Unsupervised Problem Decomposition Using Genetic Programming

Ahmed Kattan, Alexandros Agapitos, and Riccardo Poli

School of Computer Science and Electronic Engineering
University of Essex, Colchester CO4 3SQ, UK
akatta@essex.ac.uk, aagapi@essex.ac.uk, rpoli@essex.ac.uk

Abstract. We propose a new framework based on Genetic Programming (GP) to automatically decompose problems into smaller and simpler tasks. The framework uses GP at two levels. At the top level GP evolves ways of splitting the fitness cases into subsets. At the lower level GP evolves programs that solve the fitness cases in each subset. The top level GP programs include two components. Each component receives a training case as the input. The components' outputs act as coordinates to project training examples onto a 2-D Euclidean space. When an individual is evaluated, K-means clustering is applied to group the fitness cases of the problem. The number of clusters is decided based on the density of the projected samples. Each cluster then invokes an independent GP run to solve its member fitness cases. The fitness of the lower level GP individuals is evaluated as usual. The fitness of the high-level GP individuals is a combination of the fitness of the best evolved programs in each of the lower level GP runs. The proposed framework has been tested on several symbolic regression problems and has been seen to significantly outperforming standard GP systems.

1 Introduction

Problem decomposition aims to simplify complex real world problems in order to better cope with them. This strategy is regularly used by humans when solving problems. For example, computer programmers often organise their code into functions and classes.

Problem decomposition is important for two reasons. Firstly, it reduces the complexity of a problem and, therefore, makes the problem easier to solve by standard machine learning techniques. Secondly, automated problem decomposition may help researchers to better understand a problem domain by discovering regularities in the problem space. One way to formalise the decomposition process is to assume there exist different patterns in the problem space, each pattern has particular characteristics and therefore it needs a special solution.

Generally, problem decomposition allows a better understanding and control of the problem's complexity. However, while it is not difficult to split a problem into several sub-problems to be solved in cooperation with different methods, using the wrong decomposition may actually increase the problems complexity.

An ideal problem decomposition system would be one that gets the data from the user and identifies different groups in the data; each of these groups should be simpler to solve than the original problem. An intelligent decomposition of problems requires understanding the problem domain and usually can only be carried out by experts. In this paper, we propose a GP system that can evolve programs that automatically decompose a problem into a collection of simpler and smaller sub-problems while simultaneously solving the sub-problems. This is an area of GP that has not been thoroughly explored thus far.

The structure of the paper is as follows. In the next section we briefly review previous work on problem decomposition. Section 3 provides a detailed description of our proposed framework. This is followed by details on our experimental setting and results in Sections 4 and 5, respectively. Finally, conclusive remarks are given in Section 6.

2 Related Work

The solution to complex problems typically requires the construction of highly complex systems. These systems typically use hierarchical, modular structures to manage and organise their complexity. Modular structures are widespread in engineering and nature. So, it is reasonable to expect that they could be valuable in GP as well. In particular, modularity and hierarchy can be essential tools for problem decomposition. Consequently, starting from Koza's automatically defined functions (ADFs) [1], they have been a subject of substantial empirical exploration from the early days of GP (e.g., see [2,3,4,5,6,7,8]). Due to space limitations, in this section we will review problem decomposition approaches that are based on the notion of dividing up the test cases into (possibly overlapping) subsets, since these are directly relevant to the work reported in this paper.

Rosca *et al.* [9] proposed a system called Evolutionary Speciation Genetic Programming (ESGP) to automatically discover natural decompositions of problems. Each individual consisted of two parts: *condition* and *output*. The *condition* element represents a Boolean function that receives a fitness case presented as an argument and returns feedback on whether the individual chooses to specialise in that case. The *output* element is a standard GP tree, which receives the chosen fitness cases as input. Naturally, some of the fitness cases may be claimed by more than one individual while others are never chosen. Thus, a fitness function was proposed which encourages individuals to fully cover the problem space and minimise the overlap of the claimed fitness cases. The approach was tested with symbolic regression problem and compared with standard GP and with GP(IF), which additionally includes if-then-else in the function set. GP(IF) is selected as it may implicitly split the problem space into different regions. Indeed, experimentation revealed that GP(IF) evolved conditions in such a way as to effectively assign different fitness cases to different pieces of code and, moreover, that GP(IF) outperformed ESGP.

Iba [10] proposed to extend GP using two well-known resampling techniques known as Bagging and Boosting and presented two systems referred to as BagGP and BoostGP. In these systems the whole population is divided into subpopulations. Each subpopulation is evolved independently using a fitness function based on a subset of the fitness cases, which are allocated by the two resampling techniques, i.e., Bagging and Boosting. Later, the best individual from each subpopulation is selected to form a voting scheme to classify unseen data. In both BagGP and BoostGP the number of subpopulations is determined by the user. Experiments on three benchmark problems showed that BagGP and BoostGP outperformed conventional GP. However, when BagGP and BoostGP were applied to a complex real world problem– the prediction of the Japanese stock market– they performed almost identically to standard GP.

More recently, Jackson [11] proposed a hierarchical architecture for GP for problem decomposition based on partitioning the input test cases into subsets. The approach requires a manual partitioning of the test cases. Then, each subset is independently processed in separate evolved branches rooted at a selection node. This node decides which branch to activate based on the given input case. The branches are evolved in isolation and do not interact with each other. The number of branches is determined by the number of subsets into which the test cases have been divided. The proposed architecture has been tested with the 4-, 5- and 10-even-parity problems and polynomial symbolic regression problems with different numbers of branches. In addition, comparisons with standard GP and GP with ADFs have been performed. Experiments showed that this architecture has outperformed conventional GP systems. Its main disadvantage is that the user is required to manually decompose the test cases.

As one can see, none of the previous methods for problem decomposition via test case subdivision is fully automated. Overcoming this limitation is one of the aims of the work presented in this paper.

3 The Approach

Our problem decomposition system works in two main stages: *i) Training*, where the system learns to divide the training cases into different groups based on their similarity and *ii) Testing*, where the system applies what it has learnt to solve unseen data. The training phase is divided into two main steps *i) resampling*, where the system tries to discover the best decomposition for the problem space and *ii) solving*, where the system tries to solve the problem by solving the sub-problems discovered in the resampling stage independently.

In the resampling stage, the system starts by randomly initialising a population of individuals using the ramped half-and-half method (e.g., see [12]). Each individual is composed of two trees: *projector X* and *projector Y*. Each tree receives a fitness case as input and returns a single value as output. The two outputs together are treated as coordinates for a fitness case in a 2-D plane. The process of mapping fitness cases to 2-D points is repeated for all the training examples. For this task, GP has been supplied with a language which allows the discovery of different patterns in the training set. Table 1 reports the primitive set of the system.

Once the training cases are projected via the two components (*projector X* and *Y*), K-means clustering is applied in order to group similar instances in different clusters. Each cluster then invokes an independent GP run to solve cases within its members. Thus, each cluster is treated as an independent problem. The next subsection will describe the clustering process in detail.

3.1 Clustering the Training Examples

We used a standard pattern classification approach on the outputs produced by the two projection trees to discover regularities in the training data. In principle, any classification

Table 1. Primitives set

Function	Arity	Input	Output
+/, -, /, *, pow	2	Real Number	Real Number
Sin, Cos, Sqrt, log	1	Real Number	Real Number
Constants 1-6	0	N/A	Real Number
X	0	N/A	Real Number

method can be used with our approach. Here, we decided to use K-means clustering (e.g., see [13]) to organise the training data (as re-represented by their two projection trees) into groups. With this algorithm, objects within a cluster are similar to each other but dissimilar from objects in other clusters. The advantage of this approach is that the experimenter doesn't need to split the training set manually. Also, the approach does not impose any significant constrains on the shape or size of the clusters. Once the training set is clustered, we can use the clusters found by K-means to perform classification of unseen data by simply assigning a new data point to the cluster whose centroid is closest to it.

K-means is a partitioning algorithm that normally requires the user to fix the number of clusters to be formed. However, in our case the optimal number of subdivisions for the problem into sub-problems is unknown. Hence, we use a simple technique to find the optimal number of classes in the projected space based on the density of the samples. Once the system groups the projected samples into classes, it invokes an independent GP search for each cluster.

Since K-means is a very fast algorithm, to find the optimal number of clusters the system repeatedly instructs K-means to divide the data set into k clusters, where $k = 2, 3, ...K_{max}$ (K_{max} = 10, in our implementation). After each call the system computes the clusters' quality. The value of k which provided the best quality clusters is then used to split the training set and invoke GP runs on the corresponding clusters.

The quality of the clusters is calculated by measuring cluster *separation* and *representativeness*. Ideal clusters are those that are separated from each other and densely grouped near their centroids.

A modified Davis Bouldin Index (DBI) [14] was used to measure cluster separation. DBI is a measure of the nearness of the clusters' members to their centroids, divided by the distance between clusters' centroids. Thus, a small DBI index indicates well separated and grouped clusters. Therefore, we favour clusters with a low DBI value.

DBI can be expressed as follows. Let C_i be the centroid of the i^{th} cluster and d_i^n the n^{th} data member of the i^{th} cluster. In addition, let the Euclidean distance between d_i^n and C_i be expressed by the function dis(d_i^n , C_i). Furthermore, let again k be the total number of clusters. Finally, let the standard deviation be denoted as *std()*. Then,

$$DBI = \frac{\sum_{i=0}^{k} std[dis(d_i^0, C_i), ..., dis(d_i^n, C_i)]}{dis(C_0, C_1, ..., C_k)}$$

The representativeness of clusters is simply evaluated by verifying whether the formed clusters are representative enough to classify unseen data. In certain conditions, the projection trees may project the data in such a way that it is unlikely to be suitable to classify unseen data. For example, clusters that have few members are unlikely to be representative of unseen data. To avoid pathologies of this kind, the system verifies whether the formed clusters have a sufficiently large number of members. In particular, it penalises the values of k that lead K-mean to form clusters where less than a minimum number of members is present. In this work, the minimum allowed number of members for each cluster was simply set to 10 samples. However, we have not thoroughly investigated whether this was optimal in all conditions. For example, it is likely that the optimum minimum size of the clusters is modulated by the total number of training examples available.

More formally, the quality, Q_k, of the clusters obtained when K-means is required to produce k clusters can be expressed as the follows. Let θ_k be the penalty value applied to the quality if there is a problem with the representativeness of the clusters produced. If any particular cluster has less than a minimum number of members we set $\theta_k = 1000$, while $\theta_k = 0$ if no problem is found. Furthermore, let DBI_k represent the corresponding cluster separation. Then,

$$Q_k = DBI_k + \theta_k$$

After running K-means for all values of k in the range 2 to K_{max}, we choose the optimal k as follows:

$$k_{best} = \arg \min_{2<k<K_{max}} Q_k$$

The main factor that affects the optimal number of clusters is the density of the projected samples. The method described above effectively analyses the density of the data from this point of view. Algorithm 1, describes the clustering process in details.

A disadvantage of this approach is that the K-means algorithm has to be executed several times per fitness evaluation, which slows down the evolution a little. However, this only needs to be done during evolution. During normal operation we simply apply the previously formed clusters (represented by their centroids) to the unseen data.

As mentioned previously, once the system has identified the optimal k value and the corresponding clusters, it invokes an independent GP search for each cluster. The purpose is to evolve a program that satisfies the fitness cases in the cluster. In the testing phase, unseen data go through the two projector components of the evolved solution and are projected onto a two-dimensional Euclidean space. Then, they are classified based on the closest centroid. Finally, the input data are passed to the evolved program associated to the corresponding cluster.

The advantage of this approach is that it greatly simplifies classification. This is because evolution pushes projection trees to represent the data in such a way as to optimise the performance of the classification algorithm. Here, we used K-means for its simplicity of implementation and its execution speed, but other techniques might work equally well.

3.2 Search Operators

We used tournament selection and the standard genetic operators: sub-tree crossover, sub-tree mutation and reproduction. Naturally, in the top-level GP runs, the genetic operators have to take the multi-tree representation of individuals into account.

```
Project(n, treeX, treeY);
List Qk;
for int k=2; k ≤ KMAX; k++ do
    //call the K-means algorithm
    K-means(k, n);
    int separation = calculate_DBI();
    if check_clusters_representativeness() == true then
        theta = 0
    else
        theta = 1000
    end
    Qk.append(separation + theta, k)
end
//find the best number of clusters
int number_of_clusters = Qk.get_min_k();
```

Algorithm 1. Finding the optimal number of clusters in the projected space

There are several options for applying genetic operators to a multi-tree representation: apply an operator to all trees within an individual, use different operators for different trees, constrain crossover to happen only between trees at the same position in the parents, allow crossover between different trees within the representation, and so on.

It is unclear what technique is best (e.g., see [15] and [16]). So, in preliminary experiments we tried a variety of approaches and found that a good way to guide evolution in our system is to allow crossover to freely pick feature-extractions trees. In other words, the *projector X* tree of one parent can be crossed over with either the *projector X* tree or *projector Y* of the other parent and *vice versa*.

3.3 Fitness Evaluation

We evaluate the top-level GP system's individuals (represented by two projection trees) by measuring how well the whole problem is solved. The clusters formed by K-means represent subsets of training examples. Each cluster invokes a GP search to solve its member's cases. We call this *inner GP search*. For simplicity, each inner GP runs for a small fixed number of generations with a fixed population size. In future research we will study the benefits and drawbacks of letting the system decide the settings of each inner GP run (e.g., based on the size of the associated cluster).

In our system all inner GP systems evolve simultaneously. The fitness of a top-level GP individual depends on the fitness of the best evolved individual in each of the inner GP runs. If k_{best} is the number of clusters found on the projected space using Algorithm 1 and f_i is the fitness of the best evolved program in the i^{th} inner GP run, then the fitness of top-level individuals is:

$$f = \frac{1}{k_{best}} \sum_{i=1}^{k_{best}} f_i.$$

This fitness function encourages the individuals to project the fitness cases in such a way that a solution for the fitness cases in each group can easily be found by the inner GP runs.

4 Experimental Setup

Experiments have been conducted to evaluate the proposed framework. To do this we chose a variety of symbolic regression problems, which we felt were difficult enough to demonstrate the characteristics and benefits of the method.

We used discontinuous functions as symbolic regression target functions. These allow us to study the ability of the system to decompose a complex problem into simpler tasks. Table 2 list the functions as well as the ranges from which we drew samples to create symbolic regression test problems. Function 1 was used in Rosca's experiments in [9] to evaluate his proposed system. Here, we used the same function to ease the comparison against Rosca's system.

In order to evaluate our results, a comparison has been conducted against both canonical GP and GP(IF), where we added an IF-THEN-ELSE primitive to the function set. This primitive has four types of conditions, namely, $<$, $>$, $>=$ and $<=$. The function

Table 2. Test functions

function	*Notation*	*Training interval*
1	$f1(x) = \begin{cases} e^{x+5}, & x < 0 \\ 1 - \log(x^2 + x + 1), & x \geq 0 \end{cases}$	[-2,2]
2	$f2(x) = \begin{cases} x + 6 \quad , & x < 0 \\ \dfrac{3}{x^2 + 1}, & x \geq 0 \end{cases}$	[-2,2]
3	$f3(x) = \begin{cases} x + \sin(x - 1), & x < 0 \\ 6 * \sin(x) * \cos(x), & 0 \leq x < 1 \\ \sqrt{x}, & x \geq 1 \end{cases}$	[-2,2]
4	$f4(x) = \begin{cases} \dfrac{30\,x}{(x-2) * x}, & x \leq -1 \\ \dfrac{x^4 - x^3 + x^2}{x + 2}, & -1 < x < 0 \\ \dfrac{x}{5}, & 0 \leq x < 1 \\ x^3 * e^x * \cos(x), & x \geq 1 \end{cases}$	[-2,2]
5	$f5(x) = \begin{cases} \dfrac{8}{2 + x^2 + 2x^4}, & x < -2 \\ \dfrac{x^3}{5}, & -2 \leq x < 0 \\ \dfrac{x}{x^2}, & x \geq 0 \end{cases}$	[-4,2]

receives four arguments: the first two are passed to the condition, while the other two represent code to be executed if the condition is true or false, respectively. GP(IF) was selected because, as seen in Rosca's work, it may implicitly split the problem space into different pieces of code and it is, therefore, likely to be competitive for symbolic regression with discontinuous functions.

Our experiments were conducted using the parameter settings in Table 3. The primitive set for both Standard GP and GP(IF) was the same as for our GP system (see Table 1).

Performance has been measured through 100 independent runs for each system (20 runs for each test function). For the training, 100 samples were uniformly selected from the training interval (see Table 2). Evolved solutions were then evaluated using 400 different samples. Each evolved solution has been evaluated with two different test sets. Firstly, we tested performance of the solutions within the training interval (interpolation). Secondly, we evaluated performance on a bigger interval (extrapolation). The extrapolation interval for all test functions was the interval $[-5,5]$, except for function 5 where we used the interval $[-7,7]$.

The fitness measurement for GP, GP(IF) and the inner GP runs in our system is the mean absolute error over all training samples. For the top-level GP runs in our system, however, the fitness evaluation described in Section 3.3 has been applied, which averages over the contribution of each cluster in solving the overall problem.

Table 3. Parameters setting

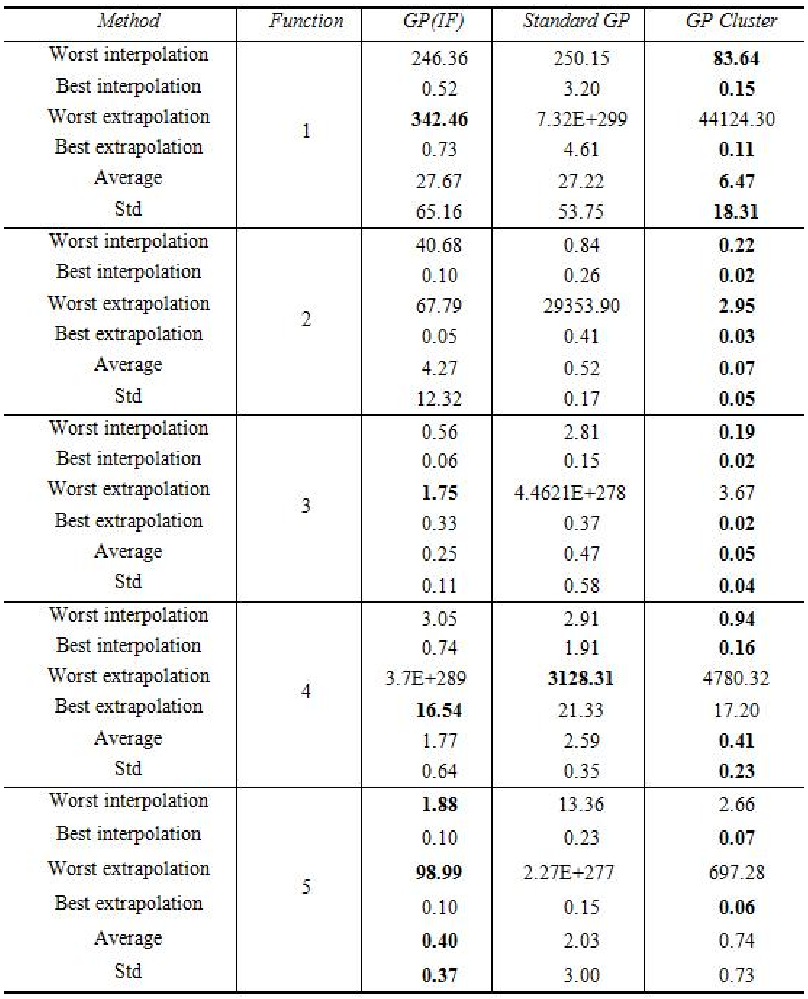
Method	GP Cluster	GP Cluster (inner GP)	GP(IF)	Standard GP
Generations	10	30	30	30
Population	10	100	1000	1000
Crossover	90%	90%	90%	90%
Mutation	5%	5%	5%	5%
Reproduction	5%	5%	5%	5%
Tournament size	2	10	10	10

5 Experimental Results

Table 4 reports the results of the experiments for all five test functions and for standard GP, GP(IF) and our system (GP Cluster). In addition, the average error obtained across 20 independent runs is reported in Table 2, in order to provide information on the stability of each system. Test functions report the best and worst interpolation and extrapolation achieved by each system in all runs and the standard deviation for all runs.

It is clear that our approach has outperformed standard GP by a significant margin in all test functions. It also outperformed GP(IF) in four out of five problems. Furthermore, in all cases standard deviations for Cluster GP were very small, indicating the reliability of the approach. This is the result of the system splitting the relatively complex shape of these discontinuous functions into simpler fragments (i.e., sub-problems). Looking at the number subsets used throughout the test runs, we see that in functions 1, 2 and 5, the system decided to split the training samples into 4 to 10 clusters. In function 3,

Table 4. Experimental results. Statistics are based on 20 independent runs for each function.



Method	*Function*	*GP(IF)*	*Standard GP*	*GP Cluster*
Worst interpolation	1	246.36	250.15	**83.64**
Best interpolation		0.52	3.20	**0.15**
Worst extrapolation		**342.46**	7.32E+299	44124.30
Best extrapolation		0.73	4.61	**0.11**
Average		27.67	27.22	**6.47**
Std		65.16	53.75	**18.31**
Worst interpolation	2	40.68	0.84	**0.22**
Best interpolation		0.10	0.26	**0.02**
Worst extrapolation		67.79	29353.90	**2.95**
Best extrapolation		0.05	0.41	**0.03**
Average		4.27	0.52	**0.07**
Std		12.32	0.17	**0.05**
Worst interpolation	3	0.56	2.81	**0.19**
Best interpolation		0.06	0.15	**0.02**
Worst extrapolation		**1.75**	4.4621E+278	3.67
Best extrapolation		0.33	0.37	**0.02**
Average		0.25	0.47	**0.05**
Std		0.11	0.58	**0.04**
Worst interpolation	4	3.05	2.91	**0.94**
Best interpolation		0.74	1.91	**0.16**
Worst extrapolation		3.7E+289	**3128.31**	4780.32
Best extrapolation		**16.54**	21.33	17.20
Average		1.77	2.59	**0.41**
Std		0.64	0.35	**0.23**
Worst interpolation	5	**1.88**	13.36	2.66
Best interpolation		0.10	0.23	**0.07**
Worst extrapolation		**98.99**	2.27E+277	697.28
Best extrapolation		0.10	0.15	**0.06**
Average		**0.40**	2.03	0.74
Std		**0.37**	3.00	0.73

*Numbers in **bold** represent the best achieved result.

the system identified 6 to 10 clusters in the problem space and in function 4, the most complex in our test set, it identified 8 to 10 clusters.

As we mentioned before, function 1 has been used in Rosca's experiments in [9]. The best achieved accuracy reported on the interval [-2,2] was 1.5, while in our system we have a best interpolation error of 0.15.

We summarise the results from Table 4 in Table 5. As one can see our approach comes on the top of the comparison. Moreover, our results also show that GP(IF) is a marginal second, while standard GP comes last.

Table 5. Experimental results summary

Method	GP(IF)	Standard GP	GP Cluster
Worst interpolation Avg.	75.14	54.01	**17.53**
Best interpolation Avg.	0.26	1.15	**0.08**
Worst extrapolation Avg.	1.1912E+289	1.465E+299	**9921.70**
Best extrapolation Avg.	5.20	5.37	**3.48**
Average of Averages	8.32	6.57	**1.55**
Std Avg.	19.78	11.57	**3.87**

Table 6. A Kolmogorov-Smirnov test

Function	**Method**	**Standard GP**	**GP Cluster**
1	GP(IF)	0.001 / 0	0 / 0.275
	Standard GP	N/A	0 / 0
2	GP(IF)	0 / 0.023	0 / 0.771
	Standard GP	N/A	0 / 0.001
3	GP(IF)	0.135 / 0.008	0 / 0.003
	Standard GP	N/A	0 / 0.003
4	GP(IF)	0.135/ 0.008	0 / 0.275
	Standard GP	N/A	0/ 0.135
5	GP(IF)	0 / 0965	0.059 / 0.275
	Standard GP	N/A	0.023 / 0.135

*The results in the table is the *P* value for Interpolation / Extrapolation

In order to evaluate the statistical significance of our results, a Kolmogorov-Smirnov two-sample test [17] has been performed on the test-case results produced by the best evolved system in each run for all pairs of systems under test and for all five test functions. The test has been repeated for both interpolation and extrapolation. Table 6 reports the *P* value for the tests. As one can see in 9 out of 10 interpolation cases our system is statistically significantly superior to both standard GP and GP(IF) at the standard 5% significance level. The superior performance of GP(IF) on function 5 observed in Table 4 is not statistically significant (albeit by a very small margin). In the extrapolation results (which are, rather obviously, affected by a much larger variance) our system is statistically significantly superior to the others in 4 out of 10 cases, although as one can infer from Table 5, one might expect that performing more runs would eventually statistically confirm the superiority of our system in more cases.

6 Conclusions

In this paper we presented a new framework to automatically decompose difficult symbolic regression tasks into smaller and simpler tasks. The proposed approach is based on the idea of first projecting the training cases onto a two-dimensional Euclidian space via two evolved projection programs, and then clustering them via the K-means algorithm to better see their similarities and differences. The clustering is guaranteed to be optimal thanks to the use of an iterative process. This process uses a quality measure based on the density of the projected samples. Once the data are clustered, they are passed to separate GP runs which evolve specialised solutions for them. Note that while the projection and clustering steps may seem excessive for scalar domains, they make our problem decomposition technique applicable to much more complex domains.

Experiments have been conducted with symbolic regression problems using five different discontinuous functions as target functions. The proposed approach has outperformed conventional GP systems significantly. Also, experiments showed a remarkable stability for our system across runs.

The main motivation behind this research was to produce an intelligent system that is able to solve complex problems by automatically decomposing the problem space into different classes and thereafter solve each class separately in order to solve the whole problem in cooperation. We feel that we have achieved our aim within the specific domain of input space decomposition, as shown by our experimentation. Of course, there are many other ways of performing problem decomposition and modularisation as mentioned in Section 2. We hope to be able to extend our clustering idea to other forms of decomposition.

This research can be extended in many different ways. In the future we will extend the experimentation by testing the technique on multi-varied problems and non-symbolic-regression problems. In addition, we will investigate the benefits and drawbacks of alternative fitness functions (particularly for the top-level GP system). For example, the fitness function might take the size of the identified clusters into consideration. Moreover, the system should be able to change the settings of each inner GP run according to the difficulty of the given sub-problem. Further, we intend to investigate the relationship between the identified number of clusters and how this affects the solutions' accuracy.

References

1. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge (May 1994)
2. Angeline, P.J., Pollack, J.B.: The evolutionary induction of subroutines. In: Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society, Bloomington, Indiana, USA, pp. 236–241. Lawrence Erlbaum, Mahwah (1992)
3. Spector, L.: Evolving control structures with automatically defined macros. In: Siegel, E.V., Koza, J.R. (eds.) Working Notes for the AAAI Symposium on Genetic Programming, November 10–12, pp. 99–105. MIT/AAAI, Cambridge (1995)
4. Rosca, J.P., Ballard, D.H.: Discovery of subroutines in genetic programming. In: Angeline, P.J., Kinnear Jr., K.E. (eds.) Advances in Genetic Programming 2, ch. 9, pp. 177–202. MIT Press, Cambridge (1996)
5. Seront, G.: External concepts reuse in genetic programming. In: Siegel, E.V., Koza, J.R. (eds.) Working Notes for the AAAI Symposium on Genetic Programming, November 10–12, pp. 94–98. MIT/AAAI, Cambridge (1995)
6. Jonyer, I., Himes, A.: Improving modularity in genetic programming using graph-based data mining. In: Sutcliffe, G.C.J., Goebel, R.G. (eds.) Proceedings of the Nineteenth International Florida Artificial Intelligence Research Society Conference, Melbourne Beach, Florida, USA, May 11-13, pp. 556–561. American Association for Artificial Intelligence (2006)

7. Hemberg, E., Gilligan, C., O'Neill, M., Brabazon, A.: A grammatical genetic programming approach to modularity in genetic algorithms. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I., et al. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 1–11. Springer, Heidelberg (2007)
8. McPhee, N.F., Crane, E.F., Lahr, S.E., Poli, R.: Developmental plasticity in linear genetic programming. In: Raidl, G., et al. (eds.) GECCO 2009: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, Montreal, July 8-12, pp. 1019–1026. ACM, New York (2009)
9. Rosca, J., Johnson, M.P., Maes, P.: Evolutionary Speciation for Problem Decomposition (1996) (Available via Citeseer)
10. Iba, H.: Bagging, boosting, and bloating in genetic programming. In: Proceedings of the Genetic and Evolutionary Computation Conference, Orlando, Florida, USA, July 13-17, vol. 2, pp. 1053–1060. Morgan Kaufmann, San Francisco (1999)
11. Jackson, D.: The performance of a selection architecture for genetic programming. In: O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A.I., De Falco, I., Della Cioppa, A., Tarantino, E. (eds.) EuroGP 2008. LNCS, vol. 4971, pp. 170–181. Springer, Heidelberg (2008)
12. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming (With contributions by J. R. Koza) (2008), `http://lulu.com`
13. Han, J., Kamber, M.: Data mining: concepts and techniques. Morgan Kaufmann, San Francisco (2006)
14. Sepulveda, F., Meckes, M., Conway, B.A.: Cluster separation index suggests usefulness of non-motor EEG channels in detecting wrist movement direction intention. In: Proceedings of the 2004 IEEE Conference on Cybernetics and Intelligent Systems, Singapore, pp. 943–947. IEEE Press, Los Alamitos (2004)
15. Muni, D.P., Pal, N.R., Das, J.: A novel approach to design classifier using genetic programming. IEEE Transactions on Evolutionary Computation 8(2), 183–196 (2004)
16. Boric, N., Estevez, P.A.: Genetic programming-based clustering using an information theoretic fitness measure. In: Srinivasan, D., Wang, L. (eds.) 2007 IEEE Congress on Evolutionary Computation, Singapore, September 25-28. IEEE Computational Intelligence Society, pp. 31–38. IEEE Press, Los Alamitos (2007)
17. Peacock, J.A.: Two-dimensional goodness-of-fit testing in astronomy. Royal Astronomical Society, Monthly Notices 202, 615–627 (1983)