Evolving Similarity Functions for Code Plagiarism Detection

Vic Ciesielski
School of CS & IT
RMIT University
PO Box 2476V, Melbourne
Vic 3078, Australia
vic.ciesielski@rmit.edu.au

Nelson Wu
School of CS & IT,
RMIT University
PO Box 2476V, Melbourne
Vic 3078, Australia
newu@cs.rmit.edu.au

Seyed Tahaghoghi
School of CS & IT,
RMIT University
PO Box 2476V, Melbourne
Vic 3078, Australia
seyed.tahaghoghi@rmit.edu.au

ABSTRACT

Detecting whether computer program code is a student's original work or has been copied from another student or some other source is a major problem for many universities. Detection methods based on the information retrieval concepts of indexing and similarity matching scale well to large collections of files, but require appropriate similarity functions for good performance. We have used particle swarm optimization and genetic programming to evolve similarity functions that are suited to computer program code. Using a training set of plagiarised and non-plagiarised programs we have evolved better parameter values for the previously published Okapi BM25 similarity function. We have then used genetic programming to evolve completely new similarity functions that do not conform to any predetermined structure. We found that the evolved similarity functions outperformed the human developed Okapi BM25 function. We also found that a detection system using the evolved functions was more accurate than the the best code plagiarism detection system in use today, and scales much better to large collections of files. The evolutionary computing techniques have been extremely useful in finding similarity functions that advance the state of the art in code plagiarism detection.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Retrieval models; I.2.6 [**Artificial Intelligence**]: Learning—*Parameter Learning*

General Terms

Algorithms, Experimentation

Keywords

Evolutionary Computing, Particle Swarm Optimization, Genetic Programming

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

1. INTRODUCTION

Evolutionary computing techniques have proven to be successful in a wide range of application areas and have been used to find human competitive solutions to many problems [6]. In this paper we describe the use of two evolutionary computing techniques, particle swarm optimization and genetic programming, to improve detection performance in systems designed to detect plagiarism in computer program code.

Plagiarism can be defined as the presentation of someone else's work or idea as one's own. Code plagiarism can be defined as the "unauthorized reuse of program structure and programming language syntax" [2]. There is strong evidence that code plagiarism is widespread at educational institutions [15, 18]. Plagiarism detection has wider application than just student assignments, for example in copyright and patent violation. Plagiarism detection is a difficult problem because code can be copied from other students and many web sites and then modified in various ways to disguise the fact that it is copied.

Plagiarism detection problems are of two forms: one-to-many and many-to-many. In the former case it is required to determine whether a particular file is a plagiarised copy of a file in a collection. In the latter case it is required to determine which files are plagiarised copies of each other. Methods that work well for the one-to-many case do not necessarily scale well to the many-to-many case. In this paper we are concerned with the many-to-many case. Detection methods based on information retrieval methods are well suited to many-to-many detection [2], but depend on good measures of document similarity.

The plagiarism detection problem addressed in this paper can be stated as follows: Given a collection of M program files, determine which pairs are plagiarised copies of each other. The files are student submissions that can be augmented with additional files provided by the instructor and taken from text books and web sites. The detection system is required to produce a list of program pairs, ranked by similarity. Programs which are highly likely to be copies should be at the top of the list. A human expert will then check the highly similar pairs and determine whether plagiarism has occurred. Thus it is highly desirable that plagiarised programs are at the top of the list to minimize the amount of human checking.

The key to a good plagiarism detection system is the measure of similarity between programs. There is a long history of research into document similarity measures in the field

of information retrieval. In information retrieval the task is to find all documents in a collection that are similar to some given query document. In the most common case the query document is a list of key words given by the user. The starting point for our work is a plagiarism detection system that uses a similarity function, Okapi BM25, designed for text and web documents[2]. This function has 3 parameters that can be tuned. We first look at finding optimal values for these parameters and then at finding better performing similarity functions that are structurally different.

Our overall goal is to determine whether similarity functions for code plagiarism detection can be improved with evolutionary computing. In particular, we address the following research questions:

1. Can the performance of the human derived Okapi BM25 similarity function be improved with particle swarm optimization?
2. Can genetic programming be used to evolve similarity functions of a different structure to Okapi BM25 which give better performance?

2. RELATED WORK

2.1 Plagiarism Detection

Plagiarists attempt to disguise plagiarism using a variety of techniques, for example changing of comments and formatting, changing identifier names, and reordering independent statements and procedures [8, 10, 16]. Some techniques, such as introducing unnecessary constants or macros, appear intended to deliberately confuse plagiarism detection tools.

Most current plagiarism detection methods are structure based [1, 5, 10, 12]. They seek to represent a program in an abstract way and use various algorithms to find similar structures in programs. These systems are computationally expensive and do not scale well to large collections [2].

An alternative approach [2], which does scale up to large collections, is based on the information retrieval principles of finding a document in a collection that is similar to a given query document. However, the similarity measure used in this work is based on text documents rather than computer programs.

Currently the benchmark system for code plagiarism detection is JPlag [12]. JPlag is structure based.

2.2 Finding formulas with Genetic Programming

There has been considerable success in using genetic programming to find formulas for various problems, for example the PID controller described in [6] which was subsequently patented. Genetic programming has also been used to find similarity functions for classical information retrieval [4].

2.3 The Okapi BM25 formula

Okapi BM25 [14] (Equation 1) is one of a family of state-of-the-art similarity functions for text information retrieval. It measures the similarity between a given query document[1] (Q) and a document from a collection (D). The similarity measure is basically the sum of contributions from the terms that are common to the both documents. It contains three tunable constants k_1, k_3, and b that control the within-document term frequency, within-query term frequency, and inverse document frequency respectively. A value of zero for k_1 and k_3 ignores multiple term occurrences, while a high value gives an approximately linear term frequency (i.e. twice the term frequency gives twice the score). The default setting of k_3 as 1000 is used as an approximation of infinity to give this linear relationship for the within-query term frequency [13]. Longer documents naturally receive a higher score. Normalization corrects this. A value of $b = 1$ does the full normalization, $b = 0$ removes it [7].

$$Okapi(Q,D) = \overbrace{w_t}^{IDF} \times \sum_{t \in Q \cap D} \frac{\overbrace{(k_1+1)f_{d,t}}^{TF}}{\underbrace{K}_{IDL} + f_{d,t}} \times \frac{\overbrace{(k_3+1)f_{q,t}}^{QTF}}{k_3 + f_{q,t}} \quad (1)$$

$$\text{where } w_t = log_e(\frac{N - f_t + 0.5}{f_t + 0.5})$$

$$K = k_1 \times ((1-b) + \frac{b \times D_{terms}}{avgD_{terms}})$$

$f_{d,t}$	Number of times the term occurs in the document (within-document term frequency).
$f_{q,t}$	Number of times the term occurs in the query (within-query term frequency).
D_{terms}	Number of terms in the document (length of the document).
Q_{terms}	Number of terms in the query (length of the query).
f_t	Number of documents in the collection the term occurs in (collection frequency).
N	Number of documents in the collection (collection size)
$avgD_{terms}$	Average number of terms per document in the collection (average document length).
IDF	Inverse Document Frequency.
IDL	Inverse Document Length.
TF	Term Frequency
QTF	Query term frequency

In [19] there is a thorough investigation of a variety of components found in similarity functions. The paper concludes that different similarity functions are required for different applications and there is no consensus on which functions should be used for particular contexts.

3. FITNESS EVALUATION

Fitness evaluation is identical for both the particle swarm and genetic programming approaches and is described in this section.

3.1 Fitness Measure

For good performance in many-to-many plagiarism detection, a plagiarism detection system should have a low rate of false positives, and catch most of the instances of plagiarism. False positives are costly to the user, who must take the time to manually check the two programs for no gain. While the user would like to find all cases of plagiarism, the user is likely to be satisfied with lower recall (Eqn 2) if it results in fewer false positives. Our preference is high precision, especially at lower recall (Eqn 3) levels.

[1]In our application a *document* is the same as a *program*.

Rank	Similarity	Prog 1	Prog 2	Plagiarised?
01	96.55%	020	103	Y
02	93.83%	247	243	Y
03	93.23%	103	020	Y
04	93.06%	086	149	Y
05	75.91%	165	210	N
06	74.66%	165	107	N
07	68.12%	055	205	N
08	65.96%	010	100	N
09	64.38%	276	196	N
10	52.06%	157	163	N

Figure 1: Ideal output from a plagiarism detection system.

$$Recall(R) = \frac{Number\ of\ correctly\ classified\ positives}{Number\ of\ actual\ positives} \tag{2}$$

$$Precision(P) = \frac{Number\ of\ correctly\ classified\ positives}{Number\ classified\ positive} \tag{3}$$

In a typical situation an instructor will have a collection of several hundred submissions (documents) and wants to know which ones are copies of each other. The instructor submits the collection to the plagiarism detection system. The ideal output would be a ranked list of pairs of documents with an associated similarity score. The plagiarised documents would be at the top of the list and the have high similarity scores. The non plagiarised documents would be at the bottom of the list and have low scores. There would a big gap in similarity scores between the plagiarised and non plagiarised groups. The instructor wound inspect the high scores and verify that plagiarism had in fact occurred. In figure 1, for example, if the first four pairs and no others are plagiarised, the output is ideal. The plagiarised pairs have a high similarity measure and there is a large gap to the non plagiarised pairs.

In our experiments we have used the following fitness function which captures some of the above properties. The fitness score is calculated by summing the reciprocal ranks of the known plagiarised pairs. This is then divided by the maximum score from such a summation so the fitness ranges from 0.0 to 1.0 for ease of interpretability. The resulting fitness function heavily favours plagiarism detections at the top of the list. We call this fitness measure the *normalized cumulative reciprocal rank* (NCRR). The NCRR is computed as follows:

$$NCRR = \sum_{i=1}^{|D|}(plag(D_i) \times \frac{1}{i}) \div \sum_{i=1}^{|R|}\frac{1}{i} \tag{4}$$

where D is the set of retrieved document pairs, R is the set of known plagiarised document pairs and $plag(d)$ returns 1 for a plagiarised document pair and 0 for a non plagiarised pair.

In a situation where there are 4 plagiarised documents, as in figure 1, the best possible score would be $1/1 + 1/2 + 1/3 + 1/4 = 2.0833$ and this would be the normalization factor. The similarity function which returned the list shown in figure 1 would then receive the maximum fitness score of $2.0833/2.0833 = 1$. Now consider the case of a similarity function that returns the same ranks as figure 1 but with the programs at rank 2 (247, 243) and rank 6 (165, 107) swapped. This ranking is no longer ideal and the corresponding NCRR would be $(1/1 + 0/2 + 1/3 + 1/4 + 0/5 + 1/6)/2.0833 = 0.8400$.

At this stage we have not attempted to formulate the desired gap in similarity measure between the plagiarised and non plagiarised documents into the fitness measure.

In deciding on this fitness function, we explored a wide range of alternatives. These included functions based on 11-point average precision, mean average precision, R-precision and classification accuracy. Some of these measures involve querying only the plagiarised documents which gives much faster computation times, but the results were not as useful from a practical standpoint. More details can be found in [17].

Table 1: The Source Code Collections

		Program Length		Plagiarised
Collection	Programs	Mean	Std. Dev	Pairs
A	256	445	178.7	25
B1	191	1148	478.1	25
B2	171	874	339.8	45
C1	158	950	167.6	35
C2	158	1353	520.7	63

3.2 Source Code Collections

For training and testing, we use five code collections from three different university subjects (See Table 1). These collections consist of actual student submissions to assignments written in the C programming language. At the beginning of this work only one of these, collection A, had carefully checked ground truth. The ground truth for the other collections was determined iteratively with existing plagiarism detection tools. Plagiarised documents mostly appear in pairs with a few groups of three or more. If three programs are copies of each other, this would be accounted for as three plagiarised pairs in Table 1. The ground truth does not differentiate between the original and any copies, but records only that members of the set are co-derived.

3.3 Train and Test Methodology

The default Okapi BM25 and JPlag methods have fixed parameters and do not require training data. The PSO and GP approaches both need training data sets. Since collection A is the most carefully checked for ground truth we use this as the training data in the first instance. The other collections will be used for testing.

We perform 10 training runs, both with PSO and GP and then use the best individuals for testing. For evaluation, the best solutions are tried against the four test collections using the NCRR measure. This measure will give a single value per system per collection, thus facilitating comparison. We also calculate a combined NCRR by averaging these scores.

In addition, we compare the evolved similarity functions against the default Okapi BM25 function, and against the popular JPlag plagiarism detection system. JPlag has the option to allow specified basecode, that is code provided by the instructor that will be common to all submissions, to be ignored when comparing similarity between documents. We use JPlag both with and without this option.

Our strategy of using just collection A as training data

carries with it two potential problems: (1) Overfitting and (2) Not enough training data for good generalization to other collections. We investigate these issues by incorporating additional collections into the training data.

3.4 Implementation

In a preprocessing step the programs are tokenized, converted to n-grams of size 4 and an index created as in [2]. Evaluation of an individual proceeds as follows: The individual is translated into a similarity function suitable for a search engine. Each program in the collection is used as a query document and the similarity measure with respect to every other document is computed. The pairs are sorted by the measure described above. Fitness evaluation takes about 10 seconds on a 2.8GHz Pentium 4 processor with 2GB of RAM. While the expensive fitness evaluation is a concern during evolution, it is not a concern in a deployed system. The evolved similarity functions are comparable in computation time to the hand crafted ones.

4. PARTICLE SWARM APPROACH

Particle swarm optimization is a versatile and robust is a technique for numerical parameter optimization based on a population of particles moving in n-dimensional space and converging on the optimal value [3, 9]. There has been considerable research into update equations for particle velocity and position. Since we have a relatively straight forward 3 dimensional problem with the Okapi BM25 formula, we use the one of the simplest variations as shown in equation 5.

$$v_{id} = v_{id} + 2 * rand() * (p_{id} - x_{id}) + 2 * rand() * (p_{gd} - x_{id}) \quad (5)$$

$$x_{id} = x_{id} + v_{id}$$

v_{id}: particle velocity
x_{id}: particle location
p_{id}: personal best location
p_{gd}: global best location
$rand()$: random number$\epsilon[0, 1)$

We give k_1 and k_3 a range of 0.0 to 1000.0 and b the full range of 0.0 to 1.0. As explained in Section 2.3, with these ranges, the term frequencies and inverse document length can be assigned a wide impact range, from maximum effect to being completely ignored.

4.1 Results

Figure 2 shows the fitness graph for 10 runs for 50 generations. It is clear that all runs have converged by the 50th generation. None of the runs have delivered an individual that gives perfect fitness.

Table 2 shows the best evolved parameter values. The first two rows of Table 4 show the performance of the default Okapi BM25 and the evolved Okapi on the four test collections. It is clear that the performance of the evolved function is superior to the default.

Figure 3 shows the comparison as a standard 11-point average precision graph. The x-axis shows the recall. The y-axis shows the corresponding precision, calculated for the 11 standard recall levels: 0%, 10%, .. 90%, 100%. At every recall level the precision of the evolved Okapi is higher than that for the default Okapi BM25.

Table 2: Default and Evolved Parameter Values for Okapi BM25

	k_1	k_3	b
Default	1.2	1000	0.75
Evolved	1.62	0.302	1.00

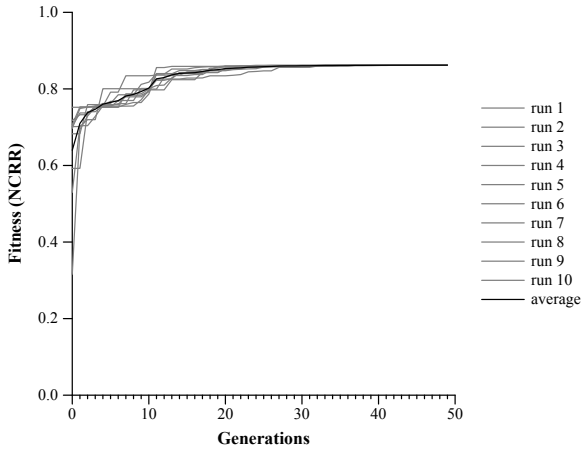

Figure 2: Best fitness during PSO training showing individual runs and average.

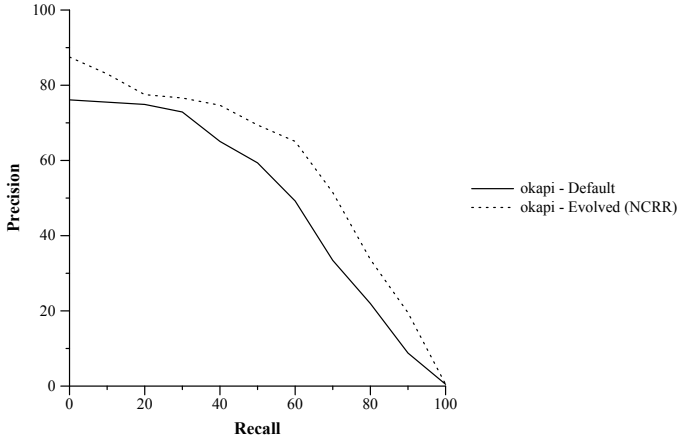

Figure 3: Precision/Recall for PSO-evolved Okapi BM25 parameters on test collections

4.2 Analysis of Evolved Parameters

The optimal value of b was 1.0 in all runs. As noted in Section 2.3, this constant controls the IDL component with a value of 1.0 giving long documents the harshest penalty. This shows that length normalization is especially important when searching for evidence of code plagiarism. This is all the more interesting as the program lengths in Collection A do not vary a great deal.

In all runs the evolved values for k_1 were between 1.59 and 1.65 (mostly 1.61-1.63) and the evolved values for k_3 were between 0.24 and 0.31 (mostly 0.30-0.31). The similarity in performance of the evolved parameters suggests that any values between these ranges are appropriate. Low values of k_1 and k_3 indicate that linear versions of within-document term frequency and within-query term frequency should not be used for code plagiarism detection. While within-query term frequency is often ignored in text information retrieval,

where queries usually do not contain duplicate terms, its importance in code plagiarism detection is underlined by evolved solutions favouring a narrow range of values for k_3, very far from the default value of 1000.

Intuitively one would expect that k_1 would be equal k_3 since query and document are interchangeable in the plagiarism situation. However, the best results are observed when k_1 is increased, and k_3 is decreased. This indicates that terms in the query should be assigned lower weight than terms in the document. For our particular application this seems counter-intuitive and merits further investigation.

5. GENETIC PROGRAMMING APPROACH

In the previous section we established that PSO could be used to find parameters for the Okapi BM25 formula that gave better detection performance than the defaults. In this section we explore whether detection performance can be further improved by using genetic programming to find novel similarity formulas. We carry out this part of the investigation in two phases - standard and specialized. In the first phase we use standard tree based genetic programming. In the second phase we use restrictions on the structure of the formulas and parsimony pressure to favour small solutions. Our expectation is that standard GP will give more accurate formulas and that specialized GP will give more understandable formulas.

Note that the the summation $\sum_{t \epsilon Q \cap D}$ is not part of the evolved individual. This summation is automatically applied to all individual during fitness evaluation.

5.1 Standard Genetic Programming

The configuration for standard genetic programming is shown in table 3.

5.1.1 Functions and Terminals

As in most applications of genetic programming considerable thought needs to be given to the components that make up the solutions. If there are too few components, or they are poorly selected, there may not be enough expressiveness to obtain a satisfactory solution. Allowing everything would make the search space impractically large.

We chose the set of terminals and functions to use after examining various similarity functions used in information retrieval, studying prior analysis of similarity functions [19], and personal testing. As noted earlier, determining the fitness measure required considerable experimentation and much of this experimentation was with different functions and terminals. Our final function set is interesting in that it contains no *minus* operator. Early in the development we had minus in the function set. This resulted in negative fitness values which caused problems in the fitness evaluation. Leaving out *minus* solved these problems without loss of accuracy. The logarithm function was omitted for the same reasons. However, since logarithms are known to be useful in similarity functions we have added terminals that incorporate logarithms.

We did not include any terminals that remain constant for a given collection. While many similarity functions contain terms such as N and $avgD_{terms}$, the evolutionary process occurs on a single collection resulting in these terms remaining constant during an evolution run. For example, when using Collection A, N and 296 would be interchangeable as N is always equal to 296 throughout the run. Such terminals, however, are still useful in certain circumstances so we combine them with other related terminals to create combined terminals. These include $\frac{f_t}{N}$, $\frac{D_{terms}}{avgD_{terms}}$ and $\frac{Q_{terms}}{avgD_{terms}}$.

We do not allow f_t to be used independently, it signifies the number of documents containing a particular term, and its value is influenced by the collection size. By using $\frac{f_t}{N}$ instead, the value is normalized between 0.0 and 1.0 and represents the proportion of the collection containing the term. This terminal is more stable across different collections. The inverse of this function is also added as it is used within various similarity functions for IDF. While it is possible for GP to invert the terminal on its own, we do not want to make it hard for the GP to use what is a fundamental component. For similar reasons we added relative versions of D_{terms} and Q_{terms}, comparing them to the average document size in the collection. While $\frac{Q_{terms}}{avgD_{terms}}$ makes little sense for text information retrieval, for our application, however, queries exist as documents in the collection.

We allow a random floating point number between 0.0 and 100.0 to be used as a terminal. Such a floating point number could help in fine-tuning the similarity functions.

5.1.2 Other Considerations

Due to a number of implementation restrictions of the search engine some of the evolved individuals are not valid and cause the search engine to crash. An example is a divide by zero error. A common solution to the divide by zero problem, protected division, is not possible here since an individual is converted into a C program and compiled and

Table 3: Genetic Programming Configuration .

Parameter	Value
Population Size	200
Crossover Rate	0.68
Mutation Rate	0.30
Elitism Rate	0.02
Max Generations	50
Selection	Tournament, size 5
Termination	Perfect fitness or 50 generations
Replacement	Generational replacement
Fitness	NCRR on training data
Functions	$+, \times, /$
Terminals	
$f_{d,t}$	within-document term frequency
$f_{q,t}$	within-query term frequency
D_{terms}	document length
Q_{terms}	query length
$\frac{f_t}{N}$	collection frequency (normalized)
$\frac{N}{f_t}$	inverse collection frequency (norm))
$\frac{D_{terms}}{avgD_{terms}}$	relative document length
$\frac{Q_{terms}}{avgD_{terms}}$	relative query length
$1 + \log_e(f_{d,t})$	within-document term frequency (logarithmic formulation)
$1 + \log_e(f_{q,t})$	within-query term frequency (logarithmic formulation)
$\log_e(1 + \frac{N}{f_t})$	inverse collection frequency (logarithmic formulation)
drand	random floating point number between 0.0 and 100.0

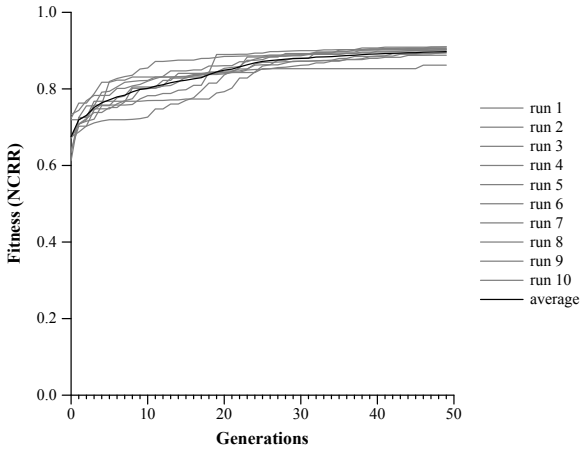

Figure 4: Best fitness during GP (Standard) training.

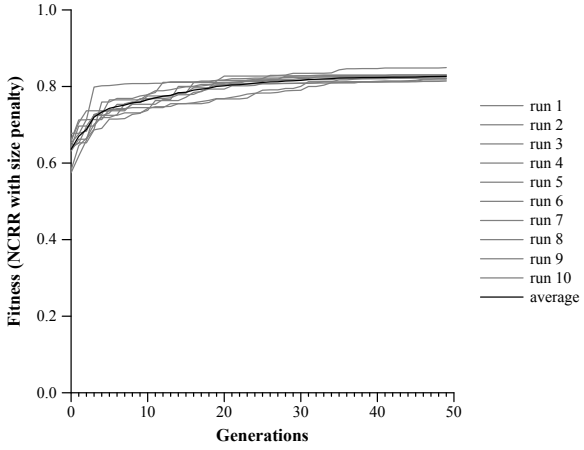

Figure 5: Best fitness during GP (specialized) training.

executed in the search engine environment. Any similarity functions that cause errors receive a fitness value of zero, and are unlikely to survive in the next generation.

The search engine cannot operate when the similarity function does not include $f_{d,t}$. This makes sense since at the very least, a similarity measure would need to know the existence of terms within a document to calculate the similarity with another document. We enforce the inclusion of this term, and do not let any programs without $f_{d,t}$ enter the population at any point. Initialization, crossover and mutation routines are modified to impose this restriction.

5.2 Specialized Genetic Programming

The goal of this part of the work was to evolve programs that could be understood. We used parsimony pressure to favour small programs and experimented with various restrictions on function set, terminal set and allowable program structure, for example, sum of products form.

We use a logarithmic size penalty:

$$fitness_i = performance_i / \log_{10^{12}}(size_i) \qquad (6)$$

We have a preference to use this instead of a linear size penalty because the penalty is consistent in proportion to the performance. Linear variations penalize low performance solutions comparatively severely. We chose the value of 10^{12} through empirical evaluation.

Some combinations of expressions that are hard to interpret involve compounding of non-linear operators [11]. For example, $log(|log(x)|)$ is valid but very difficult to interpret for humans. We address this issue by restricting the functions such as *log* to certain situations only. Rather than including *log* in the function set we provide new combined terminals that represent popular uses of logarithms in similarity functions.

Random numbers have been removed from the terminal set. This is likely to improve generality, and preliminary tests showed that removing random numbers has little effect on the quality of solutions.

We experimented with a number of structural limitations in the expectation of improved understandability. These included a sum of products form similar to that suggested in [11] and a function set consisting only of $\{+\}$ with the hope that a kind of accumulation-of-evidence-for-plagiarism strategy might evolve and we would be able to interpret the different components. However, the domain experts did not find these functions any more understandable than the ones evolved with parsimony pressure. The experts basically got sick of us asking them to interpret our formulas. This involves considerable mental effort and in general each evolved formula had some components that "made sense" and the experts could see why they were there, but also contained other components that could not be explained.

The final configuration for the specialized GP was the same as in table 3 but without the random constants and with the parsimony pressure described above. The major factor in improving performance in the specialized GP approach was the inclusion of complex terminals based on domain knowledge.

5.3 Results

The results for the 10 training runs on collection A are shown in figure 4 for standard genetic programming. Those for specialized genetic programming are shown in figure 5. They are quite consistent and there is not much variability. The training fitness for standard GP at 50 generations is slightly higher than the for the specialized case due to the size penalty. It is possible that the solutions could improve if the evolutionary process were allowed to continue for more generations, but we think that such improvements would be minor.

The performance on the test collections is given in the lines 'GP Std' and 'GP Spec' in table 4. The last column in this table is the average fitness over all four collections. Both GP approaches outperform the evolved Okapi. Contrary to our expectations the average NCRR for specialized GP is higher than that for standard GP. However, closer inspection reveals that this is mostly due to vastly improved performance on B1. Performance of the two GP methods on the other collections is very similar.

While table 4 provides a comparison of the NCRR scores for all of the methods it does not give very much intuition into what the user would see and be required to deal with. Some of this is presented in a limited, but normalized, fashion in table 5. In this table Maxscore is the number of

Table 4: Comparison of Approaches (NCRR).

	Collections				Average
	B1	B2	C1	C2	NCRR
Okapi Def	0.1628	0.6693	0.8913	0.9499	0.6683
Okapi Evo	0.3705	0.8780	0.9031	0.9738	0.7813
GP Std	0.4763	0.9416	0.9064	0.9684	0.8232
GP Spec	0.8128	0.9414	0.8993	0.9689	0.9056
GP Std 2		0.9410	0.9209	0.9486	
GP Std 3			0.9050	0.9508	
GP Std 4				0.9469	

Table 5: Comparison of Approaches (Detections).

	B1	B2	C1	C2
Okapi - Default	2	48	42	93
Okapi - Evolved	6	56	45	102
GP Std	30	64	46	99
GP - Spec	30	63	46	102
JPlag - without basecode	22	48	39	84
JPlag - with basecode	22	66	46	88
Maxscore	50	90	70	126

plagiarised pairs in a collection. A perfect result for a collection would have the have the all of the plagiarised pairs as the first Maxscore outputs. The table shows the actual number of plagiarised pairs in the first Maxscore outputs, for example, for collection B1 for Okapi Default, 2 of the top 50 outputs are plagiarised pairs. Note that while 'GP Std' and 'GP Spec' both have 30 plagiarised pairs in the top 50 for collection B1, the NCRR score for 'GP Std' is much lower. This is because there are two false positives near the top of the 'GP Std' list. The table includes the performance of the JPlag system on the same data.

A full comparison with the JPlag system is not possible due to operational restrictions on the JPlag system. It was not possible to get comparable JPlag data for table 4 and the JPlag scores for collection C2 may be underestimates. Also, since the evolved systems do not use basecode it is fair to compare them to JPlag without basecode on collections B1, B2 and C1. On these collections the GP systems are superior. The evolved Okapi is very poor on collection B1, but otherwise competitive. It is interesting to note that the evolved GP functions are competitive with JPlag with basecode. Further work is needed to determine whether the evolved systems could be improved with the basecode feature.

5.4 Analysis of Evolved Functions

Some of the best evolved functions using specialized GP are shown in figure 6. Some manual simplification has been done.

The prevalence of components related to inverse document frequency in all three functions shows that term rarity is very important. Interestingly, document length does not appear in the evolved functions. This could be because it is not important, or because we did not include suitable terminals for the lengths to be used in the restricted format. We suspect that the latter case is true, since our earlier PSO experiments suggest that document length is important.

We analyze the first function in figure 6. This function rates a query and document as similar when the terms common to both are rare within the collection but frequent within the query and document. This is reinforced when there are multiple occurrences of these terms. The lowest scores are given when the terms common to both occur frequently within the collection. The length of the document (D_{terms}) works to negate the advantage aggregation gives to longer documents since they naturally have more terms (Two long documents will generally have more shared terms than two short documents).

The fact that terms related to inverse document frequency are so common in the evolved formulas suggests there are better ways to favour term rarity than those currently in use in human devised similarity functions.

5.5 Generalization

In section 3.3 we noted that using just collection A as the training data might not lead to good generalization to other collections. The results given in the first four lines of table 4 suggest that there is good generalization to other collections. An open question at this point is whether the performance can be improved by using more training data. We have investigated this question by performing some additional runs with more collections in the training data. Row 'GP Std 2' shows the results for using two collections, A and B1, as the training data. Note that the B1 column entry is now blank. This is because collection B1 can no longer be used as an independent test set. The Average NCRR entry is also blank because an average for this row would no longer be comparable to the values above. Row 'GP Std 3' shows the results for using three collections, A, B1 and B2 as the training data. Row 'GP Std 4' shows the results A, B1, B2 and C1.

In comparing 'GP Std' with 'GP Std 2-4' it can be seen that the additional training data has helped in some cases and not in others. While we have not carried out an exhaustive set of experiments with all possible combinations of collections as the training data, it seems that the examples in collection A are adequate for evolving detectors with good generalization.

6. CONCLUSIONS

Our goal in this work was to determine whether evolutionary computing could improve the similarity functions used in code plagiarism detection. We found that particle swarm optimization could be used to give better tunable parameters for the human devised Okapi BM25 similarity function currently in use. We also found that genetic programming could be used to give better performing similarity functions with novel structure. On test collections where a fair comparison was possible, the plagiarism detection systems using the evolved GP functions outperformed JPlag, the most widely used state-of-the art plagiarism detection system. Analysis of the evolved formulas suggested that human derived similarity formulas could be improved by more consideration of the effect of terms that are common in the collection, but rare in a particular query and a similar document.

While we expended considerable effort in attempting to evolve understandable functions, we were not totally successful. Further work on alternative terminals, functions and formula structures based on domain knowledge, and perhaps multi-objective algorithms, is needed.

The focus of our work was on improved similarity functions rather than developing a working plagiarism detection

$$\sum_{t \epsilon Q \cap D} \frac{1}{D_{terms} \frac{(\frac{f_t}{N})^2}{log(1+\frac{N}{f_t})(f_{d,t}+log(1+\frac{N}{f_t}))} + log(1+\frac{N}{f_t})^2} \tag{7}$$

$$\sum_{t \epsilon Q \cap D} \frac{\frac{D_{terms}}{avgD_{terms}}}{\frac{Q_{terms}}{avgD_{terms}} + \frac{f_{q,t}}{\frac{f_{d,t}}{1+log(f_{d,t})}} + f_{d,t} + \frac{2D_{terms}}{avgD_{terms}}} \times \frac{f_{d,t}}{1+log(f_{d,t})} \times \frac{1}{D_{terms}} \times log(1+\frac{N}{f_t})^9 \times (\frac{f_t}{N})^3 \tag{8}$$

$$\sum_{t \epsilon Q \cap D} \frac{f_{q,t} \times f_{d,t} + \frac{f_t}{N} + (1+log(f_{d,t}))}{2(1+log(f_{q,t})) \times 2^{\frac{(1+log(f_{q,t})) \times \frac{f_t}{N}}{\frac{Q_{terms}}{avgD_{terms}}}} \times \frac{D_{terms}}{avgD_{terms}} + f_{d,t}} \tag{9}$$

Figure 6: Best similarity functions evolved with specialized GP.

system. However, some additional work on user interface development and highlighting of similar code blocks between files would give a practical plagiarism detection system.

Acknowledgments

We thank Andrew Turpin and Falk Scholer for their assistance with the interpretation of the evolved formualas.

7. REFERENCES

[1] K. Bowyer and L. Hall. Experience Using 'MOSS' to Detect Cheating On Programming Assignments. In *Proceedings of the Frontiers in Education Conference*, volume 3, pages 18–22, 1999.

[2] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel. Efficient Plagiarism Detection for Large Code Repositories. *Software - Practice & Experience*, 37(2):151–175, 2007.

[3] M. Clerc and J. Kennedy. The Particle Swarm: Explosion, Stability, and Convergence in a Multi-Dimensional Complex Space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002.

[4] W. Fan, M. Gordon, and P. Pathak. A Generic Ranking Function Discovery Framework by Genetic Programming for Information Retrieval. *Information Processing and Management*, 40(4):587–602, 2004.

[5] D. Gitchell and N. Tran. SIM: A Utility For Detecting Similarity in Computer Programs. In *Proceedings of the 30th Technical Symposium on Computer Science Education*, pages 266–270, New York, NY, 1999. ACM Press.

[6] John R. Koza et al. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[7] K. S. Jones, S. Walker, and S. E. Robertson. A probabilistic model of information retrieval: Development and comparative experiments - part 2. *Information Processing and Management*, 36(6):809–840, 2000.

[8] M. Joy and M. Luck. Plagiarism in Programming Assignments. *IEEE Transactions on Education*, 42(2):129–133, 1999.

[9] J. Kennedy and R. Eberhart. Particle Swarm Optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

[10] C. Liu, C. Chen, J. Han, and P. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 872–881, New York, NY, 2006.

[11] T. McConaghy and G. Gielen. Canonical Form Functions as a Simple Means for Genetic Programming to Evolve Human-Interpretable Functions. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 855–862, New York, NY, 2006. ACM Press.

[12] L. Prechelt, G. Malpohl, and M. Philippsen. Finding Plagiarisms among a Set of Programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002.

[13] S. Robertson and S. Walker. Okapi/Keenbow at TREC-8. *Overview of the Eighth Text REtrieval Conference (TREC-8)*, pages 151–162, 1999.

[14] S. Robertson, S. Walker, and M. Hancock-Beaulieu. Experimentation as a way of life: Okapi at TREC. *Information Processing and Management*, 36(1):95–108, 2000.

[15] J. Sheard, M. Dick, S. Markham, I. Macdonald, and M. Walsh. Cheating and Plagiarism: Perceptions and Practices of First Year IT Students. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education*, pages 183–187, New York, NY, 2002. ACM Press.

[16] G. Whale. Identification of Program Similarity in Large Populations. *The Computer Journal*, 33(2):140–146, 1990.

[17] N. Wu. *Evolving Similarity Functions for Code Plagiarism Detection*. Honours Thesis, RMIT, School of Computer Science and Information Technology, 2007.

[18] J. Zobel. "Uni Cheats Racket": A Case Study in Plagiarism Investigation. In *Proceedings of the Sixth Australasian Computing Education Conference*, volume 30, pages 357–365, Darlinghurst, Australia, 2004. Australian Computer Society.

[19] J. Zobel and A. Moffat. Exploring the similarity space. *ACM SIGIR Forum*, 32(1):18–34, Spring 1998.