of both ants but this is only the result of shorter games (which means ant 1 of player 1 moves less often) as player 2 gathers food faster. It can also be seen that time and time again species emerged in the populations of both players that switched the use of the ants. These species were able to survive some time. It is unfortunate that there were no such




(a) Player 1

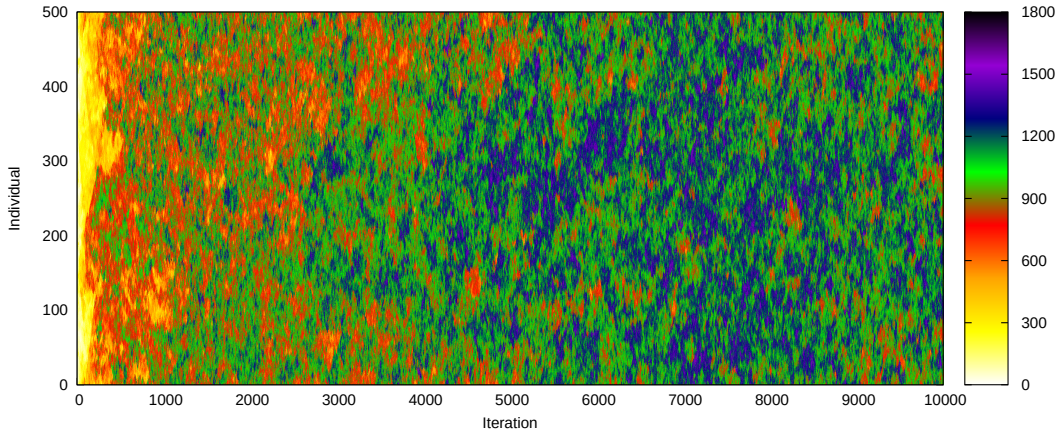



(b) Player 2

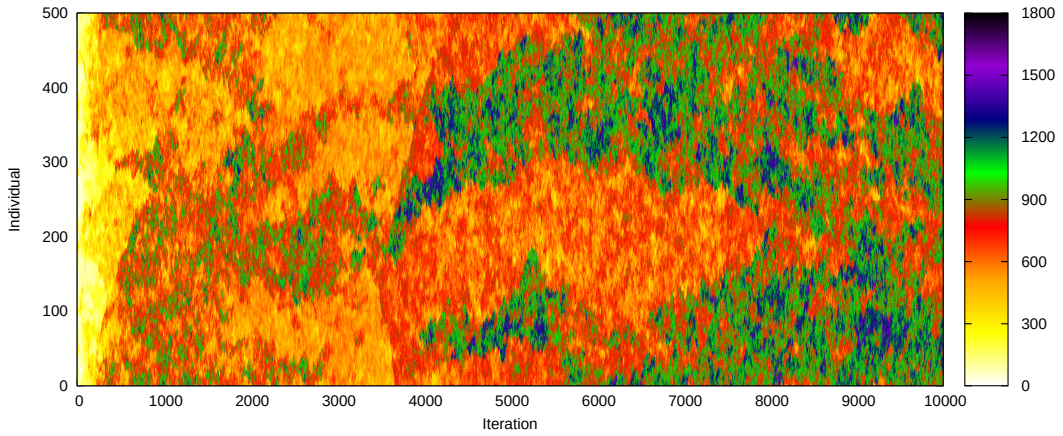
Figure 9.17: Probability of choosing ant 1 for a move.

switched species at the same time and location in both populations, maybe then they could have survived. This is based on the assumption that moving first with the same ant as the opponent is a disadvantage because of a higher risk of running into the opponent's ant.

Previous runs showed how the population size developed over time, and this run was no exception. Figure 9.18 on the facing page permits to gain insight into the size development of *Individuals* in the populations. Surprisingly, the populations diverged in terms of maximum size. Player 1's population was saturated with large *Individuals* while large sections of player 2's population contained medium sized *Individuals*, even after the use of both ants developed (which needs bigger *Individuals*). Normally one would expect both populations to behave in a similar way because they are manipulated in the same manner. The only difference is the actual composition of *Individuals*, so the structure of the *Individuals* might play a more important role for code bloat than previously thought.




(a) Player 1




(b) Player 2

Figure 9.18: Size of the *Individuals* playing 2-AntWars.

## 9.4 Long Run with Asymmetric Evaluation

This section presents the long term evolutionary development of a run spanning 6500 generations with asymmetric evaluation.

Figure 9.19 shows the probability of a random *Individual* winning a match. Once again the population did not converge to stable performance levels, both players fought for superiority instead. The actual reasons as to why one player was better than the other one proved to be elusive. The initial advantage of player 1 was due to the fact that it took player 2 200 generations to develop the use of both ants. During the rest of the run the belief *Function* and the adaption of the movement and decision *Functions* seemed to be pivotal to the superiority of one player over the other. The analysis of the games of the best *Individuals* of generation 1004 showed that the belief *Function* of player 1 switched the belief in food on unseen positions very early to zero and his ants stopped moving because of that, which gave player 2 a significant advantage. In generation 4500 the situation was reversed, player 2 was hindered by absent food belief at unseen positions. At the end of the run, player 2 regained the upper hand by using a belief *Function* that set the food belief from the forth to the seventh row from the top of the playing field to zero after 13 moves. This corresponds to the immediate surroundings of ant 2 of player 1. The section of zero belief grew larger until only the top-most and bottom-most row had food belief remaining. This has the effect that when an ant has explored one of the rows in search of food it traverses the playing field to explore the other row. During the traversal the ant has a chance to find food that it does not believe in. This chance would not be present if the food belief was reduced to zero column wise from the starting positions of the opposing ants towards the own starting positions, which was the behaviour observed in section 9.1 on page 91.

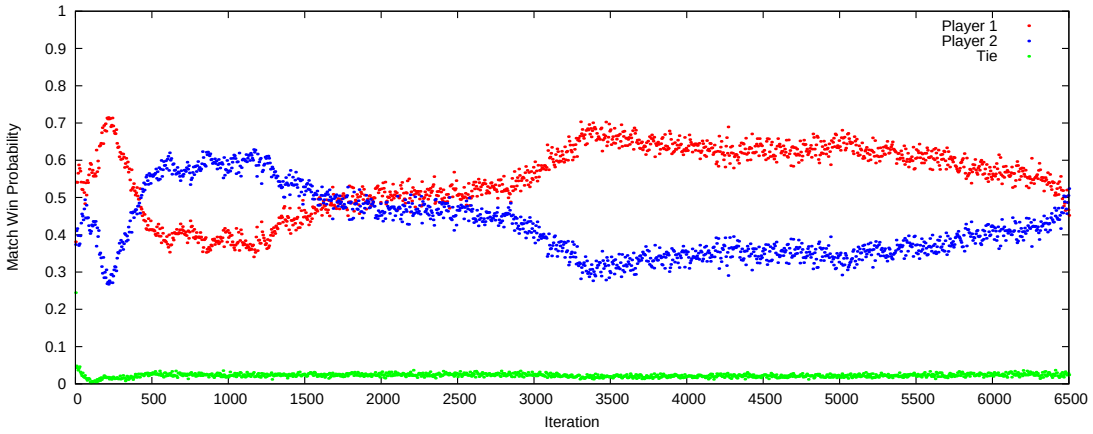



Figure 9.19: Probability of a random *Individual* winning a match of every fifth generation.

Ant 2 of player 1 (which was the preferred ant of that player) was more aggressive than any other ant but was so consistently during the whole run and therefore did not influence the changes of the chance to win a game. In contrast to the previous run with asymmetric evaluation, no coadapted species emerged that occupied parts of the population.

CHAPTER 10

# Special Analysis

This chapter examines properties of the 2-AntWars problem and GPS that did not receive sufficient attention in the previous chapters. Section 10.1 reports on experiments that tried to develop players against multiple opponent strategies simultaneously. In section 10.2 on page 107 the stability of the results of GPS when solving the 2-AntWars problem is investigated. Section 10.3 on page 109 determines how much the developed players are adapted to their opponents and how much the predict *Functions* influence the performance of the developed players.

## 10.1 Mixed Opponent Strategies

The runs against fixed strategies until now developed players by opposing them with one strategy. As a result, the developed players adapted to the specific strategy of their opponent to beat it (the extent of this adaption is discussed in section 10.3 on page 109). However, it is not clear if players can be developed that beat multiple strategies simultaneously. In this section the results of two runs against a mix of strategies are presented. The population of player 2 did contain the second implementations of the Greedy, Scorched Earth and Hunter strategies. Two different configurations of the player 2 population were tested, Block and Stripe.

In the Block configuration each strategy is placed at a continuous range of positions spanning one third of the population. For the run with this configuration the Greedy strategy was placed at positions 1 to 333, Scorched Earth at 334 to 666 and Hunter at 667 to 1000. Figure 10.1 on the next page shows the fitness development of the population, the three opponent strategies can be clearly distinguished. The population achieved the best results against the Scorched Earth strategy and the worst results against the Hunter strategy. As can be seen from Figure 10.2 on the following page, the developed players were not able to beat any of the opposing strategies. It took 900 generations until a species emerged that could beat Scorched Earth. This species was the first one fit enough to survive by using both ants to collect food.

In the Stripe configuration each strategy is placed at every third position, i.e. the Greedy strategy is at the first position, the Scorched Earth strategy is at the second position, the Hunter strategy is at the third position, the Greedy strategy is at the fourth position and so on. Figure 10.3

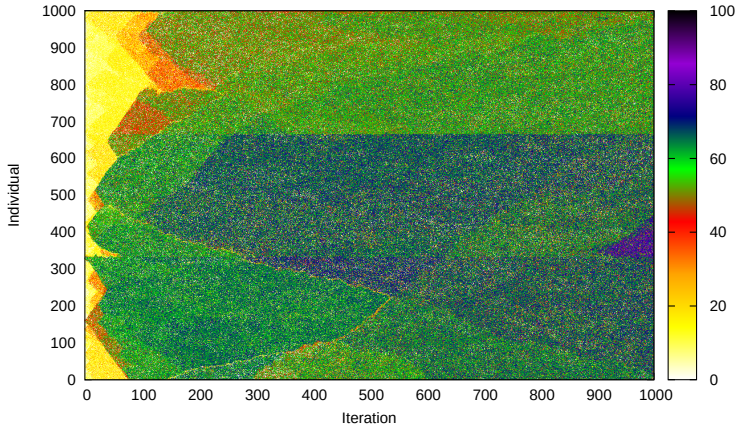
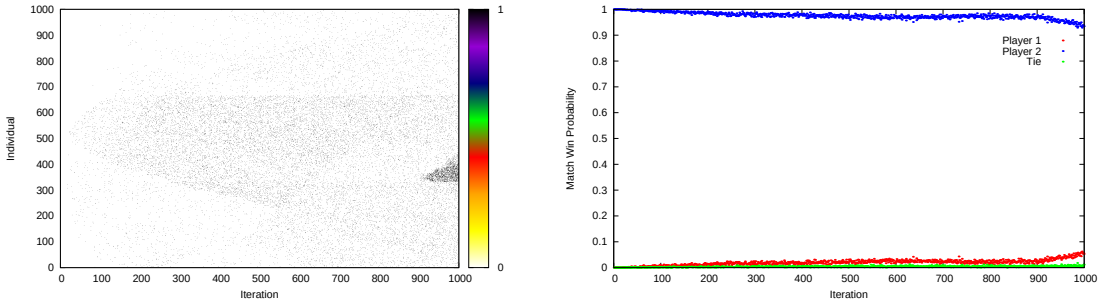



Figure 10.1: Found food of the *Individuals* playing 2-AntWars against the Block configuration.




(a) Players winning against the Block configuration

(b) Probability of winning a match

Figure 10.2: (a) Location of players able to win and (b) probability of winning a match against the Block configuration.

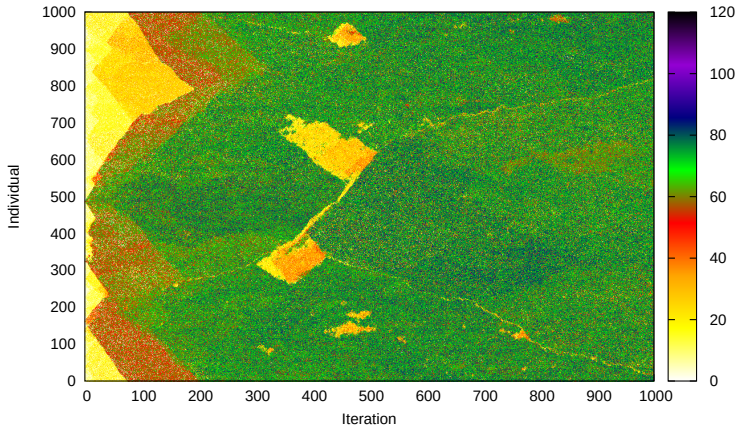
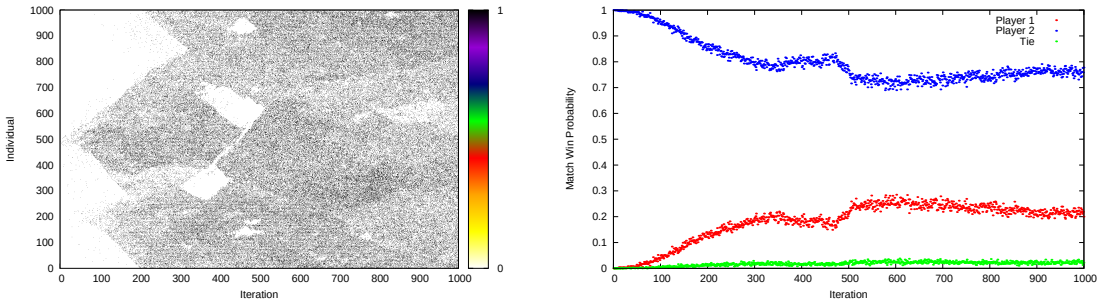



Figure 10.3: Found food of the *Individuals* playing 2-AntWars against the Stripe configuration.




(a) Players winning against the Stripe configuration

(b) Probability of winning a match

Figure 10.4: (a) Location of players able to win and (b) probability of winning a match against the Stripe configuration.

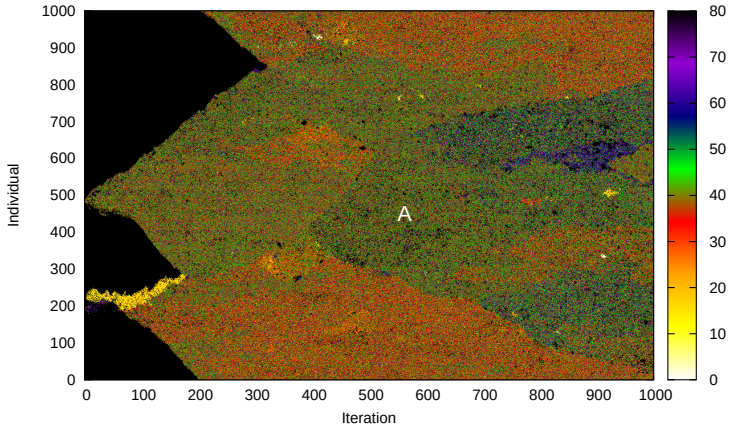



Figure 10.5: Average time until both ants have left their starting positions.

shows that the run against the Stripe configuration was more successful than the one against the Block configuration but the developed players were still not able to beat all the opponent strategies, as can be seen in figure 10.4. The dip in the win probability shown in figure 10.4b is caused by the spread of the inferior species A (figure 10.5) which does not move both ants in some of its games. Once again, this effect was traced back to the spread of a fitter belief *Function* the movement *FunctionGroup* was not adapted to.

## 10.2 Stability of Results

Due to the computational requirements of 2-AntWars runs, the results presented so far were based on a single run. A single run is not enough evidence to reason about the general behaviour of GPS in the 2-AntWars context. To mitigate this shortcoming, the run against the second implementation of the Greedy strategy was repeated 30 times. This particular opponent was chosen

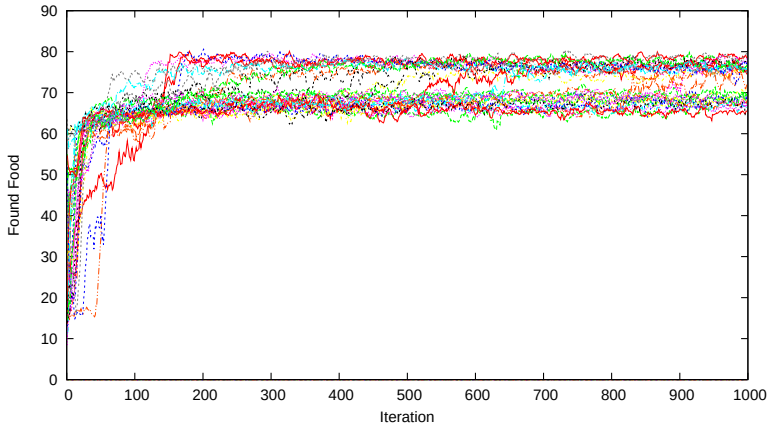



Figure 10.6: Running average (using 10 values) of found pieces of food per match of the best *Individuals* of 30 runs against the second implementation of the Greedy strategy.

because the one run that was presented in section 8.1 on page 73 did not yield a satisfactory result. Figure 10.6 shows the fitness development of the 30 runs. It can be seen that there were two principal outcomes of a run, the best *Individuals* either reached about 68 or 77 pieces of food. The difference between those clusters is that the *Individuals* contained in the better group used both ants to gather food. 13 best *Individuals* of the last generation were not able to do that. The games that those *Individuals* played showed that the decision *Function* was not the sole problem, more than once even the one ant that was used stopped moving after the food belief at unseen positions was set to zero. In 10 runs the best *Individuals* (regardless from which generation) never used both ants. The graph of the fitness development also shows that the successful use of both ants can develop at any time, as early as the 50-th generation or as late as the 900-th.

To answer the question whether GPS was able to develop a 2-AntWars player capable of beating the Greedy strategy the best *Individuals* of the 10 last generations of every run were selected to play 1000 matches against the Greedy strategy. On average the *Individuals* only had a $19.8\%$ chance of winning a match. The *Individuals* of the worst run had a $4.2\%$ chance to win while the *Individuals* of the best run had a chance of $43.7\%$ on average. Only one *Individual* from the best run had a higher chance of winning than the Greedy strategy but the difference was not statistically significant, based on a one-tailed t-test. All of the following significance results were calculated with this test (two-tailed where necessary).

Since the data of 30 runs against the Greedy strategy was available it was decided to test the influence of the mutation operator on the end result. With the standard settings a single *Statement* has a probability of $0.1\%$ to mutate which lets one question the beneficial influence mutation could possibly have. 16 runs against the Greedy strategy were performed without mutation while keeping all the other settings. Figure 10.7 on the facing page shows the fitness development of those runs. It can be seen that again 2 clusters formed. Only the *Individuals* belonging to the two runs in the better cluster used both ants effectively.

To evaluate the capabilities of the created players the same procedure as before was used, i.e.

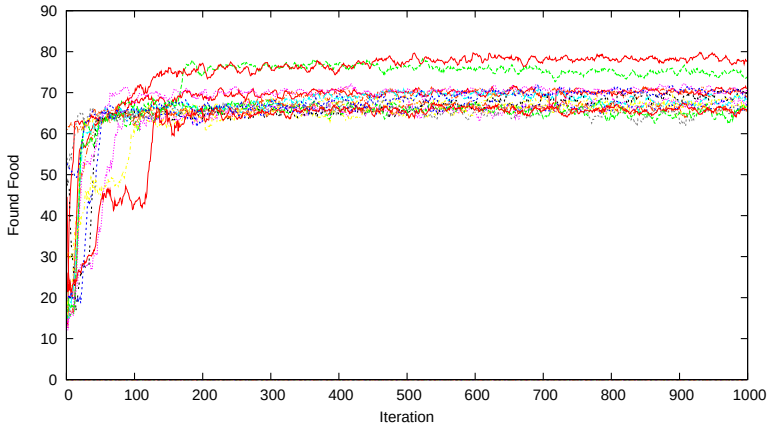



Figure 10.7: Running average (using 10 values) of found pieces of food per match of the best *Individuals* of 16 runs without mutation against the second implementation of the Greedy strategy.

the best *Individuals* of the last 10 generations of every run had to play 1000 games against the Greedy strategy. On average, the *Individuals* had a $7.8\%$ chance of winning, which is far worse than the runs with enabled mutation. The difference is significant at the $1\%$ level. This proves that, at least when developing players against the second implementation of the Greedy strategy, runs using mutation yield far better results. The usefulness of the crossover operator was already seen in the runs where beneficial movement behaviour switched between the movement1 and movement2 *Functions*.

## 10.3 Playing against Unknown Opponents

While 2-AntWars players are developed, they are always battled against the same (or the three same) fixed strategies or against players from the same generation in coevolutionary runs. It is expected that the developed players adapt to their opponents to be able to beat them. The extent of the adaption, however, is unknown. For instance, are players that were developed against the Scorched Earth strategy able to beat other strategies or is their behaviour only successful against this one strategy? A related question concerns the development of players in coevolutionary runs. Is a player that is superior to its current opponent also able to beat opponents from previous generations? The expected answer is no, because coevolution does in no way ensure this; see the discussion of historical progress in chapter 2 on page 5. With this kind of data available also a third question can be answered. Do the developed players use the position predictions of the enemy's ants in a constructive way or do they ignore this information? In the analysis of the runs against fixed strategies it could not be observed that the position predictions have any influence at all on the performance of the developed players.

To determine the performance of the developed players, the best *Individuals* of the 10 last generations of a run were chosen to represent the result of the run. Each of these *Individuals* then

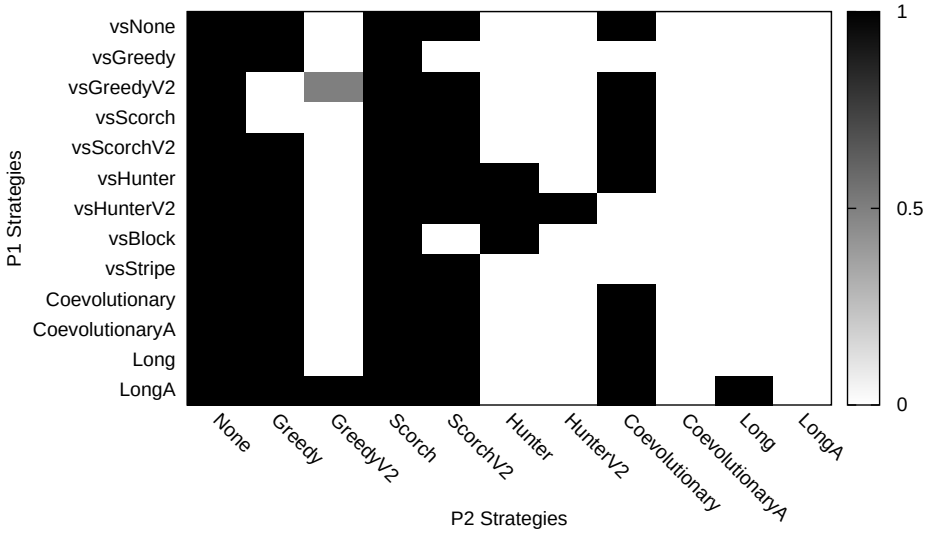



Figure 10.8: Results of battling developed and fixed 2-AntWars strategies against each other. 1 denotes that the strategy of player 1 is superior, match-ups where it loses are marked with 0. Differences in the probability of winning between player 1 and 2 that are not statistically significant at the $1\%$ level are marked with $0.5$.

played 100 matches against each of the 10 *Individuals* representing the result of the opposing strategy which was chosen in the same way. If player 2 was a fixed strategy, it was simply used 10 times. The results of the matches were used to determine the win probabilities of player 1 and player 2 which can be found in appendix A on page 121.

Figure 10.8 shows the results of battling the developed players and fixed strategies against each other. The most difficult fixed strategies to beat were the second implementation of Greedy (GreedyV2) and both Hunter strategies. It can be seen that there is no statistically significant difference between the performance of the player developed against the GreedyV2 strategy (vsGreedyV2) and GreedyV2. However, when looking at the results of the 10 *Individuals* of vsGreedyV2 separately, four *Individuals* beat GreedyV2 at the $1\%$ level and two additional *Individuals* beat GreedyV2 at the $5\%$ level, so GPS was successful in finding a player capable of beating GreedyV2. It is notable that vsGreedyV2 was beaten by Greedy. The only other player that was able to beat GreedyV2 was the result of the long coevolutionary run with asymmetric evaluation (LongA). The HunterV2 strategy was only beaten by vsHunterV2, the player that was developed specifically to beat this strategy. This means that HunterV2 requires special adaption to beat it. The performance of the players developed against the Block and Stripe configurations was very poor. Only the player developed against the Stripe configuration was able to beat one of the opponent strategies, ScorchV2. The best found players were the player 2 *Individuals* of the coevolutionary runs with asymmetric evaluation which were able to beat every other developed player.

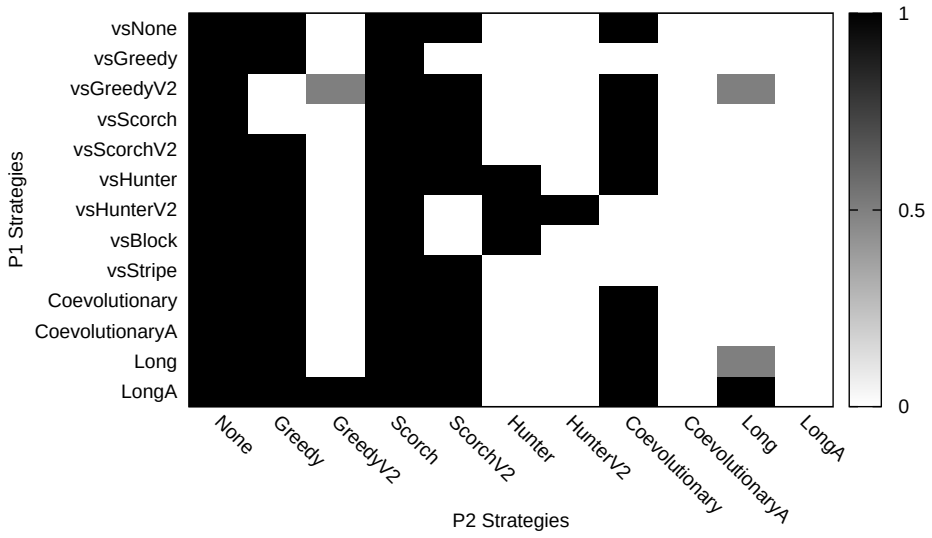



Figure 10.9: Results of battling developed and fixed 2-AntWars strategies against each other. The predict *Functions* of all developed strategies were overridden with a *Function* that always returns the last prediction. 1 denotes that the strategy of player 1 is superior, match-ups where it loses are marked with 0. Differences in the probability of winning between player 1 and 2 that are not statistically significant at the $1\%$ level are marked with $0.5$.

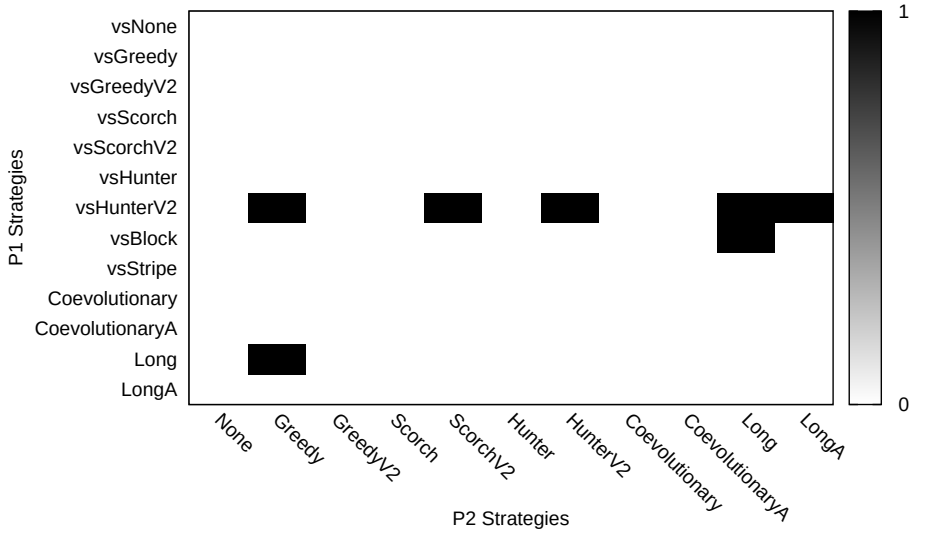



Figure 10.10: Match-ups between strategies where the probability of winning of either player 1 or player 2 differs significantly (at the $1\%$ level) between matches with the original and overridden prediction *Functions*.

Figure 10.9 on the preceding page shows the same results after the prediction *Functions* of the developed players were overridden with a prediction *Function* that always returns the last prediction, which is equivalent to the starting position or the location where the enemy's ant was last seen. The results are essentially the same, but in two additional match-ups the performance difference is not statistically significant. This is a hint that the predict *Functions* indeed have an influence on the performance of the developed players. Figure 10.10 on the previous page shows this more clearly. Especially the performance of vsHunterV2 is significantly influenced by changed predict *Functions*. Contrary to expectations the performance of vsHunterV2 actually improved from a $52.7\%$ to a $55.3\%$ chance of winning a match against HunterV2. In the other two match-ups the player 1 strategy suffers a reduction in performance when the predict *Functions* are overridden.

To evaluate the strategies developed during the long coevolutionary run, 10 *Individuals* were chosen from six distinct phases of the fitness development of this run that was shown in figure 9.16 on page 101. No *Individuals* were chosen from the first 5000 generations because player 2 had not developed the use of both ants. Player 2 dominated at generations 5000 and 5500, player 1 gained the upper hand at generation 6800. At generation 8000 both players seemed to be on equal footing. Player 2 won around generation 9500 and player 1 at the end of the run. The *Individuals* of each phase were chosen among the best *Individuals* from generations around the mentioned ones. Only *Individuals* were chosen that conformed to the current trend, for instance for generation 5000, players from a particular generation (around generation 5000) were only chosen if player 2 beat player 1. For the phase around generation 8000 five generations

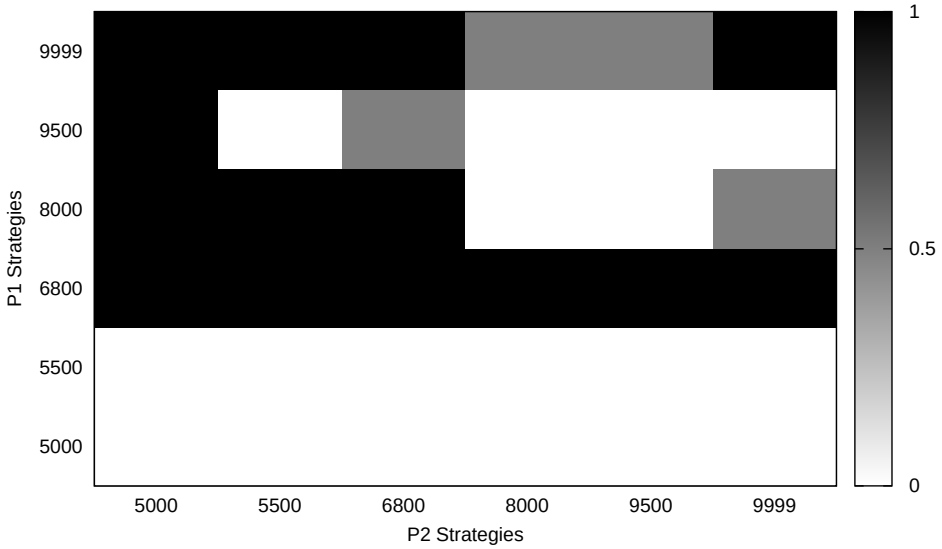



Figure 10.11: Results of battling *Individuals* from different generations of the long coevolutionary run against each other. 1 denotes that the strategy of player 1 is superior, match-ups where it loses are marked with 0. Differences in the probability of winning between player 1 and 2 that are not statistically significant at the $1\%$ level are marked with $0.5$.

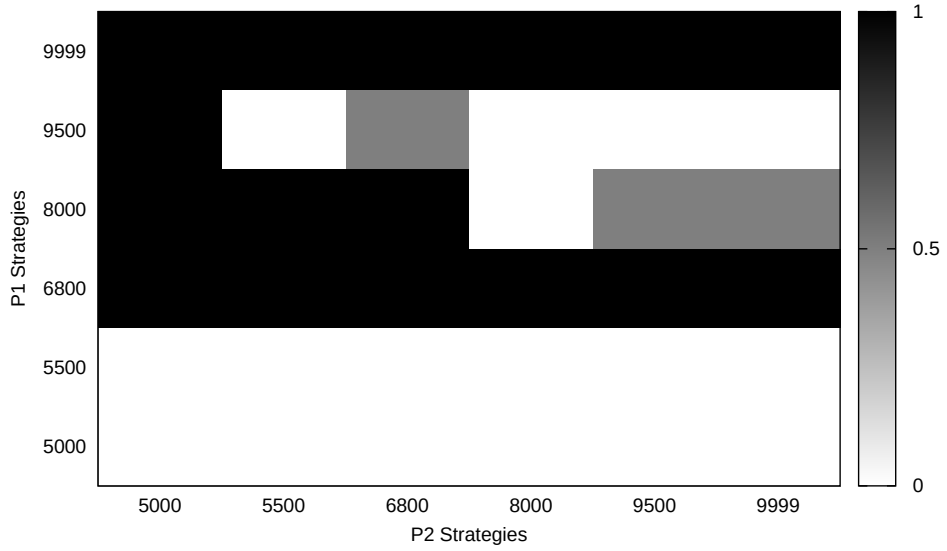



Figure 10.12: Results of battling *Individuals* from different generations of the long coevolutionary run against each other. The predict *Functions* of all *Individuals* were overridden with a *Function* that always returns the last prediction. 1 denotes that the strategy of player 1 is superior, match-ups where it loses are marked with 0. Differences in the probability of winning between player 1 and 2 that are not statistically significant at the $1\%$ level are marked with $0.5$.

were chosen where player 1 won and five where player 2 won. Each player of each development phase played 100 matches against every other player of every phase. The results are presented in figure 10.11 on the preceding page. The sampled player 2 strategies from generation 8000 were able to beat the sampled player 1 strategies. The player 1 strategies from generation 6800 demonstrate the effect that was searched for. They are able to beat every player 2 strategy but the later player 1 strategies are not able to do so. Even the player 1 strategies from the end of the run which were superior to their opponents (from the same generation) were not able to beat every strategy.

With overridden predict *Functions* the player 1 strategies of the end of the run were able to beat all strategies of player 2 as shown in figure 10.12. Figure 10.13 on the next page shows which match-ups were significantly influenced by the removal of the predict *Functions*. As already mentioned, the player 1 strategies of the end of the run improved but it is not clear if they really improved or the opposing player 2 strategies depended on functioning predict *Functions* and were damaged by their removal. Note that this kind uncertainty does not exist for the performance improvement of vsHunterV2 because the prediction *Functions* of the fixed strategies (and therefore also HunterV2) were not changed. The second set of significant differences also saw an increase in performance of player 1.

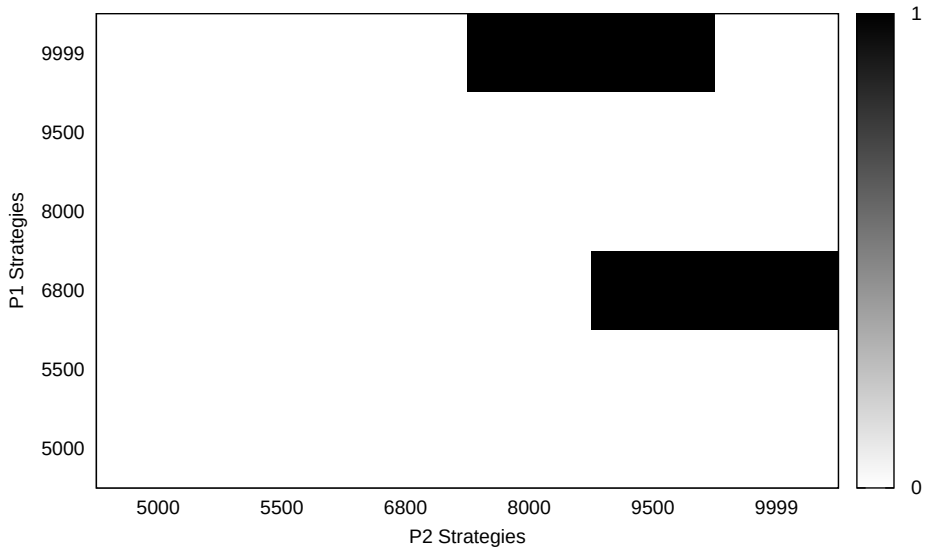



Figure 10.13: Match-ups between strategies extracted from the long coevolutionary run where the probability of winning of either player 1 or player 2 differs significantly (at the $1\%$ level) between matches with the original and overridden prediction *Functions*.

CHAPTER 11

# Conclusion

In the course of this thesis, the results of nine evolutionary and four coevolutionary experiments to develop capable 2-AntWars players were presented. It was demonstrated that the used genetic programming system was powerful enough to beat all of the seven static game strategies.

The basis of all developed players was greedy movement in order to to collect as much food as possible, but the details varied a lot and were tailored to the specific strategy of the opponent. The first implementation of the Scorched Earth strategy for example was beaten by Reverse Scorched Earth; the developed player moved one of his ants into the opponent's half of the playing field and collected the food located there before the opponent (playing Scorched Earth) had the opportunity. This strategy even reduced the greediness of one ant to get it faster into the opponent's half. The two implementations of the Hunter strategy were beaten by baiting the, and hiding from the, hunting ant respectively. In addition, the battle support mechanic was used to combat the hunting ant. During most of the runs the developed players showed a rather pacifistic behaviour by ignoring the opposing ants. One exception, however, was the run against the second implementation of the Greedy strategy, the developed player routinely neutralized one of the opponent's ants.

Another trend that was observed during the runs was that the developed players always used one ant until it was not able to move anymore before the second ant was moved. Which ant moved first depended on the opponent. For instance, against the first implementation of the Hunter strategy ant 1 was moved first, against the second implementation of the Greedy strategy ant 2 was dominant. When faced with the second implementation of the Hunter strategy, using a specific ant first did not yield any advantage and the population was split in two halves, one moving ant 1 first and one ant 2. Of course there were exceptions to this general trend. In nearly every run species emerged that switched the moved ant more freely, but those species did not survive unless it was a clear advantage. This was the case in the run against the second implementation of the Hunter strategy, where both ants were moved early to hide from the hunting ant of the opponent. Repeated runs against the second implementation of the Greedy strategy showed that about ⅓ of the conducted runs did not create players that are able to use both ants effectively.

The success of the runs was not only due to high performing movement and decision *Functions*. The belief *Functions* in particular developed some interesting methods of approximating the disappearance of food from the playing field. Most of them were based on switching the food belief from one to zero instead of interpolating between those values based on the elapsed time. The interpolation of belief only occurred in the run against the first implementation of the Greedy strategy. During all other runs the developed belief *Function* started out with food belief everywhere and then switched it off based on time. This switch-off occurred in multiple variants depending on the opponent that was faced. Sometimes the food belief for all unseen fields was set to zero and sometimes the disbelief grew row wise or column wise from the starting positions of the opponent's ants and once the disbelief grew row wise from the top of the playing field to the bottom. The behaviour of the belief *Function* was observed to have significant influence on the performance of the player. Unfortunately, this influence was not always positive, in at least three runs otherwise superior species went extinct or suffered a significant performance penalty because they were depending on a non-optimal belief *Function*. In the run against the second implementation of the Scorched Earth strategy a species was able to adapt to a changed behaviour of the belief *Function* and regained its lost performance.

The contribution of the predict *Functions* to the overall performance of players was questionable. Overriding the developed *Functions* either did not change the fitness in a significant way or actually improved it. Two principal behaviours were observed with the predict *Functions*, jumping with the prediction to some location that is a reasonable approximation of the position of the predicted ant or moving the prediction from the starting position to such a location.

The coevolutionary runs showed that it is possible for the players to adapt to each other in a way that one species in one population beats the *Individuals* at their positions in the other population while the other population contains a second species that beats *Individuals* in the first population at their positions. The experiments testing long term coevolution demonstrated that the populations were not subject to loss of gradient or disengagement. They battled fiercely for dominance and the inferior player always developed a way to gain the upper hand again. Historical progress could not be observed as the players of later generations in some cases lost to players from earlier generations. Asymmetric evaluation did not yield any notable changes in the evolutionary process besides taking three times longer than symmetric evaluation.

The deeper analysis of the runs unearthed some quite interesting evolutionary behaviour. The first coevolutionary run showed that movement capabilities of one movement *Function* can switch almost seamlessly to the movement *Function* of the other ant. The same run also demonstrated that the combination of two species can yield a new species that is superior to both when a species with an exceptional movement *Function* for one ant combined with a species that used both ants to collect food. This proves the usefulness of the crossover operator for the 2-AntWars problem. Mutation was also instrumental for the success of the runs as removing mutation increased the chance of not discovering the use of both ants from 1/3 to 7/8. In the first run with asymmetric evaluation there were two species in a row that reduced the amount of positions they uncovered to improve their performance, which is quite counter-intuitive because normally more uncovered fields result in more opportunities to find food. In these cases not exploring every position was advantageous because the unexplored positions were most likely already harvested by the enemy's ants and so the available moves could be spent more wisely.

Generally, the populations converged to one level of fitness but that does not mean that there was a loss of diversity. Often there were species contained in the populations that chose different ants for the majority of the game, moved less random than others or were more aggressive, so after a run had finished there were lots of *Individuals* with different successful strategies available. This was especially true of the long coevolutionary runs where both parts of the populations developed capable players and only small variances in playing style tipped the scales from one player to the other.

From the point of view of evaluating the rules of 2-AntWars, it seemed that there are not enough reasons to use both ants simultaneously. In some runs this simultaneous use emerged only to die out again. With the rules as they are now, a game is split up into two phases. In the first phase one ant of each player tries to find as much food as possible and in the second phase the other ant of each player tries to find the food that was left behind. When two competent players face each other, the second phase is usually short, with the ants finding about 2 pieces of food on average. Maybe it would have been a good idea to reduce the allowed moves by the ants. That way a movable ant is more valuable and both ants of both players are really needed to find all the food on the playing field.

There are multiple directions for future research. One main task is to make it feasible to repeat runs multiple times to investigate the influence the population model and selection operator have on the final result. This thesis demonstrated that the chosen model and selection operator enabled the creation of multiple species inside the population but it is not clear that this has a positive influence on the final outcome. The chosen selection operator could probably be improved by taking the stochastic nature of evaluation into account, for instance it could use a statistic test to determine if one *Individual* is significantly better than another one. Furthermore, it could be beneficial to develop a system that automatically recognises an emerging species and puts it into a repository, so that one long coevolutionary run is enough to create a wide range of successful playing strategies without human intervention. Another direction for research is adapting the rules of 2-AntWars to explore the influence of changed rules on the playing style of the developed players. Reducing the number of allowed moves per ant was already mentioned, other possibilities are adding a third ant for each player, making the capabilities of the ants asymmetric (e.g. one ant can move faster but the other ant sees further) or reducing the number of games per match to see when the noise of random food placement prevents evolutionary progress. A third direction could be adapting the model of the 2-AntWars player. It was already seen that the predict *Functions* did not help the performance of the players but more fundamental changes could also result in a more capable playing style. The presented model was decision based, i.e. *Functions* calculated explicitly the next move. Another possibility would be to develop scoring functions for each possible move and the move with the highest score will be executed.

# Part IV

# Appendix

APPENDIX A

# Strategy Evaluation

This chapter contains the data that section 10.3 on page 109 was based upon.

Table A.1: Win probabilities of player 1 ($p_1$) and player 2 ($p_2$) strategies extracted from different development phases (labeled by the generations they occurred in) of the coevolutionary run spanning 10000 generations based on 10000 matches.

| | Player 2 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 5000 | | 5500 | | 6800 | | 8000 | | 9500 | | 9999 | |
| Player 1 | $p_1$ | $p_2$ | $p_1$ | $p_2$ | $p_1$ | $p_2$ | $p_1$ | $p_2$ | $p_1$ | $p_2$ | $p_1$ | $p_2$ |
| 5000 | 0.441 | 0.514 | 0.390 | 0.566 | 0.400 | 0.558 | 0.332 | 0.620 | 0.355 | 0.597 | 0.336 | 0.618 |
| 5500 | 0.439 | 0.521 | 0.345 | 0.618 | 0.383 | 0.577 | 0.313 | 0.639 | 0.320 | 0.639 | 0.326 | 0.630 |
| 6800 | 0.726 | 0.244 | 0.649 | 0.315 | 0.635 | 0.329 | 0.546 | 0.404 | 0.541 | 0.411 | 0.545 | 0.406 |
| 8000 | 0.640 | 0.331 | 0.538 | 0.426 | 0.544 | 0.421 | 0.444 | 0.514 | 0.453 | 0.506 | 0.475 | 0.484 |
| 9500 | 0.542 | 0.425 | 0.450 | 0.518 | 0.480 | 0.482 | 0.385 | 0.570 | 0.382 | 0.581 | 0.395 | 0.561 |
| 9999 | 0.656 | 0.315 | 0.574 | 0.395 | 0.569 | 0.395 | 0.478 | 0.477 | 0.477 | 0.481 | 0.500 | 0.453 |

Table A.2: Win probabilities of player 1 ($p_1$) and player 2 ($p_2$) strategies extracted from different development phases (labeled by the generations they occurred in) of the coevolutionary run spanning 10000 generations based on 10000 matches. The predict *Functions* of the strategies were overwritten with *Functions* that always return the last prediction.

| | Player 2 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 5000 | | 5500 | | 6800 | | 8000 | | 9500 | | 9999 | |
| Player 1 | $p_1$ | $p_2$ | $p_1$ | $p_2$ | $p_1$ | $p_2$ | $p_1$ | $p_2$ | $p_1$ | $p_2$ | $p_1$ | $p_2$ |
| 5000 | 0.447 | 0.512 | 0.397 | 0.557 | 0.401 | 0.555 | 0.331 | 0.619 | 0.358 | 0.594 | 0.336 | 0.619 |
| 5500 | 0.442 | 0.519 | 0.349 | 0.610 | 0.377 | 0.585 | 0.303 | 0.651 | 0.308 | 0.645 | 0.330 | 0.621 |
| 6800 | 0.729 | 0.243 | 0.648 | 0.312 | 0.642 | 0.319 | 0.570 | 0.382 | 0.585 | 0.370 | 0.576 | 0.380 |
| 8000 | 0.645 | 0.327 | 0.534 | 0.429 | 0.535 | 0.431 | 0.467 | 0.493 | 0.471 | 0.491 | 0.482 | 0.475 |
| 9500 | 0.551 | 0.417 | 0.446 | 0.518 | 0.481 | 0.486 | 0.394 | 0.560 | 0.397 | 0.563 | 0.401 | 0.557 |
| 9999 | 0.673 | 0.296 | 0.569 | 0.398 | 0.577 | 0.384 | 0.505 | 0.450 | 0.519 | 0.438 | 0.516 | 0.438 |

Table A.3: Win probabilities of player 1 ($p_1$) and player 2 ($p_2$) strategies extracted from the results of the runs against fixed strategies (no opponent (N), Greedy opponent (G,G2), Scorched Earth opponent (S,S2), Hunter opponent (H,H2)) and the coevolutionary runs (coevolutionary with standard settings (C), with asymmetric evaluation (CA), long run (L) and long run with asymmetric evaluation (LA)). The player 1 strategies also contain the results of the run against the Block (BL) and Stripe (ST) opponent configurations. The probabilities are based on the results of 10000 matches.

| | | Player 2 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Player 1 | | N | G | G2 | S | S2 | H | H2 | C | CA | L | LA |
| N | $p_1$ | 1.000 | 0.919 | 0.370 | 0.994 | 0.508 | 0.356 | 0.268 | 0.556 | 0.216 | 0.310 | 0.357 |
| | $p_2$ | 0.000 | 0.064 | 0.588 | 0.004 | 0.452 | 0.625 | 0.709 | 0.402 | 0.750 | 0.640 | 0.594 |
| G | $p_1$ | 1.000 | 0.864 | 0.250 | 0.995 | 0.335 | 0.229 | 0.108 | 0.466 | 0.137 | 0.201 | 0.254 |
| | $p_2$ | 0.000 | 0.114 | 0.715 | 0.004 | 0.628 | 0.756 | 0.878 | 0.502 | 0.839 | 0.763 | 0.711 |
| G2 | $p_1$ | 1.000 | 0.410 | 0.491 | 1.000 | 0.570 | 0.209 | 0.085 | 0.607 | 0.346 | 0.447 | 0.455 |
| | $p_2$ | 0.000 | 0.549 | 0.476 | 0.000 | 0.397 | 0.770 | 0.902 | 0.362 | 0.618 | 0.515 | 0.505 |
| S | $p_1$ | 1.000 | 0.315 | 0.347 | 1.000 | 0.892 | 0.266 | 0.138 | 0.736 | 0.237 | 0.365 | 0.284 |
| | $p_2$ | 0.000 | 0.647 | 0.609 | 0.000 | 0.095 | 0.713 | 0.847 | 0.243 | 0.728 | 0.588 | 0.683 |
| S2 | $p_1$ | 1.000 | 0.753 | 0.396 | 1.000 | 0.802 | 0.465 | 0.284 | 0.623 | 0.191 | 0.375 | 0.358 |
| | $p_2$ | 0.000 | 0.220 | 0.571 | 0.000 | 0.178 | 0.509 | 0.687 | 0.348 | 0.782 | 0.584 | 0.606 |
| H | $p_1$ | 1.000 | 0.911 | 0.392 | 0.995 | 0.535 | 0.626 | 0.410 | 0.546 | 0.252 | 0.307 | 0.386 |
| | $p_2$ | 0.000 | 0.073 | 0.566 | 0.004 | 0.423 | 0.347 | 0.561 | 0.412 | 0.706 | 0.640 | 0.568 |
| H2 | $p_1$ | 1.000 | 0.565 | 0.285 | 0.879 | 0.502 | 0.659 | 0.527 | 0.339 | 0.174 | 0.231 | 0.372 |
| | $p_2$ | 0.000 | 0.399 | 0.678 | 0.116 | 0.461 | 0.323 | 0.452 | 0.633 | 0.802 | 0.735 | 0.591 |
| BL | $p_1$ | 1.000 | 0.712 | 0.325 | 0.866 | 0.468 | 0.528 | 0.347 | 0.247 | 0.113 | 0.301 | 0.360 |
| | $p_2$ | 0.000 | 0.255 | 0.639 | 0.117 | 0.494 | 0.446 | 0.628 | 0.727 | 0.871 | 0.651 | 0.598 |
| ST | $p_1$ | 1.000 | 0.601 | 0.251 | 0.953 | 0.515 | 0.347 | 0.161 | 0.339 | 0.152 | 0.226 | 0.260 |
| | $p_2$ | 0.000 | 0.378 | 0.718 | 0.043 | 0.454 | 0.633 | 0.826 | 0.634 | 0.821 | 0.739 | 0.702 |
| C | $p_1$ | 1.000 | 0.615 | 0.369 | 0.999 | 0.769 | 0.286 | 0.177 | 0.640 | 0.206 | 0.352 | 0.321 |
| | $p_2$ | 0.000 | 0.357 | 0.590 | 0.001 | 0.210 | 0.690 | 0.805 | 0.328 | 0.762 | 0.606 | 0.640 |
| CA | $p_1$ | 1.000 | 0.797 | 0.384 | 0.994 | 0.618 | 0.431 | 0.228 | 0.532 | 0.259 | 0.344 | 0.365 |
| | $p_2$ | 0.000 | 0.176 | 0.575 | 0.005 | 0.350 | 0.544 | 0.750 | 0.435 | 0.706 | 0.608 | 0.595 |
| L | $p_1$ | 1.000 | 0.756 | 0.450 | 1.000 | 0.840 | 0.248 | 0.148 | 0.741 | 0.256 | 0.457 | 0.348 |
| | $p_2$ | 0.000 | 0.235 | 0.514 | 0.000 | 0.141 | 0.734 | 0.834 | 0.235 | 0.711 | 0.500 | 0.619 |
| LA | $p_1$ | 1.000 | 0.831 | 0.499 | 1.000 | 0.879 | 0.333 | 0.234 | 0.827 | 0.266 | 0.524 | 0.384 |
| | $p_2$ | 0.000 | 0.161 | 0.467 | 0.000 | 0.109 | 0.641 | 0.744 | 0.155 | 0.701 | 0.434 | 0.578 |

Table A.4: Win probabilities of player 1 ($p_1$) and player 2 ($p_2$) strategies extracted from the results of the runs against fixed strategies (no opponent (N), Greedy opponent (G,G2), Scorched Earth opponent (S,S2), Hunter opponent (H,H2)) and the coevolutionary runs (coevolutionary with standard settings (C), with asymmetric evaluation (CA), long run (L) and long run with asymmetric evaluation (LA). The player 1 strategies also contain the results of the run against the Block (BL) and Stripe (ST) opponent configurations. The predict *Functions* of the player 1 and the coevolutionary player 2 strategies were overwritten with a *Function* that always returns the last prediction. The probabilities are based on the results of 10000 matches.

| | | Player 2 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Player 1 | | N | G | G2 | S | S2 | H | H2 | C | CA | L | LA |
| N | $p_1$ | 1.000 | 0.921 | 0.363 | 0.994 | 0.518 | 0.368 | 0.276 | 0.555 | 0.211 | 0.293 | 0.366 |
| | $p_2$ | 0.000 | 0.062 | 0.597 | 0.004 | 0.443 | 0.613 | 0.702 | 0.405 | 0.759 | 0.656 | 0.591 |
| G | $p_1$ | 1.000 | 0.856 | 0.243 | 0.997 | 0.334 | 0.230 | 0.111 | 0.460 | 0.136 | 0.189 | 0.241 |
| | $p_2$ | 0.000 | 0.120 | 0.725 | 0.002 | 0.633 | 0.752 | 0.875 | 0.506 | 0.842 | 0.777 | 0.726 |
| G2 | $p_1$ | 1.000 | 0.423 | 0.488 | 1.000 | 0.566 | 0.203 | 0.078 | 0.606 | 0.347 | 0.470 | 0.455 |
| | $p_2$ | 0.000 | 0.533 | 0.478 | 0.000 | 0.395 | 0.783 | 0.914 | 0.358 | 0.620 | 0.490 | 0.508 |
| S | $p_1$ | 1.000 | 0.305 | 0.345 | 1.000 | 0.887 | 0.264 | 0.138 | 0.723 | 0.231 | 0.378 | 0.278 |
| | $p_2$ | 0.000 | 0.656 | 0.612 | 0.000 | 0.100 | 0.712 | 0.846 | 0.255 | 0.736 | 0.577 | 0.688 |
| S2 | $p_1$ | 1.000 | 0.758 | 0.388 | 1.000 | 0.798 | 0.457 | 0.291 | 0.619 | 0.187 | 0.380 | 0.352 |
| | $p_2$ | 0.000 | 0.217 | 0.576 | 0.000 | 0.181 | 0.518 | 0.685 | 0.352 | 0.784 | 0.579 | 0.608 |
| H | $p_1$ | 1.000 | 0.911 | 0.387 | 0.996 | 0.538 | 0.639 | 0.409 | 0.542 | 0.254 | 0.283 | 0.386 |
| | $p_2$ | 0.000 | 0.074 | 0.566 | 0.003 | 0.421 | 0.339 | 0.564 | 0.411 | 0.709 | 0.666 | 0.566 |
| H2 | $p_1$ | 1.000 | 0.606 | 0.301 | 0.877 | 0.412 | 0.676 | 0.553 | 0.345 | 0.180 | 0.194 | 0.402 |
| | $p_2$ | 0.000 | 0.354 | 0.661 | 0.116 | 0.552 | 0.308 | 0.423 | 0.625 | 0.800 | 0.767 | 0.557 |
| BL | $p_1$ | 1.000 | 0.710 | 0.332 | 0.863 | 0.463 | 0.530 | 0.347 | 0.252 | 0.108 | 0.259 | 0.359 |
| | $p_2$ | 0.000 | 0.257 | 0.631 | 0.122 | 0.501 | 0.442 | 0.627 | 0.723 | 0.873 | 0.695 | 0.597 |
| ST | $p_1$ | 1.000 | 0.604 | 0.248 | 0.952 | 0.520 | 0.343 | 0.164 | 0.330 | 0.152 | 0.214 | 0.257 |
| | $p_2$ | 0.000 | 0.372 | 0.721 | 0.043 | 0.450 | 0.636 | 0.819 | 0.640 | 0.825 | 0.755 | 0.708 |
| C | $p_1$ | 1.000 | 0.620 | 0.368 | 0.999 | 0.762 | 0.285 | 0.172 | 0.640 | 0.206 | 0.350 | 0.324 |
| | $p_2$ | 0.000 | 0.351 | 0.593 | 0.001 | 0.217 | 0.691 | 0.812 | 0.331 | 0.760 | 0.610 | 0.638 |
| CA | $p_1$ | 1.000 | 0.800 | 0.400 | 0.995 | 0.617 | 0.432 | 0.232 | 0.523 | 0.258 | 0.336 | 0.364 |
| | $p_2$ | 0.000 | 0.175 | 0.562 | 0.004 | 0.351 | 0.545 | 0.748 | 0.439 | 0.706 | 0.623 | 0.594 |
| L | $p_1$ | 1.000 | 0.707 | 0.440 | 1.000 | 0.842 | 0.244 | 0.146 | 0.736 | 0.248 | 0.470 | 0.344 |
| | $p_2$ | 0.000 | 0.278 | 0.523 | 0.000 | 0.142 | 0.735 | 0.836 | 0.240 | 0.720 | 0.488 | 0.618 |
| LA | $p_1$ | 1.000 | 0.833 | 0.501 | 1.000 | 0.882 | 0.340 | 0.241 | 0.822 | 0.265 | 0.537 | 0.389 |
| | $p_2$ | 0.000 | 0.159 | 0.465 | 0.000 | 0.106 | 0.632 | 0.737 | 0.159 | 0.701 | 0.423 | 0.582 |

# Bibliography


[1] Antwars competition, April 2010. `http://www.sigevo.org/gecco-2007/competitions.html#c3`.

[2] Christoph Salge, Christian Lipski, Tobias Mahlmann, and Brigitte Mathiak. Using genetically optimized artificial intelligence to improve gameplaying fun for strategical games. In *Sandbox '08: Proceedings of the 2008 ACM SIGGRAPH symposium on Video games*, pages 7–14, New York, NY, USA, 2008. ACM.

[3] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, December 1992.

[4] Tiago Francisco and Gustavo Miguel Jorge dos Reis. Evolving combat algorithms to control space ships in a 2d space simulation game with co-evolution using genetic programming and decision trees. In *GECCO '08: Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, pages 1887–1892, New York, NY, USA, 2008. ACM.

[5] Tiago Francisco and Gustavo Miguel Jorge dos Reis. Evolving predator and prey behaviours with co-evolution using genetic programming and decision trees. In *GECCO '08: Proceedings of the 2008 GECCO conference companion on Genetic and evolutionary computation*, pages 1893–1900, New York, NY, USA, 2008. ACM.

[6] Sean Luke and Lee Spector. Evolving teamwork and coordination with genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 150–156, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[7] Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James Hendler. Co-evolving soccer softbot team coordination with genetic programming. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Computer Science*, pages 398–411. Springer-Verlag, 1997.

[8] Ami Hauptman. Gp-endchess: Using genetic programming to evolve chess endgame players. In Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano van Hemert, and Marco Tomassini, editors, *Proceedings of 8th European Conference on Genetic Programming (EuroGP2005)*, volume 3447 of *Lecture Notes in Computer Science*, pages 120–131. Springer, 2005.

[9] Timothy Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[10] Markus F. Brameier and Wolfgang Banzhaf. *Linear Genetic Programming (Genetic and Evolutionary Computation)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[11] Julian F. Miller and Peter Thomson. Cartesian genetic programming. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *Lecture Notes in Computer Science*, pages 121–132, Edinburgh, 15-16 April 2000. Springer-Verlag.

[12] Riccardo Poli. Parallel distributed genetic programming. In David Corne, Marco Dorigo, Fred Glover, Dipankar Dasgupta, Pablo Moscato, Riccardo Poli, and Kenneth V. Price, editors, *New ideas in optimization*, pages 403–432. McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999.

[13] Peter A. Whigham. Grammatically-based genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA, 9 July 1995.

[14] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3:199–230, 1994.

[15] John R. Woodward and Ruibin Bai. Why evolution is not a good paradigm for program induction: a critique of genetic programming. In *GEC '09: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 593–600, New York, NY, USA, 2009. ACM.

[16] Lorenz Huelsbergen. Finding general solutions to the parity problem by evolving machine-language representations. In Morgan Kaufmann, editor, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 158–166. University of Wisconsin, Madison, Wisconsin, USA, July 1998.

[17] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evol. Comput.*, 14(3):309–344, 2006.

[18] Sara Silva and Ernesto Costa. Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10(2):141–179, 2009.

[19] Daniel W. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena*, 42(1-3):228 – 234, 1990.

[20] Mitchell A. Potter and Kenneth A. de Jong. A cooperative coevolutionary approach to function optimization. In *PPSN III: Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature*,

volume 866 of *Lecture Notes in Computer Science*, pages 249–257, London, UK, 1994. Springer-Verlag.

[21] Peter J. Angeline and Jordan B. Pollack. Competitive environments evolve better solutions for complex tasks. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270, University of Illinois at Urbana-Champaign, 17-21 July 1993. Morgan Kaufmann.

[22] Thomas Miconi. Why coevolution doesn't "work": Superiority and progress in coevolution. In Leonardo Vanneschi, Steven Gustafson, Alberto Moraglio, Ivanoe De Falco, and Marc Ebner, editors, *EuroGP*, volume 5481 of *Lecture Notes in Computer Science*, pages 49–60. Springer, 2009.

[23] Edwin D. de Jong, Kenneth O. Stanley, and R. Paul Wiegand. Introductory tutorial on coevolution. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 3133–3157, New York, NY, USA, 2007. ACM.

[24] John Nash. Non-cooperative games. *The Annals of Mathematics*, 54(2):286–295, September 1951.

[25] Sevan G. Ficici and Anthony Bucci. Advanced tutorial on coevolution. In *GECCO '07: Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, pages 3172–3204, New York, NY, USA, 2007. ACM.

[26] Christopher D. Rosin and Richard K. Belew. New methods for competitive coevolution. *Evolutionary Computation*, 5(1):1–29, 1997.

[27] Kenneth O. Stanley and Risto Miikkulainen. The dominance tournament method of monitoring progress in coevolution. In Alwyn M. Barry, editor, *GECCO 2002: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference*, pages 242–248, New York, 8 July 2002. AAAI.

[28] Sevan G. Ficici and Jordan B. Pollack. A game-theoretic memory mechanism for coevolution. In *GECCO'03: Proceedings of the 2003 international conference on Genetic and evolutionary computation*, volume 2723 of *Lecture Notes in Computer Science*, pages 286–297, Berlin, Heidelberg, 2003. Springer-Verlag.

[29] Edwin D. de Jong. The incremental pareto-coevolution archive. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO04*, volume 3102 of *Lecture Notes in Computer Science*, pages 525–536. Springer Verlag, 2004.

[30] Nathan Williams and Melanie Mitchell. Investigating the success of spatial coevolution. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 523–530, New York, NY, USA, 2005. ACM.

[31] Mike J. Keith and Martin C. Martin. Genetic programming in c++: implementation issues. *Advances in genetic programming*, pages 285–310, 1994.

[32] Wojciech Jaskowski, Krzysztof Krawiec, and Bartosz Wieloch. Winning ant wars: Evolving a human-competitive game strategy using fitnessless selection. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Esparcia-Alcázar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *EuroGP*, volume 4971 of *Lecture Notes in Computer Science*, pages 13–24. Springer, 2008.