Automated Problem Decomposition for the Boolean Domain with Genetic Programming

Fernando E. B. Otero and Colin G. Johnson

School of Computing, University of Kent, Canterbury, UK
{F.E.B.Otero,C.G.Johnson}@kent.ac.uk

Abstract. Researchers have been interested in exploring the regularities and modularity of the problem space in genetic programming (GP) with the aim of decomposing the original problem into several smaller subproblems. The main motivation is to allow GP to deal with more complex problems. Most previous works on modularity in GP emphasise the structure of modules used to encapsulate code and/or promote code reuse, instead of in the decomposition of the original problem. In this paper we propose a problem decomposition strategy that allows the use of a GP search to find solutions for subproblems and combine the individual solutions into the complete solution to the problem.

1 Introduction

Many problems in the genetic programming (GP) literature have demonstrated the scalability issues of GP algorithms—e.g., it is relatively easy to find a solution for the even-4-parity problem [1], while a solution for the even-8-parity problem is much harder to find using a traditional GP. In order to be able to deal with larger and more complex problems, researchers have been interested in exploring the regularities and modularity of the problem space with the aim of decomposing the original problem into several smaller (more tractable) subproblems.

One of the first approaches for exploiting the problem regularities is Koza's Automatic Defined Functions (ADFs) [1, 2]. In ADFs, the structure of program trees is defined in a way that subtrees with different roles are evolved in parallel— e.g., there are *function-defining* subtrees and a *result-producing* subtree, which can contain references to the different function-defining subtrees mimicking function calls. Other authors investigated the creation of modules (functions) by identifying subtrees on existing individuals [3–6]. The main idea is to create modules based on fit or useful subtrees, either encapsulating their functionality or creating parameterised modules.

In this paper, we investigate the use of an automated problem decomposition strategy in the context of GP. The motivation is to use a heuristic to modularise the GP search—i.e., use a GP search to explicitly find solutions to subproblems, which can then be combined to create the solution to the original problem. Therefore, the GP is not concerned with searching for the complete solution; the original problem is decomposed in a series of smaller subproblems.

The remainder of this paper is organised as follows. Section 2 reviews prior efforts to explore the regularities and modularity of the problem space in GP. Section 3 discusses the proposed strategy to modularise the GP search and Section 4 gives details of a specific implementation of this strategy. The computational results are presented in Section 5. Finally, Section 6 concludes this paper and presents future research directions.

2　Background

Automatically Defined Functions (ADFs) were introduced by Koza [1, 2] as a technique to explore the regularities and modularities of the problem space in order to deal with complex problems, and it is probably the most popular and studied automatic approach to create modules (sub-routines) in GP. Koza proposed the use of ADFs to decompose the problem into several smaller sub-problems. The solution of the original (complete) problem is then obtained by combining the individual solutions to the subproblems. This process, defined by Koza as *hierarchical problem-solving process*, is illustrated in Figure 1. There are three important steps in this process: the first one is where the original problem is decomposed, the second is where the solutions of each subproblem are obtained, and the third one is where the complete solution is built by combining the individual solutions of the subproblems. Koza's ADF approach implements these three steps within a run of a GP algorithm: a modular ADF architecture based on 'function-defining branches' is determined prior to evolving the solutions (decomposition of the problem); the body of each ADF is evolved during the run (subproblem solution search); these ADFs are available to the 'result-producing branch' of candidate solutions (combination of subsolutions), which is also being evolved during the run.

While Koza's ADF approach allows the GP to exploit the problem regularities through a modular architecture, the problem decomposition into an ADF architecture (i.e., number of ADFs, the number of arguments of each ADF) is done manually before the run of the GP. This also includes the definition of the interaction between ADFs—which ADFs are allowed to call which other ADFs. Therefore, the architecture of the candidate solutions is fixed to a pre-defined number of ADFs and the 'result-producing branch'. In [7], a set of architecture altering operations relaxed this restriction, allowing candidate solutions to have a different number of ADFs and each ADF to have a different number of parameters, although the maximum number of ADFs and arguments of each ADF are restricted by user-defined values.

Other works have proposed the creation of modules based on the genetic material from individuals of the population. Koza [1] proposed the use of a subtree *encapsulation* operator. The approach consists in randomly selecting a subtree from a fit individual and creating a terminal primitive to reference the subtree—i.e., a terminal that *encapsulates* the behaviour of the subtree. The motivation is to protect the encapsulated subtree from potential changes as a result of genetic operators, and to facilitate its reuse by allowing the mutation
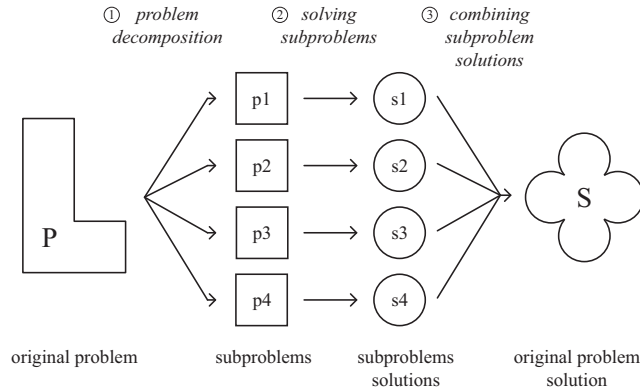
Fig. 1. The hierarchical problem-solving process (adapted from [1]): the original problem P is decomposed in a set of subproblems (step 1); the goal is then to solve each of the subproblems (step 2); finally, the solution S to the original problem P is created by using the solutions to the subproblems (step 3).

operator to incorporate new references in the population. The terminal primitive created by the encapsulation operator can be seen as a module (function) with no arguments. Angeline and Pollack [3, 8] proposed the Genetic Library Builder (GLiB) system, which employs special mutation operators to define new modules based on subtrees from individuals. The first mutation operator is *compression*, which consists of randomly selecting a subtree to define a new module. The newly created module is then stored in a global module library and the occurrence of the subtree is replaced by a reference to the module. The arguments of the module are determined based either on the maximum depth of the module or by the terminal (leaf) nodes used in the subtree. The second mutation operator is *expand*, which consists in expanding the module by replacing its reference with the subtree stored in the module library that defines the module. While module definitions in GLiB are selected at random, Rosca and Ballard [4] proposed a method to create new modules using heuristics to identify 'useful' building blocks: fit blocks (blocks with high fitness value) and frequent blocks (blocks that appear frequently in the entire population). Once a block has been identified and its arguments determined (based on the terminals used in the subtree), its definition is added to the function set as a new function and a replacement operator introduces new individuals using the extended function set.

The idea of identifying building blocks in the population to create a library of modules was extended further to include information accumulated from multiple runs of a GP algorithm. Roberts et al. [5] proposed the use of a *subtree database* to monitor the frequency of use of each subtree during the GP run. At the end of the run, the most frequent subtrees are encapsulated as terminal primitives in a similar manner as the encapsulation operator proposed by Koza [1]. The subsequent runs can then take advantage of the subtree database by using an extended terminal set incorporating the encapsulated subtrees. Keijzer et al.

[6] introduced the idea of Run Transferable Libraries (RTL), in which the GP system uses the RTL in two phases: (1) training of the library, where a number of runs is used to refine the randomly generated candidate modules (referred to as Tag Addressable Functions) of the RTL; (2) subsequent runs of the GP use the modules of the RTL. The motivation is that the RTL can be trained in smaller (simpler) problem instances and then be applied to larger (harder) problem instances. A similar idea was used by Christensen and Oppacher [9], where a 'training phase' consisting in generating all small trees in the search space of GP creates a library of useful modules. Christensen and Oppacher's approach explores the fact that there are more solutions of small size for the Santa Fe Trail problem compared to larger ones. The generated modules (small trees) are then used in the search for the complete solution.

Several other approaches for the creation/identification of modules have been proposed in the literature—e.g., in the context of grammar-based GP (grammatical evolution) [10, 11], in Cartesian Genetic Programming (CGP) [12] and in GP systems using the Push language [13]; other approaches are discussed in [14]. The majority of approaches for modularity (including the ones discussed above) focus on the discovery of modules rather than on the use of modules to decompose the problem into smaller (more tractable) subproblems, relying on the idea that if modules can be created/identified, their usefulness will emerge through the GP search. An indication of this is the fact that modules are usually created/identified during the run of the GP, at the same time that the GP is searching for a solution to the problem. There is no control to check whether different modules are solving different parts of the problem or not, and the quality of the modules is evaluated indirectly by evaluating how well an individual that contains the module reference solves the problem.

This emphasis on the structure of modules to encapsulate code and/or promote code reuse of most previous works in GP modules motivated Jackson and Gibbons [15] to propose the use of *layered learning*—an approach that aims at solving simpler problems in order to deal with harder problems—in the context of GP. The idea is to first use a layer to solve a lower-order version of the original problem. Then, when a solution to the first layer is found, it is converted to a parameterised module. Finally, a second layer is used to search for the solution to the original problem, which can invoke the module created by the first layer. Drawing a comparison with ADFs, the first layer in the layered learning approach can be seen as a function-defining branch and the second layer can be seen as the result-producing branch. The main difference between ADFs and the layered learning approach is that in the latter, all computational effort is first focused in the function-defining branch (layer 1) until it evolved into something potentially useful, and then switched to the result-producing branch (layer 2), while in ADFs both function-defining and result-producing branches are evolved simultaneously. One limitation of the layered learning is the need to manually identify (and specify) a lower-order of the problem to be solved by the first layer. A second limitation is that there is a single decomposition step, and more complex problems may require multiple decomposition steps.

3 Modularisation of the GP search

The problem-solving procedure of GP can be viewed as a supervised learning procedure:

(1) the training data is represented by a set of input-output pairs, which correspond to the desired behaviour;
(2) the fitness function is used to evaluate how good a candidates solutions' predictions are (i.e., how many correct predictions are made or how close the predictions are to the correct output);
(3) the goal of the GP search is to find a program that can predict the correct output for each of the inputs or, in cases where it cannot find the program that generates the correct output, find one that provides the best fitness score given by the fitness function.

Many supervised learning methods employ a strategy to decompose the problem at hand into smaller subproblems. For example, decision tree induction algorithms usually employ a divide-and-conquer strategy to build a decision tree in a top-down fashion. Starting from the root node, a test is selected to divide the training instances—a descendant of the root node is created for each possible outcome of the test and the training instances are sorted to the appropriate descendant node. This procedure is then repeated for each descendant node using the subset of the training instances associated with the node—a test is selected for each of the descendant nodes to further divide the training instances. Another example is the strategy used by rule induction algorithms. Instead of attempting to create a complete list or set of rules at once, they employ a sequential covering strategy to reduce the problem into to a sequence of simpler problems, each consisting in creating a single rule. The sequential covering is an iterative procedure in which a single rule is created and the training instances correctly classified by the rule are removed from the training data, effectively reducing the search space for the next iterations of the procedure.

Most works in GP focuses on searching for a complete solution. While the use of ADFs (or other module/building block creation method) provides a syntactic modularisation, where different subtrees might focus on different parts of the problem, there is still an evolutionary pressure to solve all parts of the problem at once. McKay [16] argues that this pressure tends to reduce diversity and in some cases prevents the search from converging to an optimal solution. To counteract this effect, McKay uses the concept of partial functions—functions whose values are not defined for some inputs—combined with the use of fitness sharing to promote diversity and allow the GP search to explore subproblem solutions. The use of partial functions can be seen as an explicit attempt to modularise the GP search, i.e., focus the search on solutions of subproblems.

The hierarchical problem-solving process presented by Koza as a motivation to use ADFs is closely related to both divide-and-conquer and sequential covering strategies commonly used in machine learning, although ADFs do not use a heuristic to decompose the problem. The use of layered learning by Jackson

and Gibbons [15] can also be seen as a divide-and-conquer, but it involves a single decomposition step represented by the manually identified lower-order version of the original problem. A natural question then arises: *could we apply a heuristic to decompose the problem into smaller problems and use GP to find a solution to subproblems?* Assuming that we successfully decompose the problem and find solutions to the subproblems, we then have a second question: *how do we combine the individual solutions to the subproblems into the complete solution?* In the next section we discuss how we can combine both the sequential covering strategy and the concept of partial functions to modularise the GP search—use GP to solve several smaller subproblems and combine the solutions to create the complete solution to the problem—and address the aforementioned questions.

4   Sequential Covering Genetic Programming

In this section we present the general idea behind the proposed sequential covering GP (SCGP). There are three distinct steps: (i) the decomposition of the problem; (ii) the search for a subproblem solution; and (iii) the combination of subproblem solutions into the complete solution. Figure 2 presents the high-level pseudocode of the SCGP procedure.

The overall SCGP procedure mimics a sequential covering: starting with the complete list of input cases (training data) and an empty *solution tree*,[1] evolves a partial solution using GP, adds the partial solution to the solution tree and removes the cases for which it gives the correct output. The procedure is repeated until there are no input cases remaining. The removal of cases at each iteration effectively changes the search space for the next iterations, which allows the GP to evolve solutions to different parts of the problem—i.e., reduces (decomposes) the problem into a sequence of simpler problems, each consisting in creating a solution for a subset of the input cases.

At each iteration of the SCGP procedure, a partial solution is evolved using a GP.[2] The fitness cases for the GP consists of the available input cases—the ones that have not been correctly predicted previously. The best (fittest) candidate solution evolved by the GP is designated as the partial solution of the iteration. There are two possible outcomes as a result of the GP search: the partial solution produces the correct output for all available input cases (i.e., it is the optimal solution for the subproblem represented by the available input cases), or the partial solution produces the correct output for a subset of the input cases. If the partial solution is the optimal solution for the subproblem, it is added as a leaf component to the solution tree and the SCGP procedure finishes, since the solution tree is able to generate the correct output for all input cases. If the partial solution only solves a subset of the input cases, a *mask selector* is created to combine the newly created partial solution with the remaining solutions of the

---

[1] Here we assume that the solution tree is where all the partial solutions (the solution to individual subproblems) are combined into the complete solution to the problem.

[2] Each iteration of SCGP involves the execution of a GP algorithm, which also evolve for a number of iterations.

```
 1.  training ← all input cases;
 2.  solution ← ∅;
 3.  while |training| not empty do
 4.     partial ← EvolveSolution(training);
 5.     if Errors(partial, training) = 0 then
 6.        solution ← AddLeafComponent(partial, solution);
 7.     else
 8.        mask ← GenerateTestMask(partial, training);
 9.        solution ← AddMaskComponent(partial, mask, solution);
10.     end if
11.     training ← RemoveCorrectCases(solution, training);
12.  end while
13.  solution ← Simplify(solution); /* optional */
14.  return  solution;
```

Fig. 2. High-level pseudocode of the Sequential Covering GP (SCGP).

solution tree. The cases for which the (extended) solution tree gives the correct output are removed and a new iteration of the procedure starts.

So far, we have demonstrated how we can use a heuristic to decompose the problem and use a GP to produce the solutions to the subproblems, which answers our first posed question. The remaining issue is how to combine the solutions to the subproblem into a single solution. We have mentioned that individual solutions are structured in a solution tree and combined together using a mask selector. Given that each partial solution in the solution tree is solving a different subproblem, their output vectors (the vector V of the outputs of the partial solution P_i when queried with the input cases C, i.e., $V(P_i) = \{P_i(c_1), \ldots, P_i(c_N)\}$) are complementary.[3] Therefore, a natural way of combining the partial solutions is to combine their output vectors. To that end, we use the semantic crossover proposed by Moraglio et al. [17] to generate *mask selectors*, which act as tests to inform which of the partial solutions to use for a given input.

The geometric semantic crossover [17] is a semantic operator that works on the output vector of two individuals (candidate solutions). For the Boolean domain, the semantic crossover (SGXB) returns an individual $T_3 = (M \wedge T_1) \vee (\overline{M} \wedge T_2)$, where M is a randomly generated boolean crossover mask. The Boolean expression represented by individual T_3 outputs the value of T_1 or T_2 depending on the value of M—i.e., for each input case c, it outputs the value $T_1(c)$ if $M(c)$ evaluates to `true`; otherwise it outputs the value $T_2(c)$. The construction of the individual T_3 is illustrated in Figure 3. We will focus on the Boolean domain from now on; refer to [17] for details of how to apply the semantic crossover in other domains.

---

[3] There might be overlaps between different vectors, but the important aspect is that for every input case at least one of the vectors provides the correct output.
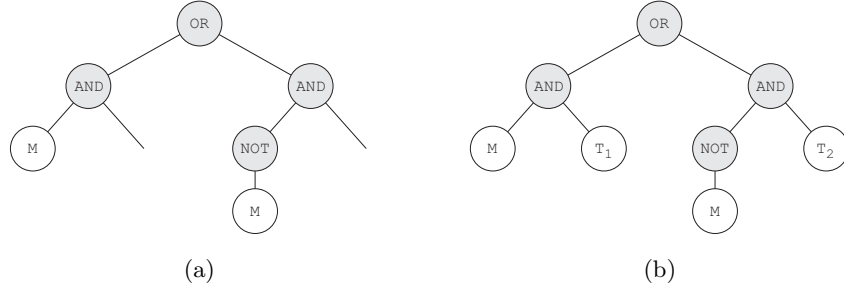
Fig. 3. In (a), the semantic crossover scheme for Boolean functions (M is the randomly generated crossover mask); in (b), the resultant individual T_3 obtained by applying the semantic crossover with individuals T_1 and T_2.

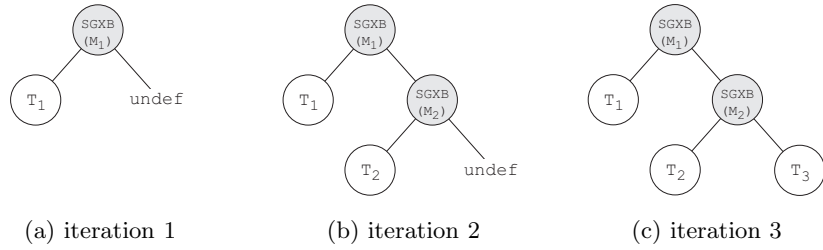(a) iteration 1   (b) iteration 2   (c) iteration 3

Fig. 4. The sequential solution construction procedure of SCGP: in (a) the solution tree after the first iteration, consisting of the *partial* solution T_1 and an incomplete semantic crossover using mask M_1; (b) the solution tree after the second iteration, after the addition of the *partial* solution T_2 and the incomplete semantic crossover using mask M_2; the complete solution tree, obtained by adding the *partial* solution T_3.

Recall that solutions are sequentially discovered by the SCGP procedure, so when a partial solution T_i (the solution created in the i-th iteration) is added to the solution tree, the T_{i+1} solution is unknown. The semantic crossover is usually incomplete, i.e., we do not have two individuals to recombine. To solve this dependency, we use the concept of partial solutions and assume that the solution tree returns an `undef` value for the cases where the mask M_i evaluates to `false`. Therefore, the crossover mask M_i acts as a selector to inform when the output of individual T_i should be used, independently of the other individuals. To ensure this property of the crossover mask, we need to impose a restriction on the creation of the (random) crossover mask M_i: M_i *is a randomly generated boolean crossover mask that, for every input case c, if $M_i(c)$ evaluates to `true`, $T_i(c)$ produces the correct output.*

Let us consider a simple example: assume that we would like to search for a boolean function with the following output `[1, 1, 0, 1, 0, 1]`. The first iteration of SCGP produces an individual T_1 with the output vector `[0, 1, 0, 0, 1, 0]` (an individual that generates the correct output for input cases 2 and 3). If we generate a crossover mask M_1 that returns `true` for input cases 2 and

3 and add both to the the solution tree, we end up with a partial solution with the output vector [undef, 1, 0, undef, undef, undef]. Before we start the next iteration of the SCGP, we remove the input cases for which the solution tree is generating the correct output, so the desired output is [1, -, -, 1, 0, 1] (the positions marked as '-' are not used in the evaluation). This will focus the search on the input cases where the (current) solution tree is not generating the correct output (the input cases for which an undef value is generated). The second iteration of SCGP produces an individual T_2 with the output vector [1, 0, 0, 0, 1, 1]. Applying the same procedure to generate a mask M_2 and adding both T_2 and M_2 to the solution tree, we end up with a partial solution with the output vector [1, 1, 0, undef, undef, 1]. Removing the correct input cases, the desired output is [-, -, -, 1, 0, -]. The next iteration of SCGP produces an individual T_3 with the output vector [1, 1, 1, 1, 0, 0]. Since T_3 generates the correct output for the remaining input cases, we don't need to create a crossover mask. Adding T_3 to the solution tree completes the SCGP procedure (there are no input cases for which the solution tree generates an undef value) and the solution tree represents the Boolean function with the desired output. The sequential solution construction procedure of SCGP is illustrated in Figure 4.

Note that the sequential construction of the solution avoids the problem of exponential growth of the size of GP individuals and the need for a simplification step [17], observed when semantic operators are used (especially the semantic crossover, since both parents are included in the offspring). The sequential procedure of the SCGP decomposes (reduces) the original problem, and each iteration is searching for a solution to a subproblem. The subproblem solutions are not used during the search of the GP, therefore the size of the current solution tree (the solution being sequentially constructed) does not affect the GP search. On the other hand, the complete solution (solution tree at the end of SCGP) can become syntactically large, depending on the number of iterations required to create the optimal solution. For applications where the size of the complete solution is important, a single simplification step can be used at the end of SCGP.

5   Computational Results

In this section we present the results of the proposed SCGP in two Boolean logic problems.[4] We used a standard tree GP to create a solution at each iteration of SCGP, using a generational scheme with tournament selection (size 5), ramped-half-and-half initialisation, subtree crossover (0.9 probability), subtree mutation (0.1 probability) and elitism (1 individual). We varied the GP parameters population size {10, 50, 100, 500, 1000}, maximum number of iterations {1, 10, 50, 100} and the maximum tree depth {2, 4, 8} to determine their effects on the overall performance of the SCGP. Greater values of the population size and the maximum number of iterations only increased the total number of fitness evaluations without any improvements on the overall performance of SCGP. The

---

[4] The SCGP algorithm was implemented using the EpochX framework [18].

only GP parameter that seems to directly affect the performance was the maximum tree depth, where a greater value allows the SCGP algorithm to create a complete solution in a smaller number of sequential covering iterations. The results reported in this section correspond to the runs of SCGP using a GP with a population size of 10, maximum number of iterations of 1 and maximum tree depth of 8—the combination that produced the best average number of fitness evaluations.

The SCGP was compared against a standard tree GP, semantic GP (SGP) and semantic stochastic hill climber (SSHC), using the same setup as in [17]: GP and SGP using a generational scheme with tournament selection (size 5), crossover and mutation; other parameters set to ECJ's defaults [19]. We selected two standard GP Boolean benchmark problems, the even-parity and multiplexer [1]. These problems present scalability issues for standard GP—solutions for lower-order versions are easily found, while solutions for higher-order versions are not found in most cases using standard GP. The function set used for both problems comprised the Boolean operators {AND, OR, NOT}. All algorithms were allocated a maximum of $2n \times 2^n$ fitness evaluations, where n is the number of input variables of the problem.

Discussion: Table 1 presents the average number of SCGP (sequential covering) iterations and fitness evaluations required by SCGP to create the complete optimal solution for each problem, calculated over 30 runs of the algorithm. In all problems, the total number of fitness evaluation required is below the allocated maximum evaluations. The average number of SCGP iterations can be seen as the number of semantic crossover operations required to create the optimal solution. This shows an interesting aspect of SCGP: while the SGP algorithm applies the semantic crossover selecting two individuals at random, the SCGP algorithm applies the semantic operator in a more directed way. It first selects an individual and the crossover mask, and then tries to evolve the best individual that would fit the remaining input cases to complete the crossover. This advantage is highlighted in the results concerning the average number of training examples correctly predicted by the best solution, presented in Table 2. SCGP is the only algorithm to be able to find the optimal solution in all the problems; neither SGP or SSHC, which also use the semantic crossover, found an optimal solution to all the problems.

6 Conclusions and Future Work

We presented a new problem decomposition strategy in the context of GP. This new strategy relies on a sequential covering approach, commonly used in machine learning, to divide the original problem into smaller subproblems. A GP was used to find solutions for the subproblems and the individual subproblems' solutions are combined using a semantic crossover operator. We conducted experiments in two standard GP Boolean benchmark problems, comparing the SCGP (sequential covering GP) against a standard tree GP, semantic GP (SGP) and

Table 1. Average (*average* ± *standard deviation*) number of SCGP iterations and fitness evaluations required by SCGP to create the complete correct (optimal) solution for each problem, calculated over 30 runs. In all problems, the total number of fitness evaluations required is below the allocated maximum (budget) evaluations.

problem	avg. SCGP iterations	avg. evaluations	budget
even-5-parity	23.4 ± 2.0	224.2 ± 20.1	320
even-6-parity	46.7 ± 4.4	457.1 ± 44.6	768
even-7-parity	90.5 ± 3.7	895.0 ± 10.4	1792
even-8-parity	181.7 ± 9.3	1814.1 ± 17.3	4096
even-9-parity	374.5 ± 7.5	3735.0 ± 75.4	9216
even-10-parity	767.6 ± 11.5	7666.6 ± 95.0	20480
multiplexer-6	20.9 ± 5.3	199.0 ± 53.4	768
multiplexer-11	136.1 ± 12.4	1350.7 ± 22.4	45056

Table 2. Average percentage (*average* ± *standard deviation*) of input cases correctly predicted by the best solution for each of the algorithms, calculated over 30 runs.

problem	GP	SGP	SSHC	SCGP
even-5-parity	52.9 ± 2.4	98.1 ± 2.1	99.7 ± 0.9	100.0 ± 0.0
even-6-parity	50.5 ± 0.7	98.8 ± 1.7	99.7 ± 0.6	100.0 ± 0.0
even-7-parity	50.1 ± 0.2	99.5 ± 0.6	99.9 ± 0.2	100.0 ± 0.0
even-8-parity	50.1 ± 0.2	99.7 ± 0.3	100.0 ± 0.0	100.0 ± 0.0
even-9-parity	50.0 ± 0.0	99.5 ± 0.3	100.0 ± 0.0	100.0 ± 0.0
even-10-parity	50.0 ± 0.0	99.4 ± 0.2	100.0 ± 0.0	100.0 ± 0.0
multiplexer-6	70.8 ± 3.3	99.5 ± 0.8	99.8 ± 0.5	100.0 ± 0.0
multiplexer-11	76.4 ± 7.9	99.9 ± 0.1	100.0 ± 0.0	100.0 ± 0.0

semantic stochastic hill climber (SSHC). The proposed SCGP algorithm was the only algorithm to find an optimal solution for all problems within the allocated maximum number of fitness evaluations.

There are several future research directions. The increase in the number of iterations of the GP search did not improve the overall performance of SCGP, which could be an indication that the crossover mask is limiting the use of an individual (one of the individuals in the crossover is only used when the mask evaluates to `true`); it would be interesting to investigate the use of different mask generation procedures. Another approach is to first select the crossover mask, which effectively is responsible to divide the input cases, and then search for each individual to complete the crossover; this would be similar to the top-down approach commonly used by decision tree induction algorithms. Given the nature of the sequential covering solution construction strategy, there is a risk of overfitting the training data. Therefore it will be interesting to investigate how

the solutions found by SCGP generalise to unseen input cases. Additionally, a semantic analysis of the crossover masks, responsible for partitioning the input cases, might give interesting insights about the problems (e.g., characterise different regions of the problem space).

References

1. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
2. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press (1994)
3. Angeline, P.J., Pollack, J.B.: The evolutionary induction of subroutines. In: Proc. of the 14th Annual Conference of the Cognitive Science Society. (1992) 236–241
4. Rosca, J., Ballard, D.: Learning by adapting representations in genetic programming. In: Proc. of the IEEE WCCI. (1994) 407–412
5. Roberts, S., Howard, D., Koza, J.: Evolving Modules in Genetic Programming by Subtree Encapsulation. In: Proc. of EuroGP 2001. LNCS 2038. (2001) 160–175
6. Keijzer, M., Ryan, C., Cattolico, M.: Run Transferable Libraries – Learning Functional Bias in Problem Domains. In: Proc. of GECCO. LNCS 3103. (2004) 531–542
7. Koza, J.R., Bennett III, F.H., Andre, D., Keane, M.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann (1999)
8. Angeline, P.J., Pollack, J.B.: Coevolving High-level Representations. In Langton, C., ed.: Artificial Life III, Addison-Wesley (1994) 55–71
9. Christensen, S., Oppacher, F.: Solving the Artificial Ant on the Santa Fe Trail Problem in 20,696 Fitness Evaluations. In: Proc. of GECCO. (2007) 1574–1579
10. Hemberg, E., Gilligan, C., O'Neill, M., Brabazon, A.: A grammatical genetic programming approach to modularity in genetic algorithms. In: Proc. of EuroGP. LNCS 4445 (2007) 1–11
11. Swafford, J., Hemberg, E., O'Neill, M., Nicolau, M., Brabazon, A.: A Non-Destructive Grammar Modification Approach to Modularity in Grammatical Evolution. In: Proc. GECCO. (2011) 1411–1418
12. Walker, J., Miller, J.: The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. IEEE Transactions on Evolutionary Computation **12**(4) (2008) 397–417
13. Spector, L., Martin, B., Harrington, K., Helmuth, T.: Tag-Based Modules in Genetic Programming. In: Proc. of GECCO. (2011) 1419–1426
14. O'Neill, M., Vanneschi, L., Gustafson, S., Banzhaf, W.: Open issues in genetic programming. Genetic Programming and Evolvable Machines **11** (2010) 339–363
15. Jackson, D., Gibbons, A.: Layered Learning in Boolean GP Problems. In: Proc. of EuroGP. LNCS 4445 (2007) 148–159
16. McKay, R.: Partial Functions in Fitness-Shared Genetic Programming. In: Proc. of CEC. (2000) 349–356
17. Moraglio, A., Krawiec, K., Johnson, C.G.: Geometric Semantic Genetic Programming. In: Proc. of PPSN. LNCS 7491 (2012) 21–31
18. Otero, F., Castle, T., Johnson, C.: EpochX: Genetic Programming in Java with Statistics and Event Monitoring. In: Proc. of GECCO Companion. (2012) 93–100
19. Luke, S.: ECJ: A Java-based Evolutionary Computation Research System. http://cs.gmu.edu/ eclab/projects/ecj/ (2012)