Chapter 5

Validating Our GP Learning System

"Name the greatest of all inventors. Accidents"
Mark Twain

Rubber in its natural form is not particularly useful. In cold conditions it is brittle and cracks, yet in hotter climates it readily melts. It was the accidental spillage of sulphur onto raw rubber by Charles Goodyear in 1839 that led to the discovery of vulcanisation, the process by which rubber can be given a stable consistency[1]. Similarly, the recipe for Coca Cola was formulated inadvertantly by John Pemberton while he tried to concoct a pain relief remedy[2]. In 1897, Ernest Duchesne submitted his thesis describing his discovery, again by accident, that certain moulds would kill bacteria. Although his research went largely unnoticed it was the first documented discovery of antibiotics[3].

While science certainly does not progress through accidental discovery alone, chance happenings like these can suggest avenues of exploration that had not previously been considered. The inherent randomness and abstract nature of Genetic Programming would appear to make it quite good at simulating creativity, but is this enough for it to compete with the best of human-written pattern recognition algorithms? The goal of this Chapter is to make several empirical comparisons in order to answer this question.

GP has already been recognised as an automatic innovator in fields outside of pattern recognition. According to the company of John Koza[62], whose seminal books on GP continue to dominate the literature, there are at least 36 recorded instances where GP has produced human competitive solutions, mainly within the domain of evolved electronics. Twenty one

[1]Goodyear was unable to patent his invention and died penniless.
[2]Pemberton later sold the recipe for $900 and died penniless.
[3]Duchesne later died penniless, ironically from Tuberculosis, which his own discovery could have treated.

of these instances either infringe or duplicate functionality of previously patented 20th/21st Century inventions, and two are sufficiently novel as to be patentable in their own right.

In the latter part of this Chapter, the author's GP toolkit will be compared to a standard GP implementation, then to other results attained using GP researchers in general, and finally to a range of other popular classification techniques. This Chapter begins, however, with a more thorough description of the author's GP system, an enumeration of its novel features, and a discussion of how a Genetic learning system, usually plagued by problem-specific paramters can be adapted into a "black-box" generic problem solver.

5.1 Building a GP System

Although the author's initial experiments made use of ECJ[6], a popular but cumbersome open-source GP toolkit, it became apparent that in order to experiment with changes to the GP system itself, it would be worthwhile to develop the GP system from scratch[4]. This toolkit (named SXGP, after its *alma mater*) is specifically designed for evolving solutions to problems involving vision.

Being practically minded, the author also sought to develop a toolkit whose results could easily be put to work on different problems. ECJ is rather difficult to integrate into other software since it relies on configuration files and does not readily output individuals as java objects[5]. The author's toolkit, SXGP, can be instantiated though a straightforward programming interface, and can deploy programs in various different ways so that they may be put to work on real problems.

The various components that make up a Genetic Programming toolkit were described in detail in Chapter 3. There are numerous choices that one can make for each component, so two Genetic Programming implementations can actually differ quite significantly. Thus, before the comparisons commence, it is necessary to describe SXGP in more detail.

5.1.1 SXGP's Specification

The main components of the author's system are specified in Table 5.1.

[4]Of course, it is also more satisfying to re-invent the wheel than to use other peoples' code!

[5]The author has written a library to do this for ECJ, http://vase.essex.ac.uk/ecj/ecj2java

Component	Method	Where Described
Representation	Tree	Page 34
Tree Builder	Ramped Half and Half	Page 35
Selection Method	Tournament Selection	Page 40
Generation Gap	Generational	Page 45
Fitness Function	Variable	Page 88
Elitism	Enabled	Page 37
Genetic Operators	Crossover and Mutation	Pages 42, 44
Population Size	Variable	Page 84
Generations	Variable	Page 84
Function Nodes	Problem Dependent	
Terminal Nodes	Problem Dependent	
Learning Paradigm	Supervised	
Island Selection	Available	Page 47

Table 5.1: Specification of SXGP, the author's Genetic Programming Toolkit

5.1.2 Implementation Notes

SXGP is implemented in Java, a language well suited to the development of abstract tools. Java also offers excellent libraries for image processing, networking and multi-threading. The cross platform nature of Java makes it particularly suitable for distributing the Genetic Programming process over a number of CPUs, a feature also implemented in SXGP.

Results by a performance profiler reveals a significant amount of computational time expended by the genetic programming process is actually devoted to the copying of individuals, which is necessary to ensure that offspring are separate instances from their parents[6]. Although the author originally used a flexible copying operator which serialises an object into a stream and then re-instantiates it as a new instance, this was found to be a significant bottleneck. Instead each object implements its own cloning method which permits the process to

[6]Otherwise the later actions of genetic operators on parents would also affect the offspring

proceed much more quickly, while maintaining the requirement to perform “deep cloning”[7].

5.1.3 Novel Features

SXGP comprises a number of features that are novel. These include:

Strong-er Typing Depending on the function set used and the maximum tree size permitted, the program search space can be very large. While the essential purpose of any genetic algorithm is to discover good solutions to a problem without resorting to an exhaustive evaluation of the search space, it is also desirable to ensure the search space itself is as small as possible.

One way of reducing the program space is through the use of *strong-typing*, where the tree builder, which is responsible for constructing the programs (and new mutations), is constrained to connect only nodes which make semantic sense. For instance, the *LessThan* operator, which returns a boolean value, should not be supplied as an argument to the *division* operator - execution could lead to divide-by-zero errors and a rather inconsistent logic. Although Koza developed a clunky solution to this problem, Montana[50] implemented a means of strong typing to GP toolkits by introducing the notion of a return type, which would have to be matched according to each node’s argument types, as is the case in the strongly-typed programming languages. As an example the *add* operator, should insist that both its arguments have an *numeric* return type; the operator itself will also return a *numeric* value.

However, Montana’s types, such as as *numeric*, or *boolean* have the disadvantage of being too prescriptive. If one is looking for a count for a loop, for instance, a floating point number is not necessarily appropriate — it is better to involve an integer. However, if one defines “integer” as a new *type* to accommodate this, then the arithmetic operators would no longer accept it, since it cannot be both numeric and an integer. Essentially a node should be able to return more than one type, for instance to say that it returns a number that is also an integer: indicating a hierachy of inheritance among different types. SXGP includes this flexibility, where each node may implement *multiple* return types. This permits the system to make use of more complex structures while making fewer trees that don’t make semantic sense.

[7]Deep cloning refers to copying both an instance and all of its class members; shallow cloning will clone the instance, but will not copy class members

Tree Checking The author's stronger-typing framework extends further. Although GP is capable of evolving many means of procrastination, the stronger-typing approach still permits the creation of "useless" code that performs no function. Code may be considered useless if it is never executed, or if it always returns the same value regardless of input. The former is usually to be found in branches of if-statements whose condition is accidentally a constant.

The tree builder in the author's GP toolkit generally avoids creating such code by following additional criteria, imposed by the function nodes themselves. For instance comparative nodes (*lessThan, moreThan, between, equals*) additionally insist that at least one of the child nodes in the subtree beneath each be some kind of image feature (a separate sub-type, based on the numeric type). This ensures that every sub-tree has the potential to perform some useful processing[8]. This further reduces the size of the search space.

The system also discards those programs that do not make use of features at all. GP programs, if left unchecked, may take advantage of the *a priori* probabilities of solutions and simply return the most popular!

Automatic Optimisation The author's GP system automatically optimizes the best individual at the end of evolution, in order to make it more suitable for deployment in the computer vision task for which efficiency is always a criterion. After running the individual on a set of training data, the system collects statistics about each node, then removes nodes that are never used and replaces nodes that always return the same value with constants.

Instant Deployment Solutions evolved by SXGP can be deployed for use in other applications immediately, in two modes:

GP Tree The GP tree is used as-is, so the program can be used immediately. Since the tree is executed in an interpreted manner, the program will not run as efficiently as possible.

Compiled Java The GP tree is converted into Java code, then compiled automatically. This kind of program will run significantly faster, but requires the Java process to be restarted or the use of class loaders in order to load the newly compiled code.

[8]Although it is still not perfect: the lessThan function could compare a feature whose return values are in the range 0-255 to a value of 1000, which would again always return the same value. However, SXGP's automatic optimisation mechanism will remove this kind of code.

Live Evaluation As well as providing a full-featured graphical interface to display the status of the evolutionary system, the performance of the current best GP program on seen and unseen images can be evaluated live by the user during evolution. This enables the user to assess how well the evolved solution is working and generalising, and provides an insight into how difficult a feat of image processing is: the user may subsequently tailor the training set to incorporate characteristics that apparently were not emphasised sufficiently in initial training set.

5.1.4 Parameter Choices for a Generic System

One can think of a genetic programming system as a whole “world” in which the birth, life and death of thousands of individuals are simulated by a series of sub-components. As such, one of the problems of developing solutions with GP is the sheer number of parameters required by each component: there are parameters for the minimum and maximum initial depth of individuals, the kind of tree builder to use, the population size, the number of generations, choice of fitness function, and so on. Moreover, it is difficult to assess the true effect of any parameter since all the components are rather inter-related. The work published by other researchers indicates that parameter tuning and experimentation is often called for, which doesn’t bode well for a supposedly generic problem solver. In order to maintain the aspiration of a generic system, it is necessary to consider the research and theory behind different parameter choices in order to establish a set of parameters that work well on a wide range of problems. Some of these are discussed below.

Genetic Operators A standard breeding pipeline involving crossover, mutation and reproduction operators is used in SXGP. The optimal “blend” of the learning operators crossover and mutation is something not all researchers agree on. Koza himself used no mutation, preferring a 100% crossover pipeline. Other researchers[63] have shown that genetic programming can proceed without using crossover at all, instead by using 100% mutation[9] Work by [64] shows that different blends of each of these operators do not make a substantive difference to the outcome. Although crossover and mutation work in different ways, their purpose is similar: to explore the ways in which useful trees can be improved.

[9]The author’s own toolkit, SXGP, for several months featured a silent bug in which crossover did not function at all. The problem was only noticed later — the bug had had little apparent effect on the results by GP, much to the embarassment of the author.

Given these results, the author used a reasonably "standard" set of values, making use of both operators, in which crossover produces about 75% of the new population; mutation 20%; the remainder obtained simply by replication. Making use of both operators is desirable, at least from a theoretical point of view, since each has its own specific advantages. Crossover helps to exchange useful building blocks between good parents, while mutation can re-introduce subtrees that might otherwise fall out of the population and thus helps maintain diversity.

Tournament Size Genetic Programming is distinguished from random search by its bias toward more performant individuals, which is achieved through the evaluation of individuals' performance (see Section 5.2) and subsequent selection of "good" individuals to produce next generation. The most widely used selection technique, tournament selection, selects t individuals from the population to form a tournament, then selects the individual with the best fitness score. Larger tournament sizes t will increase the selective pressure, albeit at the expense of variability. Other selection methods were discussed on page 38.

Although the genetic algorithm community has traditionally used a tournament size $t = 2$ which preserves as much diversity as possible within the population, while maintaining some degree of selective pressure, GP researchers generally use $t = 7$ as this is thought to instill higher selective pressure on the population, tending to improve the learning rate during the limited time of a GP run [10]. The author has conducted a number of comparative experiments using a variety of different tournament sizes, the result of which is that no significant difference could be detected in training accuracy, evolution time or average population size. Neither was observed that the performance of runs with different tournament sizes was any more variable for different values of t. As it is difficult to decide exactly which is the best parameter, one of the author's classification schemes errs on the safe side by using different sizes for different runs, and in general SXGP uses the more obvious compromise: a tournament size of $t = 5$.

Population Size The computational expense of an evolutionary run is usually proportional to the product of two parameters which specify the breadth and length of the search: namely the population size and generation count. Populations are used by evolutionary learning algorithms to store a diverse base of knowledge that can be built upon. Although the degree of selectivity employed has a significant effect on the variability within a population, such

[10]GA is potentially faster to run than GP, so can be indulged in evaluating more generations

variability can only be stored within a population of solutions of sufficient size. While there is clearly a lower bound for the population size, there is no practical upper bound, since the search space is potentially enormous.

An impression of the size of the search space can be attained through a brief example. If one imagines a GP system which uses n functions each of arity 2[11], a single feature, and maximum tree depth d, using a full tree builder (see Page 35), the search space, or number of potential programs is:

$$size = f^{\left(\sum_{i=0}^{d-1} 2^i\right)}$$

From the exponential nature of this calculation, it can be seen that modest increases in the number of functions or the tree depth will quickly yield spectacularly large search spaces. The addition of continuous random constants can increase the size of the search space to infinite size, which in any case is such that the GP system will never be able to explore it comprehensively. Given that the population size can only ever be a tiny fraction of the size of the search space, its choice might be considered relatively arbitrary: it doesn't appear necessary to tune for individual problems. Many GP researchers settle on a population size of 500. Nonetheless, the use of the ramped-half-and-half tree builder (see Page 35), introduces a significant level of duplication into the population (for reasons explained later), which is alleviated in SXGP using fitness caching (see 6.5).

Beyond the duplicates in Generation 0, genetic populations have a tendency to become dominated by a particular solution, a phenomenon described as premature convergence. Various "convergence manipulation" protocols exist to avoid the homogenisation of a population, including island selection, which was discussed on Page 47, but each introduce extra parameters to the system. [65] suggested the use of a different tournament selection algorithm which would repel individuals which shared a similar ancestry. The algorithm starts by selecting an individual at random[12], then searching for an appropriate partner to participate in crossover in a "repulsion tournament"[13]. The partner chosen is the one with the least shared ancestors with the first individual. Thus the crossover operator is forced to conti-

[11]The number of arguments taken by the function

[12]This is the protocol by Murphy and Ryan, in SXGP it starts by selecting an individual using tournament selection as normal, which seems to be rather more sensible

[13]Although the author feels the term "hereditary repulsion" is something of a missed opportunity; "Incest police" is rather more enjoyable terminology.

nue exploring previously uncharted areas of the search space, while ensuring the population maintains a higher degree of diversity. Hereditary Repulsion is integrated into SXGP.

5.1.5 Performance Considerations

The nature of evolutionary learning systems is to produce a lot of solutions then throw the majority of them away. Although this process is key to learning by simulated natural selection, it is also inherently wasteful. As a stochastic process, it is usually wise to run the Genetic learning system several times in order to estimate the true performance, which introduces further computational expense. Although the programs evolved are often very concise, the computational expense of learning by GP, especially on computer vision problems, is always a concern — and possibly one of the reasons members of the computer vision community do not often use GP! By contrast, techniques such as XCS[66] aim to evolve and improve a single system through less blunt reinforcement and thus may converge upon a solution more quickly. The nature of the genetic operators themselves is also of concern. The process of crossover, by which new combinations are found, can become more destructive than constructive. In nature, the chance happening of mutation is lethal to the individual in almost all cases [14]. In GP, however, mutation rates are much higher than would be the case in the natural world.

Although GP will always be more processor intensive than other techniques, there are ways in which one can preserve the learning capabilities of GP while significantly reducing the time required to evaluate a given number of individuals. In this section we shall briefly discuss two ways of doing so.

Bloat Control

A common issue in GP is the presence of useless code segments in individuals, known as 'code bloat,' that reduce the efficiency of the learning process. Indeed, one might argue that individuals that are padded out with unused, and thus expendable, pieces of code are more likely to survive the processes of crossover and mutation intact: there is a selective pressure which encourages the proliferation of code bloat.

However, the tendency to embelish programs with useless code reduces the efficiency of the evolutionary process, as it becomes increasingly less likely that the GP operators will have any noticeable effect at all – to the extent that the process of learning effectively ceases. In the author's experience, limiting the size of programs will often increase the rate of learning.

[14]Fortunately mutation is also a very rare occurrence

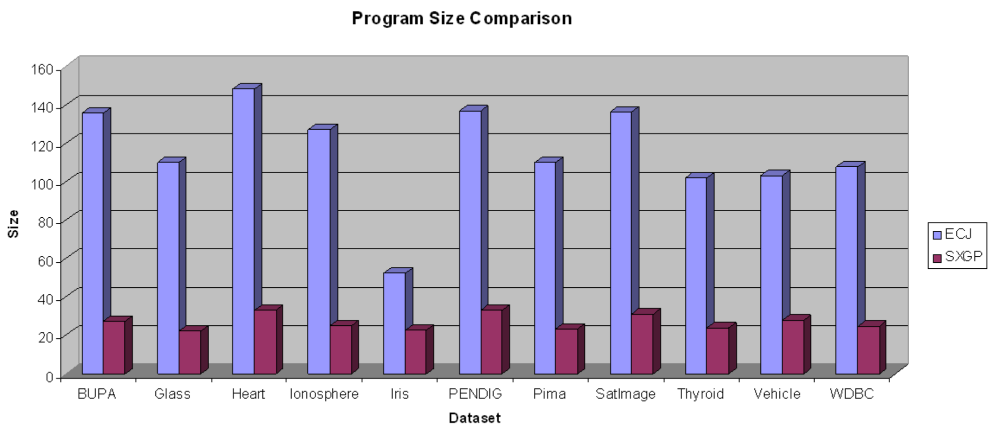

Figure 5.1: Comparison between best-of-run program sizes at 50 generations between SXGP and ECJ

Various techniques are used to reduce bloat, including parsimony pressure and dynamic maximum tree depth [67], which penalise or remove large individuals respectively. However, for a generic system, it is difficult to quantify what is "large", so an alternative solution was employed. Elitism, a commonly used technique which copies the best individuals in the population directly into the next generation, was used as studies have shown that evolution with elitism enabled has smaller average population sizes [32]. This may be because the direct copying into the next generation removes the selective pressure that encourages individuals to protect themselves with unused code.

SXGP's tournament selection mechanism will also choose smaller individuals in the case that two identically fit programs are competing for the same tournament. The tree checking and strong-er typing functions also prevent bloat code from becoming prevalent. Since SXGP does not allow many forms of inactive code, additional trees that have a negative effect on the individual's ability are quickly eliminated.

The effect of these additions can be seen in Figure 5.1, which compares the size of best-of-run programs for both SXGP and ECJ following an evolutionary run of 50 generations. Although the *max-tree-depth* parameter was set to the same value in both toolkits (all other parameters were similarly matched), ECJ tends to produce significantly larger individuals than does SXGP[15], whose best of run programs are generally a third of the size of ECJ programs. Consequentially, executing 500 individuals over the course of 50 generations takes about three times longer in ECJ.

[15]Although the SXGP equivalent is generally slightly fitter

5.2 The Evaluation of Fitness

Essential to any system which learns through a process of artificial selection is the “fitness” function that quantifies the performance of an individual, permitting direct comparisons to be made between individuals. Individuals which make fewer mistakes or otherwise perform well are assigned a lower fitness value[16].

There are various aspects of performance that one may wish to take into account during the calculation of fitness, in order that each be maximised as far as possible. Examples include as the efficiency of the individual, its ability to generalise, or its usage of as much information as possible. It may also be the case that some of these requirements are based on context, or even change dynamically during the course of the evolutionary process.

The most straightforward fitness function is defined as follows. Given a set of classes $C = \{c_1, ..., c_n\}$ and a set of samples $\{(x_1, y_1, w_1), ..., (x_n, y_n, w_n)\}$, with each sample comprising a feature vector $x \in X$, the correct class $y \in C$, and a corresponding weight $w = 1$. The fitness of function $f : X \to C$ may be calculated as:

$$fitness_1 = \sum_{i=1}^{N} w_i[f(x_i) \neq y_i)]$$

In other words, $fitness_1$ sums the weights of samples mistakenly classified. In the author’s experience this kind of fitness function performs well, in spite of its apparent simplicity. Since the function is calculated using only one measure, it is possible that a number of classifiers may be assigned identical fitness; in which case the tournament selector, when faced with two or more equally fit individuals, will choose the smallest. The training data error is thus the overriding factor in determining an individual’s fitness: other desirable features are secondary. The author’s experience is that the addition of other factors to the fitness function invariably harm its eventual accuracy.

5.2.1 Fitness for Detection Problems

Nonetheless, some problems do demand slightly more complex fitness functions which permit certain adjustments to be made, usually relating to the relative cost of making different kinds of mistake. A classifier can generally make two types of mistake: either a *false positive*, where

[16]In this sense “fitness function” is something of a misnomer: error function is more accurate. Measuring mistakes is easier, because an error value of zero signals to the GP system that an ideal solution has been found, prompting it to end the evolutionary process

a sample from another class is incorrectly identified as the class in question, or a *false negative*, where a genuine sample from the class in question is not identified as such. In some cases, notably medical diagnosis or credit management, the cost associated of each kind of mistake is different. When diagnosing disease the false positive may be alarming to the patient but is substantially less costly than the false negative, which can leave a person unaware of a potentially serious health problem which, upon its eventual discovery, would no doubt leave the medical establishment facing a substantial claim.

In machine vision, the relative costliness of different mistakes is dependent upon the domain domain. In many of the categorisation problems, such as digit recognition, it is difficult to establish which mistakes are more costly than others. Other tasks, especially those involving binary detection are different: it may be necessary to decide whether to evolve a highly sensitive classifier that doesn't make many false negatives, or a more efficient classifier that does miss out some outliers. In face detection, for instance, there are many more non-face objects than there are face objects, so it is beneficial to take the relative occurrence of each class into account in the fitness function.

The relative importance of different mistakes can be encoded by introducing extra coefficients into the fitness function, often as a ratio between α and β. An example used by other GP researchers[29] is shown below:

$$fitness_2 = \frac{\alpha TP}{totalTP + \beta FP}$$

The calculation of $fitness_2$ uses TP, the total weight of correctly identified true samples, $totalTP$, the total weight of positive training samples and FP, the total number of false positives. Oddly, this formula calculates $\alpha TP/totalTP$ as a percentage, but also adds FP to the denominator meaning that the affect of α relative to β is dependent on the total number of false samples $totalFP$. It occurs to the author that the following calculation makes more sense:

$$fitness_3 = \alpha \frac{FN}{totalFN} + \beta \frac{FP}{totalFP}$$

In $fitness_3$ the fitness is composed of two parts, the first relating to the total weight of false negatives and the second relating to the total weight of false positives. The relative importance of each kind of mistake is encoded using an appropriate $\alpha : \beta$ ratio. Still, in the author's opinion, the ideal way to tackle the problem is not to introduce an extra parameter into the data, but instead to use a training set whose samples and relative class sizes closely match the case in the real world.

5.2.2 Even Class Allocation

A consequence of evaluating performance entirely using a fitness value is that the GP system will become prone to falling into fitness minima. These minima can be most pronounced in situations where some classes are more prevalent than others. In these cases high fitness values do not necesarrily correspond to effective classifiers. For instance, faces account for only 2% of the CBCL face training set, reflecting that there are many more non faces in the world than faces. Nonetheless, the classifier could evolve a solution with accuracy exceeding 98% simply by returning false for everything. One way to approach this problem is either to allocate the same number of samples to each class, or to weight each sample such that each class has the same total influence.

5.2.3 Sample Weighting

Having considered the relative importances of different kinds of mistake, and whether one may wish to weight samples collectively, it is also worth considering whether individual samples should be allocated different weights according to their difficulty, since some samples will be easier to classify than others. By weighting the samples that appear more difficult (that are misclassified most often), it may be possible to improve performance of the classifier, or at least ensure that the system does not become mired in local minima.

A useful benchmark when considering this sort of approach is AdaBoost[58]. Adaboost is a meta learning algorithm which develops classifier ensembles over the course of several learning sessions. The weights associated with different training data samples are adjusted to influence the learning system to solve more difficult training data.

The algorithm works by calculating the error of the last learned classifier. The error consists of the sum of weights of each sample, so a classifier receives credit according to the importance of the samples it classifies correctly, rather than the number of correct classifications.

After the error is calculated the distributions are updated such that unsolved samples maintain their weight while solved samples have their weight reduced in proportion to how well the classifier solved the sample set as a whole. This process is repeated T times or until a further useful classifier cannot be found.

After training is complete, a strong classifier is constructed from the set of weak classifiers using their associated error values which give some indication of the confidence in the classifier. Every time a weak classifier returns a label, a value associated with that label is incremented by the confidence factor, with the label with the highest overall confidence level being returned.

Adaboost depends on a number of learning sessions, and as a result makes the search for the ensemble classifier much slower.

The author considered a different method, by which the difficulty of each item of training data be assessed at the end of every generation with its weight adjusted accordingly. Regretfully this strategy failed miserably.

5.2.4 Encouraging Generalisation Ability

The fitness functions thus far have concentrated on the classifier's error on a given training set. However, a common problem in non-parametric classification is overfitting, where a classifier learns the behaviour of a specific set of data set so well that its ability to work on novel data is compromised. It is desirable, therefore, to evolve classifiers that can learn to *generalise*. As has been shown elsewhere in this thesis (57) different techniques may produce a lower fitness score on training data, but nonetheless outperform the fitter classifier on test data. Given a set of individuals evolved by a learning function, choosing the best of them according to training data error alone is therefore somewhat insufficient.

The usual solution is to reserve a portion of the training set for the purpose of validating the classifier; the performance on the validation set gives a better indication of the classifier's ability to generalise. Of course, once the validation set has been used for this purpose it is no longer an unbiased estimator of performance, so a completely unseen test set is also employed to make the final assessment of a classifier's capability.

Training Set Splitting Of course, the use of a validation set takes away a proportion of samples from the training set that would otherwise be used for training, which itself may have an effect on the generalisation ability of the classifier.

The author devised an alternative method, intended to gain information about the classifier's generalisation ability without sacrificing any data, and without resorting to computationally costly methods such as cross validation or jackknifing.

The technique involved splitting the training data into two sets, which are evaluated separately. The classifier was evaluated on each set, with the sum of errors yielding the training fitness before. To the error, the *difference* between the two sets is also added, thus imposing a fitness pressure toward *consistency*. However, it was not possible to detect a significant improvement; indeed this algorithm sometimes worsened the performance of the classifier both on training and test datasets, perhaps reflecting that the two sub datasets could not necessarily be compared directly. This experiment reveals once more the perils of tinkering with fitness functions.

The Validation Dataset Discarding the author's penchant for experimentation for one moment, the conventional way to use validation sets is to leave them untouched until the evolution process has completed, then to use them as estimators of the classifier's performance on unseen data. This allows the generalisation capability of the classifier to be assessed without affecting the delicate fitness function. As GP evolution sessions typically involve producing a number individuals during the course of several runs[17], their performance on validation data, as opposed to their training fitness, can be used to select the most promising individual. A similar process can be used when trying out classifiers produced by classifier fusion.

One significant advantage of using validation data in this way is that the GP evolution time is shortened, since the training set is reduced in size. The proportion of data to use for validation sets poses a dilemma: choosing too high a proportion may make the training set too small, while choosing too low a proportion may make the validation set unreliable. However, the results from the authors experiments with validation results did not reveal any significant advantage of using validation sets.

5.3 Comparisons

Having discussed the GP toolkit in more detail, we now move onto some empirical comparisons, using the datasets summarised on Page 1.

5.3.1 Dataset Interface

Of the selection of 11 datasets used throughout this thesis, four include their own test set permitting validation on unseen test data, which provides an indication of a given classifier's

[17]This is done to avoid the effects of initial populations. The GP environment should always converge on an equally fit solution given enough time, but undertaking multiple runs is the most reliable way to ensure that a particular result is representative

ability to generalise. The rest are validated using 10 fold cross validation, a standard technique for measuring error. Cross validation is usually performed by splitting the dataset into 10 sub-sets, from which one is chosen as a test set, with the rest used for training. The learning process is invoked for every test/training combination, so the overall training error can be averaged, and the test error estimated. The author's implementation of k-fold cross validation uses a *stratified* selection policy to generate the folds, which ensures that the relative class distributions within each sub-set are kept as similar to that of the main set as possible. Some benchmarks, such as the Vehicle dataset, define the folds explicitly; these types are also supported. The author's interface permits each type of dataset to be delivered to the classification system, whether it be one of the public training datasets, or a dataset generated from images.

5.3.2 Comparing with ECJ

Before going on to evaluate SXGP in a more general sense, it is necessary to compare it to a benchmark GP implementation. For this comparison, the popular ECJ toolkit[6] was chosen, which at time of writing was available in its 18^th version. Like the author's own toolkit, ECJ is implemented in Java so comparisons in terms of performance are meaningful. ECJ's default implementation is, as far as possible, based on Koza's original specification, making it a suitable benchmark. Finally, and despite a rather complex architecture, ECJ is surprisingly efficient — so much so that this author had to expend some considerable time on optimising his own toolkit first!

Here, each toolkit is run on the same problems involving the classification of data. In order to make the comparison as meaningful as possible, most of the novel features in SXGP were deactivated such that the two toolkits could compete on an equal footing. As far as possible, equivalent classification problems were coded using each toolkit.

20 classifiers were evolved by each toolkit over the course of 20 runs, each lasting for 50 generations. The average test fitness of these classifiers on a series of datasets is presented in Table 5.2. The results are illustrated graphically in Figure 5.2. The difference between classifiers, and the results of an independent samples T test are also shown. The results show that SXGP is able to compete very effectively with ECJ: SXGP produces the better classifiers on every occasion where the difference between classifiers' average fitness is significant.

It is interesting to question why ECJ's performance is not more equivalent with SXGP, given the similarity of the problem implementations and otherwise identical parameters. One

Dataset	ECJ	SXGP	Better	Significance
BUPA	0.394 ± 0.024	0.376 ± 0.026	SXGP, 4.5%	Very Significant
GLASS	0.429 ± 0.023	0.417 ± 0.031	–	Not Significant
HEART	0.241 ± 0.019	0.223 ± 0.018	SXGP, 7.5%	Very Significant
IONOSPHERE	0.120 ± 0.015	0.101 ± 0.014	SXGP, 15.8%	Extremely Significant
IRIS	0.109 ± 0.030	0.054 ± 0.013	SXGP, 50.5%	Extremely Significant
PENDIG	0.408 ± 0.032	0.405 ± 0.026	–	Not Significant
PIMA	0.257 ± 0.008	0.247 ± 0.008	SXGP, 3.9%	Very Significant
SATIMAGE	0.245 ± 0.027	0.252 ± 0.023	–	Not Significant
THYROID	0.029 ± 0.008	0.023 ± 0.003	SXGP, 14.8%	Very Significant
VEHICLE	0.394 ± 0.014	0.395 ± 0.017	–	Not Significant
WDBC	0.057 ± 0.008	0.041 ± 0.005	SXGP, 50.5%	Extremely Significant

Table 5.2: Toolkit Comparison: SXGP Vs ECJ v18 on various datasets. Averages from 20 runs, $\pm$ value indicates the standard deviation

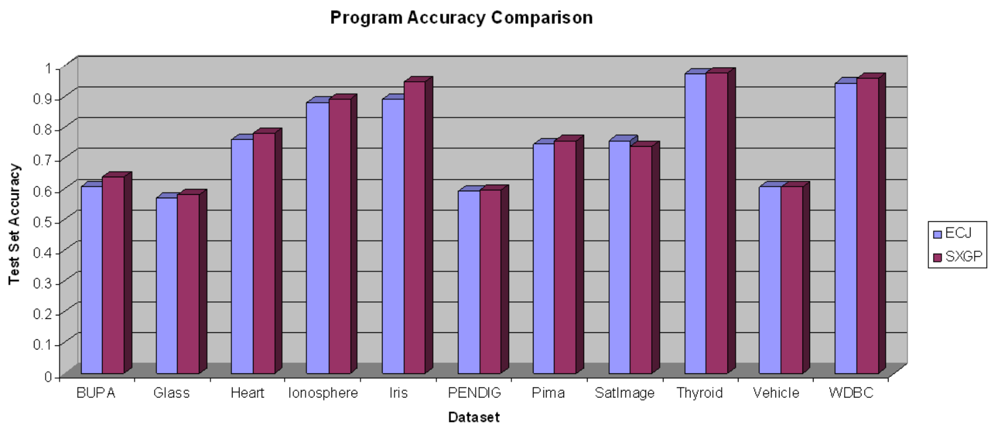

Figure 5.2: Comparison between best-of-run program test errors after 50 generations between SXGP and ECJ

answer may be found in the average program size of individuals evolved by ECJ and SXGP respectively, already presented in Figure 5.1, which shows that ECJ produces substantially larger individuals most of the time. Since SXGP can achieve equivalent or better solutions with substantially smaller individuals, it is fair to say that the additional size is a result of bloat, as opposed to useful code. As well as reducing the efficiency of GP learning, bloat can prevent learning from taking place, by turning the GP system into an automatic means of swapping useless code fragments.

These results suggest that the author's toolkit is comparable to a standard and widely used Genetic Programming toolkit, which has been under continuous development for the best part of a decade[18]. This experiment justifies the author's decision to use SXGP exclusively as the GP toolkit in this thesis.

5.3.3 Comparing to other GP Results

In the second process of validation, the author's GP Toolkit was compared to published results obtained by other GP researchers in order to establish whether the SXGP classifier is competitive with the state of the art in Genetic Programming in a more general sense. SXGP was put to work on certain public datasets, for which figures had already been published by other GP researchers. The results of the experiment are presented in Table 5.3.

The results in Table 5.3 show that the author's Genetic Programming toolkit consistenly produces classifiers that are very competitive with other published results, although as usual there is no single best technique for every dataset. The author's work is based most closely upon the work by Loveard [68]; it can be seen that the author's toolkit delivers significantly improved results over the work published by Loveard.

5.3.4 Comparing to other Classification Techniques

In this final comparison, the author's toolkit is compared to other techniques from outside the Genetic Programming community in order to establish the extent to which evolved classifiers are competitive with human-written learning algorithms. The comparisons are performed once again using publicly available datasets. This has already been done in the past: a previous study by Lim [71] compared 33 different classification algorithms on a series of public datasets. Equivalent results using Genetic Programming were later added by [68]. With

[18]This particular comparison is valid only for problems of this type, using GP: ECJ does support considerably more evolutionary learning paradigms than does SXGP

Data Set	sXGP	Loveard[68]	Chien [69]	Bot[51]	Muni[47]	[70]
BUPA	71.3* $\pm$ 1.6	69.2 $\pm$ 1.6			-	
Heart	84.8* $\pm$ 1.7				-	
Glass	67.3* $\pm$ 2.0			58.9 $\pm$ 11.4	-	
Ionosphere	95.7* $\pm$ 0.8		92.8 $\pm$ 2.3	90.2 $\pm$ 5.5	-	
Iris	99.4* $\pm$ 0.1		95.3 $\pm$ 1.0		98.7 $\pm$ 0.0	
Pen-Digits	92.0* $\pm$ 2.3				-	83.1
Pima	76.5* $\pm$ 0.8	75.8 $\pm$ 0.8			-	
SatImage	87.8* $\pm$ 0.5	80.7 $\pm$ 1.3			-	81.4
Thyroid	98.9* $\pm$ 0.3	97.6 $\pm$ 0.2			-	
Vehicle	73.0 $\pm$ 3.1	62.4 $\pm$ 2.9	75.3* $\pm$ 2.4		61.8 $\pm$ 0.1	
WDBC	97.1 $\pm$ 0.1	96.4 $\pm$ 0.2	-		97.2* $\pm$ 0.0	

Table 5.3: Results Compared to Those Published by Other GP Researchers

the algorithms compared in rank order, one particular GP representation (DRS, discussed in Chapter 4) was found to be reasonably competitive with the 33 other techniques at solving binary problems (ranking as high as fourth), but in general the GP programs' rankings were mediocre.

The study by Lim was dominated by a number of decision tree algorithms; the results here also include comparisons to other, more recent classifiers. The algorithms compared here include Support Vector Machines (SVM), Multilayer Perceptron Neural Networks (MLP), the C4.5 Decision Tree Algorithm, and various implementations of k-Nearest-Neighbour algorithm (kNN). The results of this comparison on test data are presented in Table 5.4.

As well as confirming the phenomenon that there is no such thing as a *best* classifier that consistently outperforms other algorithms, the results show that SXGP is competitive with a range of other techniques from outside the realm of evolutionary computation on a series of different problems. It should be noted that the same GP implementation with identical parameters was used to derive all of the SXGP results.

The Vehicle Dataset, in which silhouettes of four different vehicle types must be established, is difficult to solve. This is not to say that the problem itself is intractable — our eyes

Data Set	SXGP	SVM	MLP	C4.5	kNN
BUPA	71.3	76.1	73.1	68.1	64.9
Heart	84.8	87.4	82.0	78.9	83.8
Glass	67.3	68.6	75.2	68.2	72.0
Ionosphere	95.7	93.2	96.0	94.9	98.7
Iris	98.0	98.0	96.0	95.3	95.7
Pen-Digits	92.0	97.5	93.4	96.6	-
Pima	76.5	77.2	76.4	73.0	76.7
SatImage	87.7	88.4	91.0	86.3	90.9
Thyroid	98.9	96.1	99.3	92.6	97.9
Vehicle	70.9	79.0	79.3	73.4	72.8
WDBC	97.1	97.2	96.7	94.7	97.1
Average	85.5	87.1	87.1	83.8	85.5

Table 5.4: SXGP versus Other Techniques on 11 Public Datasets

can differentiate all kinds of silhouettes accurately. Rather it is the limitations in the attributes chosen to describe each silhouette that make the problem difficult since a certain amount of important information is lost. Many of the UCI datasets were constructed from a finite set of measurements taken during the course of manual experimentation. However a computer can take literally thousands of measurements from a given image without very little effort, leaving one spoiled for choice. It is difficult to decide which attributes would aid accurate classification and which would be irrelevant. In fact, irrelevant attributes may actually decrease the performance some classification algorithms, such as kNN. Different feature sets which may facilitate classification will be investigated in Chapter 6.

5.4 A Discussion

Before moving onto the vision aspects of this thesis, the author shall complete this Chapter with a short discussion of classification in general.

The many classification algorithms can ordinarily be split into two groups: parametric and non parametric. The parametric methods generally aim to identify a set of coefficients, one for each variable, that express a combination of features whose output Y can predict a certain event. Techniques such as Fisher's Linear Discriminant[REF] estimate the values of these coefficients using the least squares technique. Since Y is considered a probability, its value can be combined with prior probabilities using Baye's theorem to obtain an answer that takes into account the relative abundances of different classes. Although the disciminant methods are intuitive, like all statistical techniques they are reliant upon on a number of rigid assumptions, such as feature independence, normal distributions or homoscedasticity, which rarely hold true for real world data.

A more pressing problem is that this kind of approach cannot accurately model non linear relationships in data. The so-called Generalised Linear Model[REF] is an extension which permits models to work with non-normally distributed data distributions, particularly those with an exponential distribution, using some form of link function. Of these, Logistic Regression, which also places fewer assumptions on the data, is one of the most extensively used techniques for parametric classification.

The advantage of the non-parametric methods is that they do not place arbitrary restrictions on data as they are generally not based on statistical theory. They usually require more training data in order to produce accurate solutions and often take longer to train than do the simpler statistical methods.

One popular family of non-parametric techniques are the recursive partitioning algorithms, or decision-tree algorithms, of which the most well known is probably Quinlan's C4.5[REF]. The objective of the decision tree algorithms is to recursively split the dataset into two or more increasingly homogenous subgroups in order to improve the classification of a target variable. Various techniques exist to choose the point at which the data is split, but the general idea is that the root node is split by the variable that best divides the training data and so on.

Although decision tree algorithms generally exhibit good performance as classifiers, and have the advantage of easy interpretation, they also suffer from two drawbacks. The first, more theoretical, drawback is that generally the splits can only be performed orthogonally with respect to the features, so the regions in space cut out by decision tree algorithms are hyper-rectangles. The algorithm may perform badly if the decision planes are actually diagonal. The second problem is that they are prone to over fitting, which occurs naturally if the splitting process is permitted to continue unchecked, although post processing "pruning" techniques generally alleviate this problem.

The most significant advantage of the non-parametric techniques is that they can develop arbitrarily complex models and thus are likely to outperform parametric statistical techniques, provided they do not over fit the data. Parametric techniques, which are usually fast to execute may be useful when deciding which variables are important and which are irrelevant.

The k Nearest Neighbour algorithm is another non-parametric technique and an example of instance-based learning. Novel feature vectors are compared to a database of the training data each according to some distance function, usually the Mahalanobis distance (which has the advantage of scale invariance). The most common class among the k nearest of its "neighbours" is chosen as the appropriate label. As such, the training procedure is trivial, although the classification stage, to which the computational effort is deferred, can be slow when for large datasets, in addition to its large storage requirement. The choice of k is also difficult to ascertain in a principled way, although in general small values where $k > 1$ make the algorithm more robust, while the choice of $k=1$ is generally best for the smallest datasets ($N < 100$).

Neural networks are also often employed in classification. While Rosenblatt's[REF] classic single-layer perceptron shares much in common with a basic linear regression approach, the neural networks generally used for classification are multi-layer perceptrons[REF], which are capable of learning complex non linear relationships between data, due to the usually non-linear activation functions working within each neuron.

A problem with neural networks in general is that the choice of network topology has a significant impact on the viability of the network after it has been trained. The hidden layer, if too small, may render the network incapable of suitable approximation; if too large the net also becomes prone to over fitting. Irrelevant features can be a danger for neural networks, as they making the back propagation learning process significantly slower.

A closely related approach to neural networks are Support Vector Machines (SVMs), one of the newest non-parametric supervised learning techniques. Like discriminant analysis, SVMs aim to locate some form of decision plane that divides clusters of data, but SVMs are non linear and also aim to maximise the margin between the plane and the groups' outliers, so wherever possible, the plane that provides the largest gap between groups is chosen. This provably reduces the upper bound on the expected generalisation error.

A significant advantage of SVMs is that the number of features in the training data does not adversely affect the performance of the completed SVM classifier, since the number of support vectors selected by the algorithm, which define the hyper plane, is usually small. On the other hand, SVMs are generally unable to classify more than two classes at once, which calls for some form of binary decomposition, and certain parameter choices can leave the SVM prone to overfitting its model to training data.

Generally speaking, SVMs and Neural networks tend to perform better when dealing with high dimensional spaces and continuous features. Categorical data is best dealt with by the decision tree algorithms. As more sophisticated algorithms, however, both require more parameters to be set that then other techniques.

Apart from the decision trees, non-parametric methods generally manifest themselves as "black boxes" since it is difficult to understand how or why they reach a particular decision. This is a particular disadvantage in areas of business, such as credit scoring, or medicine, where people are uneasy leaving important decisions to an algorithm they don't understand, but is less of a concern when producing vision systems.

Many of the algorithms, both parametric and non-parametric, are in some way dependent upon the initial random values assigned to certain values, whether it be parametric coefficients or neuronal weightings. An advantage of the population approach to learning, employed by both Genetic Algorithms and Genetic Programming, is that the solutions cover a broader surface of the search space instead of a single point, so initial random choices are less of an issue. Of course, the population paradigm places an additional computational overhead so different runs may yet provide different performances if not run long enough.

Although the results in this Chapter show that Genetic Programming is competitive with state-of-the-art techniques on given problems, perhaps more so than have done previous GP researchers, the results also correspond with a long line of comparative studies which all state that there exists no "magic bullet" solution to problems involving classification.

Furthermore, although this discussion draws attention to the pros and cons of each technique in an analytical sense, these issues do not necessarily manifest themselves in real world scenarios: A study by Domigos and Pazzani (1997)[REF] found that the naive Baye's classifier, despite its requirement for feature independence, could sometimes be superior to more sophisticated algorithms.

Nonetheless the most appropriate course of action is to select the classification technique which appears most appropriate for the task. Given the kind of vision tasks tackled within this thesis, there are certain techniques that can be ruled out. Linear discriminant analysis, for instance, generally requires that the classes have equal numbers of members, which is rarely the case in vision and certainly not in generic vision. kNN places a similar requirement and like neural networks is regarded as sensitive to noise and irrelevant features, both of which may be present in images.

Evolutionary techniques are not often mentioned in discussions of classification. Unlike the above approaches, genetic techniques are particularly bound by any given representation: GAs can be used to determine neural network weights, or weights for disciminant functions in place of the least squares technique. GP can readily develop decision trees, and the concept of a margin can be built into a GP fitness function to offer advantages similar to SVMs. Since the problem of classification requires different approaches for different tasks, it makes sense to make use of the most flexible technique in our learning framework. In the following Chapters we shall see how well equipped is Genetic Programming for developing generic vision systems.

Bibliography

[1] N. Eldredge and S. J. Gould. *Models in Paleobiology: Punctuated Equilibria: An Alternative to Phyletic Gradualism*, chapter 5, pages 82–115. Freeman, Cooper and Co, 1972.

[2] Andrew Parker. *In the Blink of an Eye*. Perseus Publishing, 2003.

[3] Paul Viola and Michael Jones. Robust real-time object detection. *International Journal of Computer Vision*, 57(2):137–154, 2004.

[4] John R. Koza. *Genetic Programming*. MIT Press, 1992.

[5] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[6] Sean Luke. ECJ: A Java-based evolutionary computation research system v14. `http://cs.gmu.edu/~eclab/projects/ecj/`.

[7] F. John Canny. A Computational Approach to Edge Detection. 8(6):679–698, 1986.

[8] E.R. Davies. On the noise suppression and image enhancement characteristics of the median, truncated median and mode filters. *Pattern Recognition Letters*, 7:87–97, 1988.

[9] Roongroj Nopsuwanchai and Prabhas Chongstitvatana. Improving robustness of robot programs generated by genetic programming for dynamic environments. In *In Proc. of Asia-Pasific Conference on Circuits and Systems (APCCAS98*, pages 523–526, 1998.

[10] Wei Yan and Christopher D. Clack. Evolving robust gp solutions for hedge fund stock selection in emerging markets. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 2234–2241, New York, NY, USA, 2007. ACM.

[11] Christopher Harris and Bernard Buxton. Evolving edge detectors with genetic programming. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 309–315, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[12] C.-H. Chao and A. P. Dhawan. Edge detection using Hopfield neural network. In S. K. Rogers and D. W. Ruck, editors, *Proc. SPIE Vol. 2243, p. 242-251, Applications of Artificial Neural Networks V, Steven K. Rogers; Dennis W. Ruck; Eds.*, volume 2243 of *Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference*, pages 242–251, March 1994.

[13] Byatia P. Srinivasan V. and S.H. Ong. Edge detection using a neural network. *Pattern Recognition*, 27(12):1653–1662, 1994.

[14] S.M. Bhandarkar, Y.Q. Zhang, and W.D. Potter. An edge-detection technique using genetic algorithm-based optimization. *Pattern Recognition*, 27(9):1159–1180, September 1994.

[15] Christopher Harris and Bernard Buxton. Low-level edge detection using genetic programming: performance, specificity and application to real-world signals. Technical Report RN/97/7, Gower Street, London, WC1E 6BT, UK, 1997.

[16] Mark E. Roberts and Ela Claridge. An artificially evolved vision system for segmenting skin lesion images. In Randy E. Ellis and Terry M. Peters, editors, *Proceedings of the 6th International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, volume 2878 of *LNCS*, pages 655–662, Montreal, Canada, November 2003. Springer-Verlag.

[17] Riccardo Poli. Genetic programming for image analysis. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 363–368, Stanford University, CA, USA, 28–31 1996. MIT Press.

[18] David G. Lowe. Object recognition from local scale-invariant features. pages 1150–1157, 1999.

[19] Herbert Bay, Tinne Tuytelaars, and Van Gool. Surf: Speeded up robust features. In *9th European Conference on Computer Vision*, Graz Austria, May 2006.

[20] M. Roberts and E. Claridge. Cooperative coevolution of image feature construction and object detection, 2004.

[21] Jean Serra. *Image Analysis and Mathematical Morphology*. Academic Press, Inc., Orlando, FL, USA, 1983.

[22] Polina K. Spivak. Discovery of optical character recognition algorithms using genetic programming. In John R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 2002*, pages 223–232. Stanford Bookstore, Stanford, California, 94305-3079 USA, June 2002.

[23] David Andre. Automatically defined features: The simultaneous evolution of 2-dimensional feature detectors and an algorithm for using them. In K. E. Kinnear Jr, editor, *Advances in Genetic Programming*. MIT Press, 1994.

[24] Huey et al. Rapid evolution of a geographic cline in size in an introduced fly. *Science*, 287(5451):308–309, 2000.

[25] Ingo Rechenberg. *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. PhD thesis, Technical University of Berlin, 1971.

[26] John H. Holland. *Adaptation in Natural and Artificial Systems*. 1975.

[27] R.A Fisher. *The Genetical Theory of Natural Selection*. Dover, 1958.

[28] Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.

[29] Simon C. Roberts and Daniel Howard. Evolution of vehicle detectors for infrared linescan imagery. In Riccardo Poli, Hans-Michael Voigt, Stefano Cagnoni, Dave Corne, George D. Smith, and Terence C. Fogarty, editors, *Evolutionary Image Analysis, Signal Processing and Telecommunications: First European Workshop, EvoIASP'99 and EuroEcTel'99*, volume 1596 of *LNCS*, pages 110–125, Goteborg, Sweden, 28-29 May 1999. Springer-Verlag.

[30] Ankur Teredesai and Venu Govindaraju. Issues in evolving GP based classifiers for a pattern recognition task. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation*, pages 509–515, Portland, Oregon, 20-23 June 2004. IEEE Press.

[31] Alan Piszcz and Terence Soule. Dynamics of evolutionary robustness. In Maarten Keijzer, Mike Cattolico, Dirk Arnold, Vladan Babovic, Christian Blum, Peter Bosman, Martin V. Butz, Carlos Coello Coello, Dipankar Dasgupta, Sevan G. Ficici, James Foster, Arturo Hernandez-Aguirre, Greg Hornby, Hod Lipson, Phil McMinn, Jason Moore, Guenther Raidl, Franz Rothlauf, Conor Ryan, and Dirk Thierens, editors, *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 871–878, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[32] Riccardo Poli, Nicholas F. McPhee, and Leonardo Vanneschi. Elitism reduces bloat in genetic programming. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Atlanta, Georgia, USA, 12-16 July 2008. ACM Press. forthcoming.

[33] Darrell Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA, 1989. Morgan Kaufman.

[34] T. Blickle and L. Thiele. A mathematical analysis of tournament selection, 1995.

[35] Riccardo Poli. Tournament selection, iterated coupon-collection problem, and backward-chaining evolutionary algorithms. In Alden H. Wright, Michael D. Vose, Kenneth A. De Jong, and Lothar M. Schmitt, editors, *Foundations of Genetic Algorithms 8*, volume 3469 of *Lecture Notes in Computer Science*, pages 132–155, Aizu-Wakamatsu City, Japan, 5-9 January 2005. Springer-Verlag.

[36] Tatsuya Motoki. Calculating the expected loss of diversity of selection schemes. *Evol. Comput.*, 10(4):397–422, 2002.

[37] P. Riccardo and L. William. Backward-chaining genetic programming, 2005.

[38] Huayang Xie, Mengjie Zhang, and Peter Andreae. Another investigation on tournament selection: modelling and visualisation. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1468–1475, New York, NY, USA, 2007. ACM.

[39] Artem Sokolov and Darrell Whitley. Unbiased tournament selection. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1131–1138, New York, NY, USA, 2005. ACM.

[40] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. TCGA Report No. 89003, 1989.

[41] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[42] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 6, pages 111–134. MIT Press, Cambridge, MA, USA, 1996.

[43] Riccardo Poli and William B. Langdon. On the search properties of different crossover operators in genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 293–301, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[44] Walter Alden Tackett. Greedy recombination and genetic search on the space of computer programs. In L. Darrell Whitley and Michael D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 271–297, Estes Park, Colorado, USA, 31 July–2 August 1994. Morgan Kaufmann. Published 1995.

[45] Kevin J. Lang. Hill climbing beats genetic search on a boolean circuit synthesis of Koza's. In *Proceedings of the Twelfth International Conference on Machine Learning*, Tahoe City, California, USA, July 1995. Morgan Kaufmann.

[46] Mengjie Zhang, Xiaoying Gao, and Weijun Lou. A new crossover operator in genetic programming for object classification. *IEEE Transactions on Systems, Man and Cybernetics, Part B*, 37(5):1332–1343, October 2007.

[47] Durga Prasad Muni, Nikhil R Pal, and Jyotirmay Das. A novel approach to design classifier using genetic programming. *IEEE Transactions on Evolutionary Computation*, 8(2):183–196, April 2004.

[48] David J.C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[49] Kenneth Alan De Jong. *An analysis of the behavior of a class of genetic adaptive systems.* PhD thesis, Ann Arbor, MI, USA, 1975.

[50] David J. Montana. Strongly typed genetic programming. Technical Report #7866, 10 Moulton Street, Cambridge, MA 02138, USA, 7 1993.

[51] Martijn C. J. Bot and William B. Langdon. Application of genetic programming to induction of linear classification trees. In Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 247–258, Edinburgh, 15-16 April 2000. Springer-Verlag.

[52] Yun Zhang and Mengjie Zhang. A new program structure in genetic programming for object classification. In David Pairman, Heather North, and Stephen McNeill, editors, *Proceeding of Image and Vision Computing NZ International Conference*, pages 459–465, Akaroa, New Zealand, November 2004. Lincoln, Landcare Research.

[53] Thomas Loveard and Victor Ciesielski. Representing classification problems in genetic programming. In *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1070–1077, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[54] Mengjie Zhang, Victor B. Ciesielski, and Peter Andreae. A domain-independent window approach to multiclass object detection using genetic programming. *EURASIP Journal on Applied Signal Processing*, 2003(8):841–859, July 2003. Special Issue on Genetic and Evolutionary Computation for Signal Processing and Image Analysis.

[55] W. B. Langdon and B. F. Buxton. Genetic programming for combining classifiers. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 66–73, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[56] Polina K. Spivak. Discovery of optical character recognition algorithms using genetic programming. In John R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 2002*, pages 223–232. Stanford Bookstore, Stanford, California, 94305-3079 USA, June 2002.

[57] Will Smart and Mengjie Zhang. Using genetic programming for multiclass classification by simultaneously solving component binary classification problems. Technical Report CS-TR-05-1, Computer Science, Victoria University of Wellington, New Zealand, 2005.

[58] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *European Conference on Computational Learning Theory*, pages 23–37, 1995.

[59] Thad Starner, Joshua Weaver, and Alex Pentland. Real-time american sign language recognition using desk and wearable computer based video. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(12):1371–1375, 1998.

[60] D.J. Newman A. Asuncion. UCI machine learning repository, 2007.

[61] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[62] John Koza. Genetic programming, 2007.

[63] Kumar Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, September 1997.

[64] Sean Luke and Lee Spector. A comparison of crossover and mutation in genetic programming. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 240–248, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[65] Gearoid Murphy and Conor Ryan. A simple powerful constraint for genetic programming. *Genetic Programming, Lecture Notes in Computer Science*, 4971, 2008.

[66] Stewart W. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995. http://prediction-dynamics.com/.

[67] Sara Silva and Jonas Almeida. Dynamic maximum tree depth. In E. Cant -Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1776–1787, Chicago, 12-16 July 2003. Springer-Verlag.

[68] Thomas Loveard and Victor Ciesielski. Representing classification problems in genetic programming. In *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1070–1077. IEEE Press, 27-30 May 2001.

[69] Been-Chian Chien, Jui-Hsiang Yang, and Wen-Yang Lin. Generating effective classifiers with supervised learning of genetic programming. In *DaWaK*, pages 192–201, 2003.

[70] Gianluigi Folino, Clara Pizzuti, and G. Spezzano. Improving cooperative gp ensemble with clustering and pruning for pattern classification. In *GECCO 2006*, pages 791–798, New York, NY, USA, 2006. ACM.

[71] Wei-yin Loh Tjen-sien Lim. An empirical comparison of decision trees and other classification methods. Technical report, 1997.

[72] Luc Vincent and Pierre Soille. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Annalysis and Machine Intelligence*, 13(6):583–598, 2006.

[73] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd Edition*. Boston: Addison-Wesley, 1993.

[74] Fuhui Long Hanchuan Peng and Chris Ding. Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy. 27(8):1226–1238, 2005.

[75] Ron Kohavi and George John. Wrappers for feature subset selection. 97(1):273–324, 1997.

[76] Marc Ebner. Evolving color constancy for an artificial retina. In *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 11–22, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.

[77] E.H. Land. Recent advances in retinex theory and some implications for cortical applications: Colour vision and the natural image. In *Proceedings of the National Academy of Science*, volume 80, pages 5163–5169, 1984.

[78] F. C. Crow. Summed-area tables for texture mapping. 18(3):207–212, July 1984.

[79] M. Roberts. The effectiveness of cost-based subtree caching mechanisms in typed genetic programming for image segmentation. In *Applications of Evolutionary Computation, Proceedings of EvoIASP 2003, Columne 2611 of LNCS*, pages 444–454, 7-10 2003.