Improving the Performance of Heuristic Spam Detection using a Multi-Objective Genetic Algorithm

James Dudley

This report is submitted as partial fulfilment of the requirements for the Honours Programme of the School of Computer Science and Software Engineering, The University of Western Australia, 2007
Abstract

With today’s reliance on email, spam is not just annoyance, but presents a real cost to companies and individuals. Several methods of spam detection exist, but each has certain weaknesses. Heuristic spam filters address these weaknesses by using several different methods at once; however the complexity of these filters makes it difficult to optimise their performance. Complicating the problem is that spam detection has more than one objective: as well as catching as much spam as possible, a spam detector must limit the amount of legitimate email that is incorrectly flagged as spam.

This dissertation investigates the use of genetic algorithms to optimise heuristic spam filter performance. The use of multi-objective genetic algorithms to find a set of trade-off solutions is also examined.

This is done by implementing both a single objective and multi-objective genetic algorithm over SpamAssassin, which is a popular heuristic spam filter.

Positive results are obtained, with both algorithms outperforming the current method of optimisation used by SpamAssassin, and the multi-objective algorithm producing a front of solutions that allows a solution to be selected based on the desired trade-off between the two objectives of catching spam and losing legitimate email.

**Keywords:** Spam detection, optimisation, genetic algorithms, multi-objective genetic algorithms, SpamAssassin

**CR Categories:** 1.2.1 [Artificial Intelligence]: Applications and Expert Systems
Acknowledgements

First of all I would like to thank my supervisor, Dr. Luigi Barone, for all his help and guidance throughout the year.

Secondly, a big thanks to my parents, for their support. Not just this year but throughout my whole life thus far. They also ensured I was fed on the rare occasions I emerged from my room during the final weeks of writing my dissertation.

Finally, thanks to all my friends for helping me get through the year. Special thanks go to Reuben and Nick, who shared their own experiences of writing a dissertation and convinced me that it could be done.
Contents

Abstract ............................................................................................................................................ 2
Acknowledgements ............................................................................................................................ 3
List of Tables .................................................................................................................................. 6
List of Figures .................................................................................................................................. 7
1 Introduction .................................................................................................................................. 8
2 Background ................................................................................................................................... 9
  2.1 Spam Detection ........................................................................................................................ 9
     2.1.1 Origin Based Filters ........................................................................................................ 9
     2.1.2 Content Based Filters ..................................................................................................... 10
     2.1.3 Collaborative Filters ...................................................................................................... 11
     2.1.4 Heuristics Based Filters .................................................................................................. 11
  2.2 Genetic Algorithms .................................................................................................................. 12
     2.2.1 Single-objective GAs ...................................................................................................... 12
     2.2.2 Multi-objective GAs ........................................................................................................ 13
  2.3 SpamAssassin ............................................................................................................................ 14
     2.3.1 The Scoring System ......................................................................................................... 14
     2.3.2 Classification of Tests ...................................................................................................... 14
     2.3.3 How Scores are Assigned ............................................................................................... 15
3 Method and Implementation ........................................................................................................ 17
  3.1 Algorithms Developed ............................................................................................................. 17
     3.1.1 Structure of the Genome ................................................................................................. 17
     3.1.2 Genetic Algorithm Operators .......................................................................................... 17
     3.1.3 The Multi-Objective GA ................................................................................................ 19
     3.1.4 The Memetic Algorithm ................................................................................................ 19
  3.2 Testing the Algorithms ............................................................................................................. 20
     3.2.1 Corpus of Emails Used To Evaluate Fitness ................................................................. 20
     3.2.2 Fitness Evaluations as a Measure of Runtime ............................................................... 20
     3.2.3 Separate Training and Test Corpuses ............................................................................ 21
  3.3 Implementation Details ............................................................................................................. 21
     3.3.1 Evaluating Fitness ........................................................................................................... 21
     3.3.2 Training and Test Corpuses ........................................................................................... 21
3.3.3 Restriction to Local Tests Only ................................................................. 22
3.3.4 Modification of the SpamAssassin GA .................................................. 22

4 Experiments and Results ............................................................................... 23
  4.1 Selecting the Optimum Settings for the Genetic Algorithm .................. 23
    4.1.1 Optimum Parameters for the GA ..................................................... 24
    4.1.2 Optimum Parameters for the MOGA ........................................... 24
  4.2 Distribution of Initial Generation ......................................................... 25
  4.3 Memetic Algorithm .............................................................................. 27
    4.3.1 Performance of the Memetic Algorithm ....................................... 27
    4.3.2 Using a Local Search to Identify Unnecessary Tests .................... 28
  4.4 Performance of the Algorithms ............................................................ 29
    4.4.1 Single-Objective Genetic Algorithm .......................................... 29
    4.4.2 Multi-Objective Genetic Algorithm ............................................. 32

5 Conclusion .................................................................................................. 38

Appendix A: Original Research Proposal .................................................... 39

Appendix B: Details of Experiments Performed to Identify Optimum Parameters for the GA .... 43

Bibliography .................................................................................................. 52
List of Tables

Table 1: Initial values chosen for GA Parameters .......................................................................................... 23
Table 2: Optimum Parameters for the GA ........................................................................................................ 24
Table 3: Optimum Parameters for the MOGA ................................................................................................. 24
Table 4: Difference in Scores for ALL_TRUSTED test between solutions with high false positives and solutions with low false positives .......................................................................................... 26
Table 5: Detail of Solution generated by GA before and after zeroing ............................................................... 28
Table 6: Performance of Solutions produced by my GA and the SpamAssassin GA ....................................... 31
Table 7: Performance of Solutions Produced by the MOGA and the SpamAssassin GA ................................. 35
List of Figures

Figure 1: Distribution of Initial Generation of Genetic Algorithm .................................................. 25
Figure 2: Difference in mean score for each test between Set A and Set B ........................................ 26
Figure 3: Comparison of Performance of Genetic and Memetic Algorithm ........................................ 27
Figure 4: Genome of solution generated by GA before and after zeroing ........................................ 28
Figure 5: Performance of Single-Objective Genetic Algorithm compared to SpamAssassin’s Genetic Algorithm .................................................................................................................. 29
Figure 6: Comparison of performance of Single-Objective GA on Training and Test Corpus ............ 30
Figure 7: Genomes for solutions produced by my Single-Objective GA and the SpamAssassin GA .... 31
Figure 8: Performance of Multi-Objective Genetic Algorithm, compared to SpamAssassin’s Genetic Algorithm .................................................................................................................. 33
Figure 9: Comparison of performance of Multi-Objective GA on Training and Test Corpus ............ 34
Figure 10: Comparison of Solutions produced by the Multi-Objective GA .......................................... 35
Figure 11: Genomes for Solutions produced by the MOGA and the SpamAssassin GA ................. 36
CHAPTER 1

Introduction

As the internet continues to become more pervasive in today’s society, the use of electronic mail systems (email) has also continued to increase. Today email has grown into a significant communication channel for both business and personal use. A study of American workers in 2003 showed that email use at work had more than doubled in the past two years, and over half of all study participants considered email essential to their work (1).

Unsolicited bulk email – known as spam – has emerged as a major problem for the electronic mail system. More simply, spam is any email received that the user does not want and did not ask for. This makes up a surprisingly high proportion of all email - a study conducted in 2005 found it to be 60% (2), and another found it had risen to 80% today (3). It also appears to be increasing – as the figures above suggest. The costs associated with Spam are also high: a conservative estimate put the total cost at US$10 Billion for US companies alone in 2003 (4). This cost consists of three main components: loss of user productivity, consumption of IT resources and help desk costs.

A variety of methods have been developed to detect spam. However, none of these spam filters are perfect. Each has certain weaknesses that cause it to either miss spam, or incorrectly identify ham (legitimate email) as spam, in certain cases. To address these flaws in spam detection, a method known as heuristic spam detection was developed. This allows several different spam detection methods to be utilised together. A heuristic spam filter performs a number of tests on each message. Each test is given a score, and typically a total score for the message is obtained by summing the scores for all tests hit by that message. If the score for the message exceeds a certain threshold, the message is marked as spam.

The use of heuristic based spam filters introduces a new problem: how to assign the correct score to each test, so that the performance of spam filtering is optimised. As a large number of tests are usually performed, and the optimum score for each is interdependent, this problem is not trivial. Furthermore, spam filtering involves a trade-off between the amount of spam caught by the filter, and the amount of ham incorrectly identified as spam. Different users will have different preferences for each objective, so no single solution will be suitable for all cases.

This dissertation investigates the use of genetic algorithms in solving this problem. Genetic algorithms use an imitation of natural evolution to solve problems, and have proven useful for complex problems such as this. An extension of genetic algorithms, called multi-objective genetic algorithms, are able to produce a number of solutions that provide trade-offs between different objectives. This would be especially useful for the problem of spam detection, as it would allow each user to select the solution which best matches their preference for ham or spam.
CHAPTER 2

Background

2.1 Spam Detection

To automatically detect and remove spam as it is received, a spam filter is used. Spam Filters work by first receiving part (or all) of the email and then analysing it in some way to decide whether it is ham (i.e. legitimate email) or spam.

The performance of a spam filter can be measured by the number of false positives (ham which is incorrectly marked as spam) and false negatives (unidentified spam) that it generates. An ideal spam filter would identify all email correctly, so would have zero false positives and negatives. Realistically this is not possible, and some tradeoff between the number of false positives and false negatives must be made.

Another consideration in spam filter performance is the time taken to process each message. This is especially significant for spam filters installed at the mail server level, which may be required to process many thousands of messages an hour. If a filter is not able to process a message within a few seconds (or cannot process enough messages concurrently to achieve the required throughput) then it will be impractical for such use, regardless of its detection rates.

Spam filters can be divided into a number of broad categories based on the method used to filter spam, including origin based, content based and collaborative filters.

2.1.1 Origin Based Filters

Origin based filters use the network information contained in emails to detect spam. This generally means the IP address, or source email address of an email. The main types of origin based filters are black lists, whitelists and reverse DNS lookups.

Blacklists

Blacklisting was pioneered by the Mail Abuse Prevention System (MAPS), which has now become Trend Micro’s RBL+ (5). Blacklists can be used to filter email received from mail servers or domains that have previously sent spam, or are suspected of doing so. Blacklists store the IP address of known and suspected spammers in a central database – the blacklist. The email address is not used, as a fake source email address can be easily be used by spammers. When an email is received, its IP address can be checked against the blacklist.

The main problem with spam filtering using a blacklist is timeliness. Most blacklists are only updated every 30-60 minutes (6), (3). As spammers send large amounts of spam at once, a lot of spam will get through the system before the blacklist is updated. Spammers are also able to use techniques such as zombie networks to circumvent blacklists by sending spam through compromised computers. This can also create false positives when legitimate users try to send email through their compromised computer.
Whitelists
Whitelists are used to define trusted addresses, meaning that any email received from those addresses will be assumed to be legitimate, and any other is spam. These are usually much smaller and easier to maintain than blacklists. However, they are not effective as a stand-alone filter, because they result in far too many false positives: any emails not from sources on the whitelist will automatically be rejected. Whitelists generally use the source email address as the basis to filter emails. As this can easily be forged, a spammer can bypass the spam filter if they can determine an address on the whitelist and use this address in their spam. For this reason, whitelists cannot be publically shared like blacklists, and common addresses, for example receipt@ebay.com, should be kept off the whitelist.

Reverse DNS Lookups
The domain name system (DNS) can be used to help identify spam. A DNS query can be run on the source IP address to obtain the host name. If the host name cannot be found this suggests the message is spam, as many spammers deliberately use misconfigured hosts in order to disguise the source of spam. However, it has no way of identifying spam sent from properly configured servers. Also, legitimate servers may also be misconfigured, or choose not to register a name with the DNS, so it cannot be relied upon as the sole evidence of spam.

2.1.2 Content Based Filters
Content based filters detect spam by analysing the contents of inbound mail for suspicious characteristics. The logic behind this is that the content of an email is ultimately what defines it as spam or ham. The difficulty is writing a program that can differentiate between the two, and avoid the tricks that spammers use to trip it up.

Header Analysis
Analysis of the headers of an email can help to identify spam. Email headers provide information on the email, such as the subject, and detail on where it has been. One simple way is to search for headers that deviate from specification (7). Many spammers spoof headers to prevent investigators from tracking them down, so malformed headers are a good indication of spam. It is also more efficient than examining the entire body of an email as the headers are much shorter. However, this cannot be relied upon as a standalone solution.

Keyword, Keyphrase and Pattern Matching
One of the first methods used to detect spam was keyword and keyphrase matching (7). This method searches the subject and body of an email for certain words or phrases that appear regularly in spam – for example ‘Viagra’. Simple tricks employed by spammers such as deliberately misspelling words, or inserting HTML comments between letters of the word, can easily bypass spam filters based on this method.

Pattern matching extends the idea of keyword searches to use regular expressions. Instead of searching for a particular word or phrase, pattern matching searches an email for a pattern which matches the specified regular expression. This is a much more powerful technique. Regular expressions are less easy to get around than keywords, and are able to identify words that are obscured by misspelling or other tricks. They can also be used to search for other characteristics in the message that suggest it may be spam – for example a large number of HTML comments, excessive use of capital letters, or strange use of quotation marks.
While pattern matching is much more effective at identifying spam than keyword or keyphrase matching, no single message characteristic can be relied upon to consistently distinguish spam from legitimate email, and they must be updated regularly as spammers adapt to them and employ different techniques. The risk of false positives is also high, and the regular expressions used must be crafted carefully to ensure they avoid this.

**Bayesian Filters**

Bayesian filters work use statistics to differentiate between legitimate email and spam (8). It works by dividing an email into a set of features (usually words) and determining how many times each feature has been recorded in normal email and spam. The probabilities for each feature are then combined into a final probability, which estimates the likelihood the email is spam or legitimate email. If this probability exceeds a certain limit, then the email is assumed to be spam.

A disadvantage of Bayesian filters is that they must be trained on a large number of emails before they can be used. Also, whilst they are very effective when tuned to an individual users’ email (7), they are less effective when run at the mail server level, where the diversity of different user’s email makes it harder to distinguish statistically between spam and legitimate email (9).

**2.1.3 Collaborative Filters**

Collaborative filters use collective feedback from a group of users to identify spam. This exploits the fact that spam is sent in bulk; so many people will receive the same spam email. Therefore once one person has identified an email as spam (either manually or using another spam filter), any other user that receive the same email can discard it as spam, without having to check it themselves. Rather than comparing emails directly, which would be error prone and a privacy risk, a user computes a checksum (or hash) for each email, and compares that to the checksum for known spam. If a match is found then the email is spam and so can be discarded. Two popular collaborative filters are Vipul’s Razor (10), and the Distributed Checksum Clearinghouse, or DCC (11).

Collaborative spam filters suffer from a number of problems. One common technique used by spammers is to make small random changes to each message – this gives each message a different checksum, so they will not be recognised as the same message. Both Razor and DCC try to address this by using ‘fuzzy checksums’ in which the message is broken up and the checksum calculated on the most significant parts. However this does not completely solve the problem (12).

The main problem with collaborative filters is the same as the one affecting blacklists: timeliness. As most spam is sent in large batches, most of it will arrive within a short time period. However, it will not be reported as spam by the collaborative filter until a user has manually identified it as such. By this time, many others will have already received it.

**2.1.4 Heuristics Based Filters**

All the various methods for filtering spam have strengths and weaknesses. Whilst they all are successful to some degree, their weaknesses mean there is no single solution to identifying spam. The best approach is to use a combination of various methods. By utilising several methods, the spam missed by one method is likely to be identified by another; it is much more difficult for a spammer to fool one method without triggering the others. This has lead to the development of a fourth type of filter – heuristics based filters.
Heuristics based filters perform a large number of tests on an email, to decide whether it is spam or ham. Each test is given a score, and then a total score for the email is typically generated by summing the scores for all tests that were hit by the email. If the email’s score is over a certain threshold, it is then declared to be spam. Using a heuristic based filter allows many methods of spam filtering to be used, resulting in better performance than any single method by itself. An example of a heuristics based filter is SpamAssassin (13), which utilises all the major methods of spam detection, including blacklisting, whitelisting, reverse DNS lookups, header analysis, pattern matching, Bayesian filtering, and collaborative filters such as Vipul’s Razor and the DCC.

By combining a number of different methods of spam detection heuristic based methods are able to outperform any single method on its own (9). However, a new problem emerges: assigning the optimum score for each test. Due to the large number of rules employed, mathematical techniques are too unwieldy and an exhaustive search is too time-consuming. Therefore, other optimisation techniques need to be used. One technique that could prove useful is Genetic Algorithms.

2.2 Genetic Algorithms

2.2.1 Single-objective GAs

Genetic algorithms (GAs) are a computer-based method of optimisation, which use a process imitating natural evolution in order to solve a problem. The basic process is as follows: First, a number of possible solutions (a population) are generated randomly. These solutions are then tested, and the best solutions are selected and the others discarded. The selected solutions are then recombined in some way to create a population of child solutions, that combine characteristics from one or more of the parent solutions and are also randomly altered, or mutated slightly, and thus while similar to their parents, they are not identical. Depending on the selection strategy used, the new population is then formed from either a combination of both parent and child solutions (an elitist, or plus, selection strategy), or from child solutions only (a comma selection strategy). This new population is then tested, and again the best solutions are selected for further breeding. This process continues until an acceptable solution to the problem has been reached.

GAs are most useful in optimisation type problems for which there is no obvious way to use more traditional problem solving techniques (14). All that is required to tackle the problem using a GA is a way of representing a solution to the problem (the genome of a solution), and a way of testing and ranking potential solutions (the fitness function). Given this information, a GA can be constructed to create and evolve a set of solutions until a good solution to the problem is reached.

This characteristic makes GAs useful for finding the optimum scores for a heuristic spam filter: a genome of a possible solution is constructed as a set of scores for each rule, and the fitness of a possible solution can be tested by measuring the performance of the filter when those scores are used.

One problem with GAs is that, whilst they can quickly find a good solution to an optimisation problem, they often struggle to find the perfect solution, as they are not particularly effective at local optimisation (15). This problem is addressed by Memetic Algorithms.

A Memetic Algorithm (MA) combines the principle of Genetic Algorithms with a local search, which is used to perform local optimisation – something that GAs struggle with. MAs use a local search to
look for solutions, close to the solutions in the population, which provide a better result. This can help the algorithm to move from a good solution to the optimum solution. MAs have been shown to be more effective than GAs for a number of problems (16).

Many optimisation problems require the simultaneous optimisation of multiple objectives. These objectives are often conflicting – improving the performance for one objective requires the trade-off in the performance of the other. In fact, spam detection itself is a multi-objective problem, with performance measured by both the rate of false positives and false negatives, as discussed earlier. These two objectives are in conflict, as increasing the strictness of the filter to stop more spam is likely to also cause more legitimate email being marked as spam. Similarly, relaxing the filter to ensure all legitimate email gets through will also lead to more spam escaping detection.

It is in fact possible to reduce any multi-objective problem (MOP) to a single objective, which can then be optimised using single objective optimisation techniques (17). This can be done by combining all the objectives into a single function, weighting each based on its perceived importance. The resulting function can then be treated like a single objective, and optimised in that way. This is the approach generally taken in spam detection – false positives and false negatives are combined to form a measure of overall fitness, with false positives given a higher weighting because it is generally considered more important to avoid missing a legitimate email than it is to avoid all spam.

The problem with this approach is that it does not consider the possible trade-offs that can be made between objectives. For example, the importance of the two objectives of spam detection can vary widely between users. A business user who would like to ensure they receive all emails sent by their customers would be willing to accept some spam to ensure all this email gets through. The email account of a child, however, should be protected from the offensive content found in many spam emails.

2.2.2 Multi-objective GAs
A better method of optimising a MOP is to extend GAs to consider each objective separately. This allows the GA to evolve the set of optimal solutions for the problem. This approach is known as a multi-objective GA (MOGA). MOGAs operate in a similar way to normal GAs. However, instead of using a single fitness function to compare solutions, a multi-objective GA calculates a fitness for each objective.

To identify the set of optimal solutions for a MOP, the notion of Pareto optimality is adopted. A Pareto optimum is a solution to a MOP for which no other feasible solution exists that improves performance for one objective without simultaneously decreasing it for another objective. The set of all such optimal solutions forms the Pareto front.

In order to rank solutions in a MOGA, the concept of domination is introduced. A solution is said to dominate another solution if it is Pareto optimal with respect to the other solution (i.e. it outperforms the other solution in at least one objective and is at least as good as it for the others). To produce a ranking of solutions, each solution is compared to all the other solutions in the population, and is given a rank (known as the Pareto rank) based on the number of solutions in the population that dominate it. This produces a set of fronts, each front consisting of solutions of the same rank. The front of solutions with a rank of zero forms the Pareto front for the population.
However, this method only produces a partial ranking of solutions. A means of selecting between solutions of the same Pareto rank is also required. One solution is provided in (18) – which introduces a MOGA known as NSGA-II. NSGA-II uses the crowding comparison operator. This sorts solutions based on the crowding distance - an estimation of the crowding of solutions. Less crowded solutions are ranked higher, with the aim of ensuring the best coverage of the objective space.

For each objective the front of solutions is ranked according to the fitness for that objective. The solutions with the highest and lowest fitness are assigned an infinite crowding distance for that objective, to ensure they are always selected. Each intermediate solution is assigned a crowding distance equal to the difference between the fitness for the two adjacent solutions. The total crowding distance for each solution is calculated by summing the crowding distance for each objective. The solutions can then be ranked based on their total crowding distance, solutions with a higher crowding distance are ranked higher and selected over solutions with a lower crowding distance.

2.3 SpamAssassin

2.3.1 The Scoring System
SpamAssassin (13) works by running a number of tests on each email. Each test has a corresponding score. This score can be any real number – it can be either positive or negative, and its range is not restricted. Each score is rounded to three decimal places by SpamAssassin.

The total score for an email is computed by adding up all the scores for tests that were hit on that email. A threshold score is defined – If the total score for an email is equal to or exceeds this threshold then the email is classified as spam. If the total score is lower than the threshold then the email is classified as ham. The default value for the threshold score is 5.0; this can be changed in order to adjust the severity of spam detection.

The consequences of this scoring system are as follows – tests with a negative score will reduce the overall score for an email if that test is hit, making the email less likely to be classified as spam. On the other hand, tests with a positive score will increase the overall score for an email if that test is hit, making the email more likely to be classified as spam. Tests with a score above the threshold value will result in an email immediately being classified as spam if that is hit (assuming no tests with negative scores are also hit).

Tests with a score of zero will contribute nothing to the total score for an email, regardless of whether they are hit or not. Therefore, SpamAssassin does not run any test if it has a score of zero, so setting a test’s score to zero means the test will not be run.

2.3.2 Classification of Tests
SpamAssassin separates tests into three categories: local tests, network tests, and bayes tests. Local tests are all tests that can be performed without a connection to the internet - for example tests on message content, and whitelisting. Network tests are those tests that require a network connection – for example DNS tests, and collaborative tests such as DCC and Razor. Bayes tests are those that use SpamAssassin’s inbuilt Bayesian filter.

Depended on the required use, SpamAssassin can be configured to run only local tests, to run local tests and network tests but no bayes tests, to run local tests and bayes tests but no network tests, or
run all three types of tests together. The best score to use for each test will vary depending on which other tests are being run. For example, the scores for local tests should probably be lowered when network tests and bayes tests are also run, as running these other tests mean that the local tests are not as important. For this reason, SpamAssassin defines four different scores for each test, one for each possible configuration, and selects the correct one depending on which types of tests are being run.

2.3.3 How Scores are Assigned
SpamAssassin comes with pre-assigned scores, which are trained on a corpus of known spam and ham emails maintained by developers. SpamAssassin originally used a GA to train the scores. However, this was replaced in version 3.0 by a Perceptron Learner, which is a simple type of neural network. The main reason given for this change was that the GA (which was developed a number of years previously) had not been maintained, and by their own admission the current development team did not understand how it worked (19). Indeed, the Perceptron Learner did not produce better results, although it did run faster. With the 3.2.0 release of SpamAssassin, the Perceptron learner no longer worked correctly, and so the GA is now used again (20), although there is no indication that it is still being maintained. The basic process used by SpamAssassin to assign scores is described below.

First, a mass-check is run. This is a script which runs SpamAssassin on an entire corpus of emails, and records which tests were hit by each email in the corpus.

The output from the mass-check is then used to define a score-range for each test. The score-range of a test is restricted based on the total number of emails that hit that test and the hit rate on spam compared to ham emails. Scores for tests that are hit by more spam than ham are restricted to positive ranges, and scores for tests that are hit by more ham than spam are restricted to negative ranges. The magnitude that a score can grow to is determined by the number of emails that hit the test, and the difference in the hit-rate for spam and ham emails. Scores for tests that are hit by more messages, and for which there is a significant difference in the hit-rate for spam and ham emails are allowed to grow larger, on the basis that they are better at distinguishing between spam and ham. Also, no score is allowed to exceed the threshold value -- so no one test can cause an email to be classified as spam.

Using the output of the mass-check, the GA is then used to train the scores to provide the best spam detection performance across the corpus of emails from the mass-check. The current scores for each rule are used as a starting point, and scores are only allowed to vary within the ranges for each rule that were determined in the previous step.

To evaluate the performance of solutions, the SpamAssassin GA measures how far, on average, messages resulting in a false positive are from being correctly recognised as ham, and messages resulting in a false negative are from being correctly recognised as spam. For false positives, this is the amount by which the total score for the message exceeds the threshold, causing the message to be incorrectly identified as spam. Conversely, for false negatives, this is the amount by which the total score is under the threshold, causing the message to be incorrectly identified as ham. The fitness function is a weighted sum of these two values, in which the distance for false positives is multiplied by a factor of 10.
Fitness = 10 \times (\text{average score of FP} - \text{Threshold score}) \\
+ (\text{Threshold score} - \text{average score of FN})

By weighting false positives their effect on the fitness of a solution is increased compared to false negatives – this is done as it is generally considered more important to avoid false positives (which result in ham email being lost) than it is to avoid false negatives (which simply result in a spam email getting through). The SpamAssassin GA attempts to minimise this fitness function – as this means that the errors the solution makes in detecting spam are closer to being corrected.
CHAPTER 3

Method and Implementation

This project involved developing a genetic algorithm, memetic algorithm and multi-objective genetic algorithm to train the scores for SpamAssassin, hopefully resulting in an improvement in performance over the existing method that SpamAssassin uses to train scores. The details of each algorithm, how they were tested, and the implementation are detailed in this section.

3.1 Algorithms Developed

3.1.1 Structure of the Genome
The problem to be solved by the genetic algorithms is to optimise the scores for each rule run by SpamAssassin, so that its spam detection performance is maximised. The required genome to represent a single solution is relatively simple in structure. Each gene of a solution is simply a single real number – representing the score for a single test. However, due to the large number of tests used by SpamAssassin, the resulting genotype is relatively large.

3.1.2 Genetic Algorithm Operators

Initial Seeding of Population
Instead of using the existing scores for each rule as a starting point (like the GA used by SpamAssassin does), the decision was made to generate the initial population randomly. This provides a diverse set of solutions to evolve from, and avoids prematurely restricting the search space of the GA.

To randomly generate a solution, each gene (i.e. score) is randomly generated with a normal distribution, with a mean of zero, and a standard deviation equal to the threshold score (of 5.0). This produces a good range of scores, with most initialised below the threshold score, but some larger. It also produces an equal number of positive and negative scores.

Selection
A simple ranking based selection method was used for the single-objective GA. The population is first sorted by fitness, and then the solutions with the best fitness are selected to produce the new population. The number of solutions selected is determined by the selection level specified at the start of the GA. The effect that different selection levels have on the performance of the GA is examined in the results section of this paper.

The GA was constructed to allow either a plus selection strategy or a comma selection strategy to be used. The type of selection strategy to be used is specified at the start of the GA. If a plus strategy is chosen, then the solutions selected from the current population are added (unaltered) to the next generation, and then, to achieve the required population size, the remaining individuals are created by recombining and mutating the selected solutions.

If a comma strategy is used, then the new generation is made up of only the solutions generated from the selected solutions. The selected solutions themselves are discarded after they have been
used to produce the new solutions. There are advantages and disadvantages to both approaches (21). The best strategy to use for this problem is investigated in the results section.

**Recombination**

To produce a new solution, first the parents are chosen randomly from the solutions selected from the population. This method ensures that no solution is recombined with itself, and there is a uniform probability of choosing each solution.

An n-point crossover function was used to recombine solutions. Each score in the genome is chosen randomly from one of the parents. Multi-parent recombination was implemented – the number of parents that are recombined to form a new solution is specified at the start of the GA. This number of parents to use in recombination can be set to one. In this case, no recombination is performed, and new solutions are generated through mutation alone. The effect of varying the number of parents used to form new solutions is investigated in the results section.

**Mutation**

Each gene of a solution is mutated with a constant probability, determined by the mutation rate which is specified at the start of the GA. The mutation function used is similar to the method used to randomly seed solutions for the initial population. A score is mutated by randomly generating a new score using a normal distribution centred on the current value of the score. However, unlike the random seeding, which uses a standard deviation that is fixed at the threshold score, the standard deviation of mutation can be varied. This allows the size of mutation to be varied: a larger standard deviation will result in generally larger mutations, and vice versa. The results section investigates the effects of varying both the mutation rate and standard deviation of mutation.

**Fitness Function used**

The fitness function used to rank solutions is a weighted average of the percentage of false positives and false negatives that SpamAssassin produces when using the scores specified by a solution. This is shown in the equation below.

\[
\text{Fitness} = \frac{10 \times \frac{\text{Number of FP}}{\text{Amount of Ham Tested}}}{11} + \frac{\text{Number of FN}}{\text{Amount of Spam Tested}}
\]

Essentially, this is the average amount of errors produced by SpamAssassin when using those scores. The aim is to minimise this function, and thus correctly identify as much ham and spam as possible. The number of false positives is weighted to increase their importance to the fitness function. This is done as it is generally considered more important to avoid false positives than false negatives, as discussed earlier. The specific weighting of 10 was chosen to match the weighting used in the fitness function of the SpamAssassin GA.

This fitness function was chosen over the one used by the SpamAssassin GA as I believe it provides a better measure of the overall fitness of a solution. The goal of any spam detector is to detect as much spam as possible, whilst reducing the amount of ham that is incorrectly marked as spam. The fitness function used by my GA provides a good measure of the progress towards that goal.

However, the measure of fitness used by the SpamAssassin GA can conflict with that goal. For example, consider two hypothetical solutions: Solution A and Solution B. Solution A produces only one false positive, catching all spam, and identifying all other ham correctly. However, the score for
that one false positive is much larger than the threshold score. Solution B on the other hand, produces many false positives and false negatives. However, the average score for the false positives and false negatives is quite close to the threshold score. Clearly, Solution A is a better choice than Solution B, as Solution A detects spam almost perfectly, and using Solution B will result in a lot more spam escaping detection, and a lot more ham being incorrectly marked as spam. However, the SpamAssassin GA would select Solution B over Solution A, as the large distance from the threshold score for Solution A’s single false positive would produce a much worse fitness than Solution B, for which the average scores for false positives and false negatives are much closer to the threshold score.

3.1.3 The Multi-Objective GA

All the parameters from the single-objective GA are carried over to the multi-objective GA, with the exception of the fitness function and the method of ranking solutions.

Instead of arbitrarily combining the number of false positives and false negatives into a single measure of fitness, the percentage of false positives and the percentage of false negatives are considered separately. This gives two measures of fitness of each solution: one which measures how well it detects spam emails, and the other which measures how well it minimises the amount of ham emails that are incorrectly marked as spam. The two fitness functions are shown below.

\[
\text{Fitness for Ham} = \frac{\text{Number of FP}}{\text{Amount of Ham Tested}}
\]

\[
\text{Fitness for Spam} = \frac{\text{Number of FN}}{\text{Amount of Spam Tested}}
\]

Because the multi-objective GA does not produce a single measure of fitness, a different means of ranking solutions must be used. An implementation of the method introduced in the NSGAII algorithm (discussed previously) is used, with solutions first sorted based on their Pareto rank, and the crowding distance measure used to select between solutions of the same Pareto rank, in order to select the solutions which provide the best coverage of the solution space.

3.1.4 The Memetic Algorithm

**Hill-Climbing Local Search**

The memetic algorithm developed is simply an extension of the genetic algorithm, which performs local search of the population at a specified interval throughout the run. The local search employed is a simple hill-climber.

The local search on a single solution proceeds as follows. The local optimum for each score in the genome is determined individually. First, the score is increased, and the new fitness for the solution is evaluated. If the change in score results in an improvement in performance, then the new score is kept. This process is repeated until increasing the score no longer results in an improvement in fitness. The same process is applied in the other direction, reducing the original value of the original score while it improves performance. The score producing the best fitness is then chosen as the new score. The order that the scores in the genome are optimised in is randomised to prevent this affecting the local search.
The memetic algorithm performs the local search on all individuals that will be selected to produce the next population.

**Using a Local Search to Identify Unnecessary Tests**
The time required for SpamAssassin to process each message is not insignificant, and is directly proportionate to the number of tests that are run on the message: the fewer tests that need to be run, the faster SpamAssassin will be able to process messages. Therefore, as long as the performance of spam detection is not negatively affected, it would be useful to minimize the number of tests run, in order to reduce the time required to process each message.

To do this, another form of local search was developed. For each test, its score is set to zero, and the fitness with the zeroed score is compared to the fitness with the original score for that test. If the fitness does not get worse, then the score is kept at zero – otherwise it is reverted to the original value. Setting the score for a test to zero is equivalent to not running the test, as a test with a score of zero will not contribute anything to the overall score of a message, regardless of whether the test is hit on that message. Indeed, for this reason, SpamAssassin does not run any tests with a score of zero, so by setting the score for a test to be zero, it will not be run.

### 3.2 Testing the Algorithms

#### 3.2.1 Corpus of Emails Used To Evaluate Fitness
To provide a means of measuring the performance of spam detection for each solution, a corpus of email (with each message already identified and marked as either ham or spam) is required. Such a corpus should be *hand classified* (verified as either ham or spam by a human being rather than a spam detection filter) in order to ensure accuracy. It should provide a representative mix of the different types of ham and spam email, and should be large enough so that results on the corpus can be accurately applied to email in general (22).

The corpus chosen is the TREC 2005 Public Corpus (23). This corpus was developed specifically to address the lack of a publicly available corpus that is suitable for testing the performance of spam filters. This corpus was developed for the 2005 Text Retrieval Conference in spam detection, and has since been widely used as a means of testing spam filters. It consists of 92,189 messages; 39,399 of which are ham, and 52,790 of which are spam.

#### 3.2.2 Fitness Evaluations as a Measure of Runtime
The number of fitness evaluations made was used to compare the runtime of the algorithms. Each fitness evaluation requires the performance of SpamAssassin for the given scores to be calculated. This entails adding up the scores for all rules hit on each message in the corpus, determining if each message would have been identified as spam, and then comparing this result to the actual status of the email. Given the use of a large corpus, and the high number of rules used by SpamAssassin, this is a long process, and the overhead created by the other operations performed by each algorithm is insignificant in comparison to the cost of each fitness evaluation.

Using fitness evaluations as a measure of runtime, instead of the elapsed wall clock time also removes the effect of any differences in the implementation of each algorithm, instead comparing the performance of each algorithm in an abstract way. This is especially useful in comparing the algorithms developed with the algorithm used by SpamAssassin.
3.2.3 Separate Training and Test Corpuses
The performance of the spam filter on the set of messages it was trained on is not important – each email in the training corpus has, by definition, already been identified as either spam or ham. The true test of a spam filter is how well it is able to identify spam from a set of messages it has never seen before. It is possible for a spam filter to become over-fit to the training corpus. This occurs when it becomes overly reliant on specific characteristics of emails in the training corpus, such that it performs very well on the training corpus, but poorly on other emails.

To avoid this problem, separate training and test corpuses were used. Each corpus consists of entirely different emails. First a solution is trained using the training corpus. The performance of the solution is the assessed by running it on the test corpus. In this way, the results obtained can be safely generalised to general spam detection performance.

3.3 Implementation Details
This section explains the major details of the implementation.

3.3.1 Evaluating Fitness
The speed at which SpamAssassin is able to process messages makes it infeasible to evaluate the fitness of each solution by running SpamAssassin over the entire corpus (using the scores specified by the solution) each time a fitness evaluation is required. SpamAssassin is able to process approximately 10 messages per second, when running only local tests. The speed of processing is slowed even more when network and bayes tests are also run. Therefore, to run SpamAssassin on the entire TREC corpus of 90,000+ emails would require over 2.5 hours – not feasible for the number of fitness evaluations that need to be run.

Instead, the mass-check script provided with SpamAssassin (discussed in the background section) was used to run SpamAssassin on the TREC corpus. This produces output giving which tests were hit for each email in the corpus. As we are simply altering the scores for each test, and not the tests themselves, the tests hit for each email in the corpus will not change. This means this data can be reused for each fitness evaluation, instead of having to run SpamAssassin each time. Once the tests hit for each email is known, the fitness can be calculated by summing the score for each test hit for each email in the corpus, estimating whether the email is spam or ham, and comparing to its known type. Thus the time required to perform each fitness evaluation is no longer infeasible, although it still is the major contributor to the runtime of the GA.

3.3.2 Training and Test Corpuses
The training and test corpuses used were both drawn from the TREC corpus. Another script provided by SpamAssassin was used to separate the output of the mass-check on the TREC corpus into two sets. The first 90% of ham and spam emails in the TREC corpus were used as the training corpus. The final 10% of emails in the TREC corpus were used as the test corpus. The separation of the two corpuses is based on the time and date the message was received: all the messages in the training corpus were received before emails in the test corpus. This nicely parallels the operation of a real spam detector – which would first be trained on a set of emails previously received, and then required to detect spam in new messages received.
3.3.3 Restriction to Local Tests Only
SpamAssassin was run with only local tests enabled. Network tests were disabled because of the nature of the email being tested. Because of the gap in time between when the emails were received (pre 2005) and the present, the results of network tests will be different to what would have been seen at time of first email receipt. Tests examining the location of a message will no longer produce valid results, and data from blacklists and collaborative filters cannot be relied upon as previous users of the corpus will likely already have already reported them.

Bayes tests were disabled for a different reason. They work on the content of the emails, so would not be affected in the same way as network tests. However, the mass-check could not be successfully run without errors when bayes tests were enabled, so data for bayes tests could not be obtained.

As only local tests are being run, the overall performance of spam detection achieved will be lower than what might be possible using a combination of all tests. However, valid comparisons between the algorithms can still be made, and any improvements in detection rates obtained for each algorithm should carry over to when network and bayes tests are employed as well.

3.3.4 Modification of the SpamAssassin GA
In order to compare my algorithms to the SpamAssassin GA, its source code was modified to record the number of fitness evaluations, and output statistics on the fitness evaluations made, and false negative and false positive rate of the best solution achieved so far. No changes were made that affect the operation of the algorithm.
Experiments and Results

4.1 Selecting the Optimum Settings for the Genetic Algorithm

After constructing and the genetic algorithm, the first step was to determine which values to use for its parameters, in order to optimise its performance for the given problem. Although genetic algorithms are good general problem solvers, their performance for different types of problems can vary dramatically based on the parameters used (24). For example, some problems may require a high level of diversity to be maintained, and will therefore benefit from a larger population and selection level. Other problems may benefit from using a small population, and a more elitist method of selection.

In order to identify the best values for the genetic algorithm parameters, first a set of ‘reasonable’ parameters was chosen, so the genetic algorithm could be run. Then each parameter was tested individually.

It is important to note that this approach may not give the best results, as it works on the flawed assumption that each parameter is independent. In reality, the parameters are not independent, and will affect each other to some extent. However, the space of all possible parameter configurations is very large, so it is not possible to search the entire parameter setup space. Luckily, although a poor choice of parameters will result in poor performance, genetic algorithms are able to tolerate some variation in parameter values. For this reason, this simple approach to setting the parameters of a genetic algorithm does work quite well. Indeed, most genetic algorithms are manually tuned in this way (25). The initial value chosen for each GA Parameter are shown in Table 1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>-</td>
</tr>
<tr>
<td>Probability of Mutation</td>
<td>0.22%</td>
</tr>
<tr>
<td>Standard Deviation for Mutations</td>
<td>5.0</td>
</tr>
<tr>
<td>Selection Level</td>
<td>50%</td>
</tr>
<tr>
<td>Selection Strategy</td>
<td>Plus</td>
</tr>
<tr>
<td>Number of Parents</td>
<td>2</td>
</tr>
</tbody>
</table>

No initial value was chosen for the population size, as this was the first parameter tested, so an initial value was not required. The value for the probability of mutation was chosen with the aim that, on average, one chromosome from each genome should be mutated. As each genome consists of the scores for 464 tests, the required probability is \( \frac{1}{464} \), which is approximately equal to 0.22%. The standard deviation for mutations was chosen to match the threshold score of 5.0, that, when exceeded, causes an email to be identified as spam. The selection level was chosen to be 50%, so the top 50% of solutions will be selected to produce the next generation. As this seemed a good balance between retaining diversity, and evolving towards the best solution. Initially a plus selection
strategy was used, so the solutions selected from the previous generation will be retained in the next generation. As such the new generation will be made up of the top 50% fittest solutions from the previous generation, as well as the newly generated solutions. Finally, two parents were chosen to be used to be recombined into a new solution, as this is the solution most commonly observed in nature, so is a reasonable starting point.

4.1.1 Optimum Parameters for the GA
To determine the optimum setting for each parameter, the genetic algorithm was run a number of times, whilst adjusting only that parameter. Combining the results of each experiment, good settings for the genetic algorithm can be obtained, and are given below in Table 2. If of interest, the details of the experiments performed to obtain these values can be found in Appendix B.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Optimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>50</td>
</tr>
<tr>
<td>Probability of Mutation</td>
<td>1.0%</td>
</tr>
<tr>
<td>Standard Deviation for Mutations</td>
<td>5.0</td>
</tr>
<tr>
<td>Selection Level</td>
<td>25%</td>
</tr>
<tr>
<td>Selection Strategy</td>
<td>Plus</td>
</tr>
<tr>
<td>Number of Parents</td>
<td>5</td>
</tr>
</tbody>
</table>

4.1.2 Optimum Parameters for the MOGA
Most of the optimum parameters determined for the single-objective genetic algorithm can be safely assumed to apply equally well to the multi-objective algorithm. However, two parameters which may differ are the population size and selection level. As the multi-objective algorithm is developing a front of solutions, rather than attempting to optimise a single measure of fitness, a greater diversity of solutions must be maintained. This means a larger population is likely to be beneficial. Similarly a higher level of selection is generally also required. However, the parameters must also be balanced carefully, as large values will reduce the speed of evolution of the genetic algorithm. A large population size requires more fitness evaluations per generation, so needs to be run for longer to achieve the same result. Similarly, if the selection level is too high, the reduced selection pressure will reduce the speed of evolution of the genetic algorithm.

A larger population and selection level was required for the MOGA. A population size of 200 and a selection level of 50% provided the best trade-off between the quality of the front produced and the speed of evolution. The optimum parameters for the MOGA are shown in Table 3 below. Again, the details of the experiments performed to obtain these values can be found in Appendix B.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Optimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population Size</td>
<td>200</td>
</tr>
<tr>
<td>Probability of Mutation</td>
<td>1.0%</td>
</tr>
<tr>
<td>Standard Deviation for Mutations</td>
<td>5.0</td>
</tr>
<tr>
<td>Selection Level</td>
<td>50%</td>
</tr>
<tr>
<td>Selection Strategy</td>
<td>Plus</td>
</tr>
<tr>
<td>Number of Parents</td>
<td>5</td>
</tr>
</tbody>
</table>
4.2 Distribution of Initial Generation

The distribution of initial solutions that was observed is different to what was expected given the method of randomly generating each solution. To further investigate, a large number of solutions (4,000) were randomly generated, in order to reduce any possible random variations that may have caused this phenomenon. The resultant distribution of the initial population is shown below, in Figure 1. This shows two distinct sets of solutions, one containing those solutions with a relatively low level of false positives (Set A), and the other containing those solutions with a relatively high level of false positives (Set B).

![Figure 1: Distribution of Initial Generation of Genetic Algorithm](image)

This is unexpected, and suggests that for some reason there is no way of obtaining a solution between the two sets. This also potentially creates a problem, as all solutions in the set with higher false positives (Set B) are initially worse than solutions in the set with lower false positives (Set A), so will be lost in the selection process. However, there is a chance that these solutions may eventually evolve into better solutions than obtainable from Set A, so they may need to be preserved at the start to allow this. Therefore the difference between these two sets of solutions must be investigated, to determine whether the solutions from the right set should be preserved, or can be safely discarded.

In order to do this, the mean score for each test was computed independently for Set A and Set B. Each score is pseudo-randomly generated with normal distribution, a standard deviation of five (the threshold score), centred around zero. Therefore, measured across the entire initial population, the mean value of each score should be zero. By examining the differences in mean for each score between the two sets, this will show which scores are different, and causing the difference in false positives observed between the two sets.
The results of this experiment are shown in Figure 2, which shows the difference in the mean score between Set B and Set A. For all tests except one, the mean score for both sets was approximately zero. The only difference between the two sets was the score for the ALL_TRUSTED test. The mean score for this test was much higher for Set B – the set of solutions with a high rate of false positives.

![Figure 2: Difference in mean score for each test between Set A and Set B](image)

This suggests the ALL_TRUSTED test is the reason for the difference between the two sets. The ALL_TRUSTED test tests whether the message has only passed through trustworthy mail servers. Hitting this test is a good indicator that the message is legitimate mail: 69.8977% of legitimate emails in the corpus hit this test, and it is not hit by any spam emails. Therefore, a good score for this test would be negative, as this would reduce the total score for any email that hits this test, making it less likely to exceed the threshold and get identified as spam.

Further examining the scores for the ALL_TRUSTED test, it can be seen that the minimum score for Set A, and the maximum score for Set B, closely coincide with the threshold score of 5.0. This is shown in Table 4, below.

<table>
<thead>
<tr>
<th>Set B (high rate of False Positives)</th>
<th>Set A (low rate of false positives)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean score</td>
<td>6.974</td>
</tr>
<tr>
<td>Minimum score</td>
<td>5.002</td>
</tr>
<tr>
<td>Maximum score</td>
<td>16.308</td>
</tr>
</tbody>
</table>

As all solutions in Set B have a score for the ALL_TRUSTED test that is greater than the threshold value of 5.0, this means that most of the emails that hit the ALL_TRUSTED test will be classified as spam by solutions in this set (depending on whether the solution’s scores for other tests hit brings the total score for the message below the threshold again). As the ALL_TRUSTED test is hit by approximately 70% of legitimate emails from the corpus, but by none of the spam emails in the corpus, this results in the high rate of false positives observed for solutions from this set. Therefore, there is no need to be concerned with preserving the solutions in this set: they are inherently worse than the solutions from Set A, as the only difference is that they assign a high score to a test (ALL_TRUSTED) that is hit only by legitimate emails.
4.3 Memetic Algorithm
The following experiments investigate whether a memetic algorithm is able to improve on the results obtained by the genetic algorithm. A memetic algorithm combines a genetic algorithm with a local search, and aims to combine the advantages of each. A genetic algorithm can quickly find a good solution, but has more trouble improving this to the best solution. A local search, however, is often more effective at moving a solution to the local optimum.

4.3.1 Performance of the Memetic Algorithm
To investigate whether the memetic algorithm is able to improve on the results obtained using the genetic algorithm, the genetic algorithm (which does not use a local search) was run, and compared to the memetic algorithm run with various frequencies of local search. The results are shown in Figure 3.

![Figure 3: Comparison of Performance of Genetic and Memetic Algorithm](image)

From Figure 3, we can clearly see that the local search actually hinders, rather than improves the performance of the genetic algorithm. The best fitness is obtained by using the genetic algorithm alone, without any local search. The memetic algorithm actually performs worse than the genetic algorithm, and as the local search is performed more frequently, the performance of the memetic algorithm continues to worsen.

The reason for this poor performance is the relatively large number of fitness evaluations required to perform a local search. The genome consists of individual scores each of the 476 local tests performed by SpamAssassin. To perform a local search on one individual, the score for each test must be both increased and decreased until the fitness no longer improves, with the fitness of the individual re-evaluated after each change in score. This requires at least two fitness evaluations to be made per test, in the case where neither increasing the score, nor decreasing the score improves the fitness (i.e. the score is already at the local minimum). This results in a minimum of 952 fitness
evaluations to perform a local search on a single individual. If improvements to any of the scores can be made then the local search will require more fitness evaluations, as it will continue to change the score and evaluate the new fitness until the fitness no longer improves.

The large number of fitness evaluations required to perform each local search accounts for the poor performance of the memetic algorithm compared to the genetic algorithm, and the deterioration of performance as the frequency of local search is increased. Simply put, performing a local search is too expensive, and the time required to run a local search is better spent running the GA for more generations.

4.3.2 Using a Local Search to Identify Unnecessary Tests
To test the effectiveness of using a local search to zero the scores of unnecessary tests, first the genetic algorithm was run and then the local search was run on the solution generated by the genetic algorithm. The genomes of the solution, before and after the scores were zeroed, are shown in Figure 4.

![Scores at end of GA](image)

![Scores after Zeroing](image)

**Figure 4: Genome of solution generated by GA before and after zeroing**

Notable differences can be observed in the genome before and after zeroing from Figure 4. There appear to be far less non-zero scores after the local search has been run. Especially notable is the absence of negative scores from the second half of the genome. This suggests that the local search has been effective at reducing the number of rules that will be run. This is confirmed in Table 5, shows the details of the changes in the genome.

<table>
<thead>
<tr>
<th></th>
<th>Positive scores</th>
<th>Negative scores</th>
<th>Scores of Zero</th>
<th>Fitness of solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution generated by GA</td>
<td>362</td>
<td>114</td>
<td>0</td>
<td>0.823</td>
</tr>
<tr>
<td>Solution after zeroing</td>
<td>247</td>
<td>30</td>
<td>199</td>
<td>0.823</td>
</tr>
<tr>
<td>Difference</td>
<td>-115</td>
<td>-84</td>
<td>199</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 5: Detail of Solution generated by GA before and after zeroing**

From Table 5 we can see that, originally, none of the scores in the solution generated by the GA were equal to zero. The local search was able to zero the scores for 199 (of 476) of the tests, without worsening the fitness of the new solution.

The reason for this improvement is the way in which the GA arrives at the solution. The GA evolves the correct scores for tests through selectionary pressure – scores that result in a better fitness are selected over scores that result in a poorer fitness. However, for ineffective tests, which do not
distinguish between ham and spam very well (for example, tests that are not hit by any messages), the score chosen for that test will not affect the fitness at all, and so there is no selectionary pressure on that score to change. In this case, because of random initialisation and mutation operations performed on scores, they will simply be normally distributed with a mean of 0 and a standard deviation of 5. By running the local search, these rules can be identified, and their score set to zero. Therefore, whilst the use of a memetic algorithm employing a hill-climbing local search did not provide any benefits over using a genetic algorithm, running a simple local search at the end of the genetic algorithm to identify unnecessary tests has proven successful. The use of this type of local search has managed to improve on the solution obtained by using the GA alone: maintaining the same fitness, but significantly reducing the number of tests that need to be run. This should result in faster processing of messages, without deterioration in the accuracy of spam detection.

4.4 Performance of the Algorithms
To determine the performance of the algorithms developed, they were compared to the results obtained using SpamAssassin’s own training algorithm. Each algorithm was first used to train a solution (or set of solutions in the case of the MOGA) using the training corpus of emails. The performance of the solution(s) generated by each algorithm was then tested by running them on a test corpus.

4.4.1 Single-Objective Genetic Algorithm

Overall Performance
To compare the performance of the single objective GA developed to that of the SpamAssassin GA both algorithms were run on the test corpus, and the performance throughout the run was compared. The results are shown in Figure 5.

![Figure 5: Performance of Single-Objective Genetic Algorithm compared to SpamAssassin's Genetic Algorithm](image-url)
Figure 5 shows a substantial performance improvement is obtained by using my single-objective genetic algorithm, instead of the genetic algorithm provided with SpamAssassin. At the start of the run, the SpamAssassin genetic algorithm produces a superior fitness – this is because it uses the default scores used by SpamAssassin as a starting point, where as my algorithm starts from a randomly generated population. However, my genetic algorithm quickly overtakes the SpamAssassin algorithm, and results in a much smaller overall fitness at the end of the run.

**Comparison of Performance on Training Corpus and Test Corpus**
To determine whether the performance of the single-objective GA on the training corpus also applies to the test corpus, the GA was run on the training corpus, and the solution with the best fitness at the end of each generation was tested against the test corpus. The results of this experiment are shown below in Figure 6.

![Figure 6: Comparison of performance of Single-Objective GA on Training and Test Corpus](image)

From Figure 6 we can see that solutions produced by the GA do not perform quite as well on the test corpus. This is to be expected, as the solutions are selected based on their performance for emails in the training corpus, and the test corpus consists of a set of entirely different emails. Clearly the performance will suffer slightly when run on different emails to which it was optimised for. Also of note is the fact that the performance on the testing corpus does not always improve as the GA progresses. This is because some changes that result in improved fitness for the training corpus will result in a poorer performance on the test corpus – again because each corpus is composed of different emails.

However, there is a strong correlation between fitness for the training corpus and fitness for the test corpus. As the algorithm progresses, in general the fitness for the test corpus is improved, and the
fitness of the final solution produced is not much worse for the test corpus compared to the training corpus. This shows that the GA does not produce a solution that is over-fit to the training corpus; it is able to accurately detect spam for a set of emails that it has not been trained on.

**Comparison of Solutions Produced**

After running both my single-objective GA and the SpamAssassin GA on the training corpus, the solutions generated by each were then tested on the test corpus, to determine their performance. The performance of each solution on the test corpus is shown below in Table 6.

<table>
<thead>
<tr>
<th>Table 6: Performance of Solutions produced by my GA and the SpamAssassin GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>False Positives</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>My Single-Objective Genetic Algorithm</td>
</tr>
<tr>
<td>SpamAssassin Genetic Algorithm</td>
</tr>
</tbody>
</table>

Not only does the solution produced by my single-objective GA perform better in terms of overall fitness, it also achieves both a substantially lower rate of both false positives and false negatives. The solution generated by my GA only fails to detect 8.162% of all spam in the test corpus, in comparison to the solution produced by the SpamAssassin GA that fails to recognise 13.553% of spam. Much less ham is also incorrectly identified as spam: 0.302% compared to 1.287%. As both the rate of false positives and false negatives are reduced in comparison to the solution produced by the SpamAssassin GA, the solution produced by my GA dominates that solution. That is, my single-objective GA produces a solution that is able to detect more spam, whilst simultaneously reducing the number of ham incorrectly marked as spam, compared to the solution produced by the SpamAssassin GA. The solution produced by my algorithm is superior, regardless of the user’s preference to avoid spam or avoid losing ham.

To gain some understanding of the reasons the two solutions perform differently, the genome of each solution was compared. The genomes of the solution produced by both my GA and the SpamAssassin GA are shown below in Figure 7.

![Figure 7: Genomes for solutions produced by my Single-Objective GA and the SpamAssassin GA](image)

From Figure 7, significant differences can be seen between the two solutions. In general, the scores for the solution produced by my GA are much higher than those for the solution produced by the SpamAssassin GA. The solution produced by my GA also has many more tests with a negative score.
The reason for these differences is that the SpamAssassin GA restricts the range of the score for each test based on the total number of emails that hit that test and the hit rate on spam compared to ham emails, as previously discussed in the section detailing how SpamAssassin assigns scores.

Taken at face value, the assumptions behind these restrictions seem reasonable: it seems logical that if a test is hit by more spam than ham then it is an indicator of spam and so should be given a positive score, and vice versa. However, the performance of the solution generated by the SpamAssassin GA, which is restricted by these assumptions, is far worse than the performance of the solution generated by my GA, which does not impose such restriction on scores, instead allowing them to move unrestricted to the value that produces the best performance.

A possible reason for this result is that the assumptions that justify the restriction of scores by the SpamAssassin GA are overly simplistic, as they do not consider the interaction between rules, instead assuming each can be treated independently. For example, it is possible that it is beneficial to give a negative score to a test that is hit by more spam than ham, although this seems intuitively wrong. If the overall scores for all spam messages that hit that test are already well above the threshold, and the overall scores for ham messages hit by that test are also above the threshold, then, by giving the test a negative score, the overall scores for the ham messages that hit the test can be reduced below the threshold, so that they would be correctly identified as ham. At the same time, the overall score for spam messages hit by the rule would not be reduced to below the threshold, so they would still correctly be recognised as spam. This demonstrates the complexities in calculating the correct set of scores, and the flaws in the assumptions made by the SpamAssassin GA.

Indeed, the complexities of the problem, which make a GA so useful in this case, also make it difficult to identify the attributes of the generated solution that make it perform so well. Whilst no definitive parts of the solution can be identified that explain its performance, it does perform demonstratively better than the SpamAssassin GA, which imposes artificial restrictions on the scores for each test.

4.4.2 Multi-Objective Genetic Algorithm

**Overall Performance**

Similarly to the single-objective GA, the performance of the multi-objective GA was compared to the SpamAssassin GA by running both on the test corpus, and examining the performance of both throughout the run, as shown in Figure 8, which shows the first Pareto rank of solutions produced by the multi-objective GA at selected generations, as well as the performance of the best solution produced by the SpamAssassin GA after a equivalent number of fitness evaluations.
Like the single-objective GA, the multi-objective GA also outperforms the SpamAssassin GA. During the early stages of the run, the solution produced by the SpamAssassin GA lies in front of the equivalent Pareto front of solutions produced by the multi-objective GA. However, after 5,000 fitness evaluations, the SpamAssassin GA solution lies approximately on the Pareto front of solutions, and from then on it lies behind the relevant Pareto front – indicating that the multi-objective GA is able to produce solutions that dominate the solution produced by the SpamAssassin GA (that is it produces solutions that have both a lower rate of false positives and false negatives). The multi-objective GA also produces a broad front of solutions – allowing a tradeoff to be made between the level of false positives and false negatives. This is discussed further in the following sections.

**Comparison of Performance on Training Corpus and Test Corpus**

To determine the difference in the performance on the training corpus and test corpus for solutions generated by the multi-objective algorithm, after the multi-objective algorithm was run the first
Pareto rank of solutions from the final generation were tested on the test corpus. The results of the comparison are shown in Figure 9.

Figure 9: Comparison of performance of Multi-Objective GA on Training and Test Corpus

Figure 9 shows that, similarly to the single-objective GA, the performance of solutions generated by the multi-objective GA is slightly worse for the test corpus of emails. Also, for the test corpus, a strict Pareto front is not formed, and some solutions are of a worse Pareto rank. This is because the fitness on the training corpus does not transfer perfectly to fitness on the test corpus. Therefore, some solutions perform worse than others on the test corpus, and therefore will have a worse Pareto rank. However, again similarly to the single-objective GA, fitness is only marginally worse on the test corpus. Therefore the solutions generated by the multi-objective GA are able to accurately detect spam for a set of emails it has not been trained on.

**Comparison of Solutions Produced**

Figure 10 shows the performance on the test corpus for the best solutions produced by the multi-objective GA compared to the best solution produced by the SpamAssassin GA. Three solutions produced by the multi-objective GA are singled out for further analysis: Solution A, which produces the lowest rate of false positives, Solution C, which produces the lowest rate of false negatives, and Solution B, which provides a good trade-off between the level of false positives and false negatives.
Like the single-objective GA, the multi-objective GA is able to produce solutions that dominate the solution produced by the SpamAssassin GA (they simultaneously provide a reduced rate of false positives and false negatives). It also provides a good range of solutions, allowing a solution to be selected that meets the preference of the user for either false positives or false negatives. However, the size of the trade-off between objectives varies across the front, and solutions at the extremes of the front require a large trade-off to be made on one objective, in order to minimise the other. This can be seen in Table 7, which shows the rate of false positives and false negatives for each solution.

From Table 7, we can see that Solution B produces a good trade-off between maintaining a low rate of false positives and a low rate of false negatives. The values for both are similar to the values obtained by the best solution produced by the single-objective GA (detailed in Table 6). At this point, both the rate of false positives and rate of false negatives are superior to those produced by the SpamAssassin GA (0.377% vs. 1.287% and 8.709% vs. 13.553%). Therefore, this solution is able to detect more spam than the solution produced by the SpamAssassin GA, whilst also incorrectly
identifying much less ham as spam. However, to maximise the performance in one objective, a large reduction in performance for the other objective is incurred. For example, moving from Solution B to Solution A results in a small decrease in the rate of false positives (from 0.377% to 0.101%), but at the cost of a large increase in false negatives (from 8.709% to 17.135%). Similarly, moving from Solution B to Solution A results in a small decrease in the rate of false negatives (from 8.709% to 6.240%), but at the cost of a large increase in false positives (from 0.377% to 5.103%).

The genomes of the solutions produced by both my multi-objective GA and the SpamAssassin GA are shown below in Figure 11.

![Figure 11: Genomes for Solutions produced by the MOGA and the SpamAssassin GA](image)

Again, the solutions produced by my multi-objective GA are very different to the solution produced by the SpamAssassin GA. Whilst in general, producing larger scores than those observed in the solution produced by the SpamAssassin GA, the scores are also noticeably lower than those produced by the single-objective GA (shown in Figure 7). It is difficult to explain the differences in performance of each solution by examining the differences in the genome. The genomes for Solutions A, B and C appear to be quite similar. One difference that could explain the low rate of false positives for Solution A could be the fact it has slightly lower scores than the other two solutions. Solution A has no scores that go above approximately 12.0. Solutions B and C have a number of scores that are close to, or exceed 20.0. This could mean that Solution A produces a lower overall score, on average, for a message, partly explaining the lower rate of false positives. However, it is difficult to identify any differences between solutions B and C that could explain the lower rate of false negatives. Solution C actually appears to have more negative scores – which would seem to suggest a higher rate of false negatives, which is contrary to the actual result. This further
demonstrates the complexities of the interaction between the tests, again demonstrating that ‘intuitive’ assumptions about the behaviour of the system are not necessarily accurate in this case.
CHAPTER 5

Conclusion

This dissertation has examined the problem of assigning scores for a heuristic spam detector. Using the developed genetic algorithm, solutions are obtained that are superior to those achieved using existing methods. The implementation of a multi-objective GA allows a set of optimum solutions to be developed, allowing the user to then select one which matches their preference for ham or spam.

The use of a memetic algorithm was also investigated, however the size of the genome to be solved means that a local search is a hindrance rather than a help, although a local search is useful to identify tests that do not need to be run, thus improving the speed at which messages can be processed.

A number of possibilities exist for further research. One is to test the performance of the genetic algorithms on a live system, to verify the results obtained. The multi-objective GA could also be extended to also optimise the speed at which messages can be processed.
Appendix A: Original Research Proposal

**Title:** Email Spam Detection using a Multi-Objective Memetic Algorithm

**Author:** James Dudley

**Supervisor:** Dr. Luigi Barone

**Background**

Unsolicited bulk email, better known as spam, is an annoying reality of internet use today. Because the cost of ‘spamming’ is so low it is widely used, indeed, today spam constitutes up to 80% of overall email volume (3). This results in a waste of bandwidth and productivity for companies, and a waste of time and an annoyance for email users.

As the problem has grown, a number of methods have been developed to detect and stop spam before it reaches the inbox. The main techniques employed are:

- IP Address blacklisting,
- Phrase Matching,
- Statistics, and
- Heuristics

IP Address blacklisting (3), (26), works by storing a list of the origin of known spam and then ignoring further email sent from that IP address, under the assumption it is also spam. This approach has two problems: spammers are able to circumvent it by regularly switching IP addresses, and after spam has been sent from an IP address hijacked by a spammer, email sent by the unsuspecting computer user is now blacklisted as spam as well.

Phrase matching (3) examines incoming email, looking for phrases that have been used in mail that has already been identified as spam. This works reasonably well at identifying spam that has already been circulated, but falls down when trying to identify new spam.

Statistical approaches (3), (27), work by learning what a particular user’s personal email looks like. They extract features from the email and determine how many times each feature has been recorded in normal email and spam. The probabilities for each feature are then combined into a final probability that determines whether the email is detected as spam or not. This works particularly well for personal email, but the features of the user’s email must first be learnt before it becomes effective.

The Heuristics method (27) tests an email against a large number of rules, each designed to test whether an email is spam or not. Each rule is generally given a weighting, and then a score for the email is generated by summing the weightings for all the tests that it failed. If the email’s score is
over a certain threshold, it is then declared to be spam. This method, along with the statistical method, have proven to be the most effective in detecting spam. The table below, based on data from (27), gives a rough idea of the effectiveness of each of the methods.

<table>
<thead>
<tr>
<th></th>
<th>IP Address Blacklisting</th>
<th>Phrase Matching</th>
<th>Statistics</th>
<th>Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>0-60%</td>
<td>80%</td>
<td>80-99%</td>
<td>95-99%</td>
</tr>
<tr>
<td>False Positives</td>
<td>10%</td>
<td>2%</td>
<td>0.1%</td>
<td>0.5%</td>
</tr>
</tbody>
</table>

Whilst heuristics has proven to be the most effective method overall, the problem with this method is determining what weighting to give each rule in order to maximise the accuracy in identifying spam. Due to the large number of rules employed, mathematical techniques are too unwieldy and an exhaustive search is too time-consuming. Therefore, other optimisation techniques need to be used. One technique that could prove useful is Genetic Algorithms.

Genetic algorithms (GAs) are a computer-based method of optimisation that use a process imitating natural evolution in order to solve a problem. The basic process is as follows: First, a number of possible solutions (a ‘population’) are generated randomly. These solutions are then tested, and the best solutions are selected and the others discarded. The selected solutions are then recombined in some way to create a population of ‘child’ solutions, that combine characteristics from one or more of the ‘parent’ solutions and are also randomly altered, or ‘mutated’ slightly, and thus while similar to their parents, they are not identical. These ‘child’ solutions are then tested, and again the best solutions are selected for further ‘breeding’. This process continues until an acceptable solution to the problem has been reached.

GAs are most useful in optimisation type problems for which there is no obvious way to use more traditional problem solving techniques (14). All that is required to tackle the problem using a GA is a way of representing a solution to the problem, and a way of testing and ranking potential solutions. Given this information, a GA can be constructed to create and evolve a set of solutions until a good solution to the problem is reached.

This characteristic makes GAs useful for finding the optimum rule weightings for heuristic spam detection: a genome of a possible solution is constructed as a list of weightings for each rule, and a possible solution can be tested by running the detection engine (using the weightings specified by the solution) on a large number of ‘training’ emails for which the results (i.e. whether it is spam or not) are already known.

Indeed, a GA has been used successfully in email detection by the program SpamAssassin. SpamAssassin is an email filter that uses a heuristics based spam detection engine to identify spam. Version 2.6 uses 872 rules to test spam and the latest version 3.0 uses 628 (19). SpamAssassin is one of the leading email detection programs currently available, and has received numerous awards (27). A previous version of SpamAssassin (version 2.6) used a GA to calculate the optimum weightings. This was replaced in version 3.0 by a Perceptron Learner, which is a type of neural network. The main reason given for this change was that the GA, which was developed a number of years previously, had not been maintained and the current development team did not understand how it
worked (19). Indeed, the Perceptron Learner only performed on an equal footing to the GA, but did prove to be faster.

Although GAs do perform quite effectively in this case, as demonstrated by SpamAssassin, performance could perhaps be improved by using a Memetic Algorithm (MA). A Memetic Algorithm combines the principle of Genetic Algorithms with a local search, which is used to perform local optimisation – something that GAs struggle with. Whilst GAs can quickly find a good solution to an optimisation problem, they are not good at finding the perfect solution (15). MAs can use a local search to quickly find if there are any solutions ‘close’ to the solutions in the population that provide a better result. This can help the algorithm to move from a good solution to the optimum solution. MAs have been shown to be more effective than GAs for a number of problems (16).

A further complication of spam detection is that the problem has not one, but two objectives which must be considered in the optimisation process. Whilst a spam filter aims to reduce the amount of spam escaping detection, it must also ensure that the number of legitimate emails that are marked as spam are minimised. These two objectives conflict, as increasing the strictness of the filter to stop more spam will likely also lead to more legitimate email being marked as spam. Similarly, relaxing the filter to ensure all legitimate email gets through will also lead to more spam escaping detection. Therefore, spam detection is in fact a multi-objective problem (MOP), as it has two conflicting objectives, and improving the performance for one objective requires a trade-off in the performance of the other.

It is in fact possible to reduce any MOP to a single objective, which can then be optimised using single objective optimisation techniques (17). This can be done by combining all the objectives into a single function, weighting each based on its perceived importance. The resulting function can then be treated like a single objective, and optimised in that way. However this approach is of limited usefulness for spam detection, as the optimised solution found only applies to that particular case, and in order to find the optimum solution in another case, the evaluation function needs to be adjusted and then the optimisation process run again.

The importance of the two objectives of spam detection can vary widely between users: for many users it is far more critical to ensure that all legitimate email are received, and they are willing to accept some spam, rather than risk missing an important email. An example of this would be a business user, who would like to ensure they receive all emails sent by their customers. For other users it is important to avoid receiving any spam, even at the risk of missing some legitimate emails. An example of this case is a child’s email account – which should be protected from the offensive content found in many spam emails. In order to optimise spam detection under a variety of situations a method of finding the set of optimal solutions is needed.

In order to find the set of optimal solutions for a MOP, the notion of Pareto optimality is usually adopted (17). A Pareto optimum is a solution to a MOP for which no other feasible solution exists that improves performance for one objective without simultaneously decreasing it for another objective. The set of all such optimal solutions forms the Pareto front. The benefit of using GAs or MAs to solve MOPs is that, unlike most optimisation techniques, only a single run of the algorithm is needed to find the front of Pareto optimal solutions. Simplistically, this is achieved by selecting from the population all those solutions which are Pareto optimal with respect to the rest of the population. This gives a set of solutions that, over a number of generations, should approach the
Pareto front, and thus optimise the MOP. This approach would allow the performance of a spam filter to be optimised for all situations simultaneously, and then the optimal solution selected depending on user preference for spam prevention or reduced false positives.

**Aim**

There are two main aims of this project:

- To investigate whether the use of a Memetic Algorithm can improve on the Spam detection performance of Genetic Algorithms.
- To utilise and study the multi-objective approach in construction of the algorithm, so that spam detection is optimised over all conditions.

If time permits, an area of further interest would be to extend the algorithm to allow for online learning – that is to continue to improve the spam filter based on the user’s actual email.

**Method**

In order to achieve the aims of this project both a multi-objective GA and a multi-objective MA will need to be developed, trained and tested. The algorithms will be designed to alter the weightings of rules used in heuristic spam detection in order to optimise the performance of spam filtering.

The SpamAssassin anti-spam software will be used as the framework with which to train and test the algorithms. SpamAssassin will handle the actual spam detection process, including message parsing and testing of emails using its inbuilt message parser and heuristic spam detection engine. It will be necessary to develop an interface to SpamAssassin to allow the weightings of rules from its spam detection to be altered, and to provide a means of training and testing the algorithms. A large corpus of email, containing both known legitimate email and known spam, must also be sourced. This will be required to provide data so that the algorithms can be both trained and tested.

A basic schedule of the required activities and the expected timeframe to complete each is given in the table below.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time Frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learn about, and code a simple Genetic Algorithm</td>
<td>March</td>
</tr>
<tr>
<td>Learn about and code a Multi-objective Genetic Algorithm</td>
<td>March – April</td>
</tr>
<tr>
<td>Find a source of emails to use as training and testing data</td>
<td>April</td>
</tr>
<tr>
<td>Develop means of altering weights of SpamAssassin rules, and to run SpamAssassin on training and test emails</td>
<td>April</td>
</tr>
<tr>
<td>Develop Genetic and Memetic Algorithms to adjust weightings of rules</td>
<td>May – June</td>
</tr>
<tr>
<td>Run experiments to determine performance of algorithms (against each other and the SpamAssassin defaults)</td>
<td>July</td>
</tr>
<tr>
<td>Write draft dissertation</td>
<td>August – September</td>
</tr>
<tr>
<td>Finalise experiments</td>
<td>September – October</td>
</tr>
<tr>
<td>Complete dissertation and prepare presentation and poster</td>
<td>October</td>
</tr>
</tbody>
</table>
Appendix B: Details of Experiments Performed to Identify Optimum Parameters for the GA

**Population Size**

The first parameter examined was the population size. Population size provides a trade-off between retaining diversity, and quickly evolving to the best solution. Small populations tend to quickly converge to a single solution, which means they evolve faster, but also may become stuck in a local minimum. Large populations may converge more slowly, but the greater diversity of solutions retained reduces the chance they will become stuck in a local minimum. To determine the best population size to use for the given problem, the genetic algorithm used was run several times, with populations ranging from 4 to 1000 solutions.

A number of points can be observed from the graph. Most obvious is the fact that very large populations perform relatively poorly. Populations of 200, 500, and 1000 clearly converge much more slowly to the optimum fitness. A very small population on the other hand, suffers from the poor quality of solutions in the initial generation. A population of 4 produces a much poorer fitness in the initial generation than larger populations. This is because the initial population is randomly generated, and a smaller population means a lower chance of randomly generating a good solution in the initial population. This means with a population of 4, the genetic algorithm begins with a much poorer fitness, and it takes some time to catch up with those using a higher population.
Another point to note is that all populations seem to eventually converge to the same fitness, of approximately 0.85. This fitness is achieved after roughly 20,000 fitness evaluations by genetic algorithms with a population of 4, 10, 20, 50, and 100, and does not significantly improve, even after another 20,000 fitness evaluations. Also, although this fitness is eventually reached by the genetic algorithms with large populations, it is not surpassed by them either. This suggests that this is the minimum fitness achievable, not simply a local minimum.

From the arguments above, very large populations and very small populations can be discarded, as they reduce the performance of the genetic algorithm. However, there is little distinguishable difference in the performance of the genetic algorithm for populations of 10, 20, 50 and 100. All appear to converge at a similar rate to the optimum fitness. Therefore, any of these population sizes would be a good choice for the genetic algorithm. A population of 50 is selected, as it lies nearest the midpoint of these population sizes, however any of the others should produce similar results.
Mutation

Rate of Mutation
Next, an attempt was made to find a good rate of mutation. If the rate of mutation is too low, then the genetic algorithm may take a long time to improve. However, if the rate of mutation is too high, then once a good solution is found, it will be difficult to further improve to the best solution, as too many changes to the genome will be made at once. To determine the best rate of mutation to use for the given problem, the genetic algorithm used was run several times, and the mutation rate varied from 0.01% to 50%.

The first thing that can be seen is that both very large and very small rates of mutation produce poor results. The genetic algorithm converges very slowly when using a 0.01%, and does not level off, even after 40,000 fitness evaluations. At a 50% rate of mutation, the genetic algorithm again converges slowly, and appears to level off at a fitness of 1.1, much higher than that obtained using other rates. This is most likely because once a nearly optimal solution is reached, the high rate of mutation means that it is hard to make the few required changes, without simultaneously mutating other parts of the chromosome. A 10% rate of mutation, whilst not quite as bad, also clearly performs worse than the others.

The other rates of mutation tested produce similar results, however two rates of mutation can be further singled out. The best fitness obtained at the end of the genetic algorithm is obtained with a rate of mutation of 0.5%, although, for less than approximately 18,000 fitness evaluations, a 1% rate of mutation performs best. With little difference in the performance of these two rates of mutation, a 1% rate was chosen for its superior initial performance.
Another parameter that will affect the performance of the genetic algorithm is the standard deviation of mutation. The mutation function used mutates a score with a normal distribution, around its current value. The standard deviation will affect the outcome of this function. Small standard deviations will result in generally smaller mutations, close to the original value for the score, and reduce the chance of large changes in the score; larger standard deviations will tend to produce larger changes. In order to attempt to identify a good standard deviation to use, the genetic algorithm was run several times, with a standard deviation varying between 0.1 and 20.

![Graph showing fitness over number of fitness evaluations for different standard deviations](image)

Clearly, very low standard deviations result in poor performance of the GA. Standard deviations of 0.1 and 0.5 have particularly poor performance. Performance improves as standard deviation increases from 0.5 to 1, 1 to 2 and 2 to 5. However, for standard deviations larger than 5, performance begins to decrease again, and a value of 10 or 20 clearly performs worse than a value of 5. Therefore, the value of 5, which was chosen initially, is indeed the optimum standard deviation of mutation. This value matches the threshold score of 5, above which an email is classified as spam. It was an intuitive choice, and this case intuition is proven correct.
Selection

Selection Strategy
The selection strategy employed by a genetic algorithm will affect its performance for certain problems. An elitist strategy, such as the plus selection strategy, always retains the top solutions from the previous generation when creating a new generation. This ensures that the fitness of the best solution never worsens from generation to generation, but reduces the number of new solutions entering the population, thus reducing the chance of improvement. The comma strategy, on the other hand, does not include any solutions from the previous generation when creating a new generation. Instead, the new generation consists of only newly generated solutions and the old generation of solutions die out, their genetic information preserved only in the new solutions generated from them. This strategy can lead to fitness worsening from generation to generation, but encourages greater diversity in the population. The genetic algorithm was run with both the plus and comma strategies, to determine which would be most effective in this case.

There is not a great difference in the performance of these two strategies. However, the more elitist plus strategy does slightly outperform the comma selection strategy. This can be seen above, in which the plus selection strategy produces a slightly better fitness than the comma strategy throughout the entire run, but especially noticeable in the early stages. As there appears to be no advantage in using the comma strategy, the plus strategy will continue to be used in the genetic algorithm.

Selection Level
The selection level is another important parameter in a genetic algorithm. The selection level determines the proportion of solutions in the current generation that are chosen to be recombined.
to produce solutions for the next generation. If a plus (or elitist) selection strategy is used, then these solutions will also be directly included in the new generation themselves. The optimum selection level must find a balance between maintaining diversity and rate of convergence, for the given problem. A low selection level will result in low diversity, increasing the chance of becoming stuck in local minima, but converging quickly. A high selection level will maintain higher diversity, but may take longer to converge. The genetic algorithm was run several times, varying the selection level between 10% and 90%.

In this case, changing the selection level appears to have minimal impact on the performance of the genetic algorithm. Lowering the selection level does slightly improve performance, with a selection level of 10% achieving the best final fitness. However the difference is marginal. A selection level of 25% was chosen, as it achieves only a marginally poorer final fitness than a selection level of 10%, and performs noticeably better in the early stages. Realistically, any other selection level could be used, with little drop in performance of the genetic algorithm.
Method of Reproduction

The final parameter examined for the genetic algorithm is the method of reproduction, which determines how new solutions are generated. The simplest method is to produce new solutions by simply cloning and mutating a single parent solution—known as asexual reproduction. New solutions can also be produced by randomly combining parts of two or more parent solutions together to form the new solution. This approach no longer relies only on random mutation to produce improvements, but can also produce better solutions by combining the ‘good’ parts of the parent solutions. To determine which form of reproduction produces the best performance, the genetic algorithm was run several times, varying the number of parents recombined to form new solutions.

The number of parents combined to produce a new solution does indeed appear to make a difference to the performance of the genetic algorithm. Using asexual reproduction, which relies on mutation only, the genetic algorithm performs poorly. Moving to sexual reproduction improves performance. The number of parents used also influences performance. Using 5 parents, instead of 2, results in a noticeable improvement. However, beyond this number the improvement stops, and indeed performance appears to get slightly worse as the number of parents increases from 5 to 10, and from 10 to 12 (which is equal to the number of solutions selected from the previous generation, and so is the maximum number of parents that can be used for this population size and selection level). Therefore, sexual reproduction using 5 parents is chosen, as this appears to be the best performing alternative.
Multi-Objective Genetic Algorithm

Optimum Population Size
Most of the optimum parameters determined for the single-objective genetic algorithm can be safely assumed to apply equally well to the multi-objective algorithm. However, one parameter which may differ is the best population size to use. As the multi-objective algorithm is developing a front of solutions, rather than attempting to optimise a single measure of fitness, a greater diversity of solutions must be maintained. This means a larger population is likely to be beneficial. In order to determine a good population size to choose for the multi-objective algorithm, a number of runs were performed, with the population size varying between 50 and 500. Each time the algorithm was run for 40,000 fitness evaluations.

![Graphs showing Pareto fronts for different population sizes](image)

A noticeable difference in the Pareto front is obtained with each population size. The fronts formed with populations of 50 and 100 appear to be more restricted than the fronts produced with a population of 200 or 500. This demonstrates how a small population can restrict the size of the front, as the population is not able to support the diversity of solutions needed to form a larger front. The front formed by the population of 500, whilst showing a more complete shape, has not advanced as much as the other fronts, and is dominated by them. Because of the larger population, more fitness evaluations are required for each generation, and the lack of generations run reduces its performance over the other populations. A population of 200 seems to be optimal, as it has produced both the widest and the best performing front of solutions.
Optimum Selection Level
A multi-objective genetic algorithm is also likely to require a different selection level to a single-objective algorithm. Generally, more diversity is required by a multi-objective algorithm, as it must evolve an entire front of solutions. This means that a higher level of selection will probably be needed. However, this must be balanced with the fact that using too high a selection level may reduce the speed of evolution of the genetic algorithm. To determine which selection level should be chosen for the multi-objective algorithm, it was run a number of times, varying the selection level between 10% and 75%.

It can be seen that low selection levels, which performed best for the single-objective genetic algorithm, perform poorly for the multi-objective algorithm. Notably, a selection level of 10% results in a Pareto front that is clearly dominated by the others. There is not a great difference between the performances of the other selection levels. A selection level of 75% seems to perform best across the majority of the front, slightly exceeding the performance of a 50% selection level, which in turn slightly exceeds the performance of a 25% selection level. However, a 50% selection level seems to produce a wider front than either a 25% selection level or a 75% selection level. A 50% selection level was chosen, because it produces the widest front, and is only marginally outperformed at most points by a 75% selection level.
Bibliography


