Journal of Computer Science

Original Research Paper

Using Background Knowledge and Random Sampling in Genetic Programming: A Case Study in Learning Boolean Parity Functions

Lappoon R. Tang

Department of Engineering, Roborn Technology Limited 8/F, Core F, Cyberport 3, 100. Cyberport Road, Hong Kong

Article history
Received: 08-02-2023
Revised: 06-06-2023
Accepted: 19-06-2023

Email: rupert.tang@gmail.com

Abstract: The Boolean even-N-parity function returns T (i.e., true) if an even number of its Boolean arguments for N arguments are T and otherwise returns NIL (i.e., false). Learning Boolean even-N-parity functions has been recognized as a difficult problem for evolutionary computation (such as genetic programming) especially when N is large (e.g., 20+). A number of approaches have been proposed for solving the benchmark problem of even-N-parity. Most approaches focus on improving the representation of individuals and/or improving the effectiveness of crossover. So far, no approach has attempted to use high-arity background knowledge/functions for automating problem decomposition in the course of evolution. Our current approach combines the use of high-arity background functions, automatically defined functions, and random sampling of fitness cases to (1) Automate problem decomposition for high-arity even-N-parity problems and (2) Promote diversity in the retainment of genetic materials across generations by using random samples of fitness cases in fitness evaluation. Experimental evaluation shows that such an approach can dramatically reduce the total number of individuals needed to be processed by genetic programming. Therefore, such an approach to genetic programming can significantly improve computational efficiency.

Keywords: Genetic Programming, Symbolic Regression, Even-N-Parity Boolean Functions, Random Sampling, Background Functions

Introduction

Learning Boolean functions from examples began in machine learning (at least) as early as 1943 when an artificial neuron was created to learn Boolean functions such as and, OR (McCulloch and Pitts, 1943). Earlier works on learning Boolean functions include learning of conjunctive normal form from examples (Hirschberg *et al*., 1994). Learning Boolean functions from examples has continued to be an important problem in machine learning (Veness and Hutter, 2014).

The Boolean even-N-parity function returns T (i.e., true) if an even number of its Boolean arguments are T and otherwise returns nil (i.e., false). Learning Boolean even-N-parity functions has been recognized as a difficult problem for evolutionary computation (such as Genetic Programming) especially when N is large (e.g., 20+). When N is equal to 21, there are O (2M) truth table rows to process in learning a target function. If the learning mechanism involved is "vanilla"-i.e., not using background knowledge and/or domain-specific language bias in learning, the search space is practically intractable.

Genetic Programming (GP) is a paradigm in machine learning that employs evolutionary search in the course of learning a target concept/function (Koza, 1992). It is an extension of John Holland's genetic algorithm 1975 in which the population consists of computer programs of varying sizes and shapes (Koza, 1992). GP automatically solves problems without having to tell the computer explicitly how to do it. GP is a systematic and domain-independent method for having computers automatically solve problems starting from a high-level statement of problem formulation. It generates a solution to a given problem by maintaining a population of S-expressions (i.e., LISP functions) which is evolved over a number of generations by the application of a tree-based crossover operator, a reproduction operator, and a mutation operator over the individuals in the population. Individual programs are evaluated by their fitness: Programs with higher fitness are selected for a crossover with a higher probability. Genetic materials for solving a problem are acquired incrementally from generation to generation until a solution that satisfies all the fitness cases is found (or the number of generations is exhausted). On learning Boolean even-N-parity functions, GP has been applied, by Koza, on solving the problem for a maximum N of up to 11. Koza used automatically defined functions for solving the problem (Koza, 1994).

An approach to solving Boolean parity problems for very large N is by using smooth uniform crossover, sub-machine-code GP, and interacting demes (i.e., sub-populations) running on separate workstations (Poli *et al*., 1999). More precisely, uniform crossover, inspired by that in the genetic algorithm, selects two points in the parents that are located within the "common" sub-tree structure of the two parents for a crossover with a probability of 0.5. Sub-machine-code GP exploits the bit-level parallelism of a CPU to do GP by making the CPU execute the same program on different data in parallel and independently; hence, it is possible to evaluate the same GP program on multiple fitness cases at the same time. Finally, a parallel implementation is employed in which GP sub-populations, or demes, are distributed over some number of workstations. This "three-pronged" approach to GP allows the system to solve even-N-parity problems up to an N of 22, which is the largest attempted complexity of the problem so far.

Yet another approach in GP for learning Boolean even-N-parity functions is called Traceless Genetic Programming (TGP) (Oltean, 2004). TGP is a novel method combining a technique for building individuals and a technique for representing individuals. More precisely, TGP does not explicitly store the mathematical expression for each individual-instead, the output values for the fitness cases are stored. TGP uses two genetic operators-crossover and insertion. In the crossover, selected parents are recombined to form new individuals. The insertion operator chooses a point in an individual and replaces it with a relatively simple expression: This can counter the problem of bloating in Genetic Programming. Finally, the algorithm starts with a random initial population of expressions, and new generations are produced by using the insertion operator and crossover operator with some probabilities. The approach was applied on solving the even-N-parity problem up to an N = 8. The runtime of the approach is at least an order of magnitude faster than standard GP.

An approach called Self-Modifying Cartesian Genetic Programming (SMCGP) was also proposed to solve the problem of learning Boolean even-N-parity functions (Miller and Harding, 2008). In SMCGP, a genotype-to-phenotype mapping is used-i.e., the genetic programming is developmental. More precisely, each genotype and phenotype are a directed graph. During the course of evolution, a genotype graph is mapped to a phenotype graph which is executed. In each generation, the best 5 individuals are automatically promoted to the next generation. A self-modification operator can be applied to a phenotype graph to change its internal structures. Other individuals are produced by using selection and mutation with some probabilities. The approach was applied on solving the even-N-parity problem up to an N = 8. It was demonstrated that such an approach is more efficient than the original CGP in solving the even-N-parity problem.

Multi Expression Programming (MEP) is another approach proposed for learning Boolean even-N-parity functions (Oltean, 2003). The approach to GP involves storing multiple solutions in a single chromosome. It starts by creating a random population of individuals. The following steps are repeated until a given number of generations is reached. Two parents are chosen using a selection procedure. The parents are randomly recombined to obtain two offspring. The offspring are then considered for mutation. The best offspring replaces the worst individual in the current population if the offspring is better. Finally, the system returns the best individual evolved over the total number of generations. The approach was applied to solve the even-N-parity problem up to N = 5. However, it was demonstrated that MEP significantly outperformed standard GP in terms of reducing the size of the population involved.

Using background knowledge in GP concerns the incorporation of domain-specific functions in the function set. A random sampling of fitness cases concerns the use of random samples (instead of the full set of fitness cases) in GP. The use of domain-specific background knowledge facilitates problem decomposition and functional composition for producing useful genetic materials in evolution. Using random samples of fitness cases allows evolution to be conducted more computationally efficiently since fitness evaluation of the individuals needs only to handle a relatively small subset of the potentially massive amounts of fitness cases. Using automatically defined functions facilitates automatic problem decomposition and allows for the reusability of invented sub-functions in the course of program evolution. So far, no approach in genetic programming has attempted to combine the use of high-arity background knowledge, the use of automatically defined functions, and random sampling of fitness cases to leverage the advantage of automatic large-scale problem decomposition and boosting efficiency in evolution for problem-solving by GP. This study demonstrates the advantages of combining the use of high-arity background knowledge, automatically defined functions, and random sampling of fitness cases in GP in solving the even-N-parity problem. More precisely, it can be shown that such an approach can significantly reduce the total number of individuals to be processed by GP in solving the even-N-parity problem; hence, the efficiency of GP can be significantly improved.

Materials and Methods

Using Background Knowledge in GP

The use of background knowledge is critical and essential in some paradigms in Machine Learning such as Inductive Logic Programming (Muggleton, 1991). More precisely, domain-specific knowledge can be incorporated into the learning process when inducing a hypothesis or a target function to explain the input data. The use of background knowledge relevant to solving a particular problem facilitates the learning of the target concept (Srinivasan *et al*., 2003). In cases where the concept or the function to be learned is compositional (i.e., it is composed of sub-concepts and/or sub-functions relevant to solving the problem), the use of background knowledge facilitates problem decomposition in learning as in using a divide-and-conquer strategy in problem-solving (Koza, 1994).

In learning Boolean even-N-parity functions, we can provide GP background functions such as some even-parity functions for $k < N$ in addition to primitive Boolean functions such as AND, OR, and XOR. The provision of such background functions to GP is arguably reasonable for two reasons. Firstly, background functions can be learned incrementally. For example, we can first learn the definition of the even-3-parity function using primitive logic gates (e.g., OR, XOR) before we try to learn the even-5-parity function where even-3-parity may serve as a background function. Hence, we can acquire background knowledge incrementally and background functions can be acquired systematically. Secondly, background functions can be created in a relatively straightforward manner if we rely on a "non-circuit" general definition of odd parity (i.e., to check if the number of input Boolean arguments is an odd number). Doing so still allows us to discover the logic circuit of an even-N-parity function for some N. Such an approach is, therefore, useful for discovering the "blueprint" of a logic circuit for an even-N-parity generator (Elprocus, 2013).

Using Automatically Defined Functions in GP

A target function for a problem can usually be expressed as a hierarchy of sub-functions in which subfunctions may be reused repeatedly within the target function (Koza *et al*., 1996). An Automatically Defined Function (ADF) is a function that is dynamically evolved during a run of Genetic Programming and which may be called by a calling program (or sub-program) that is concurrently being evolved. When automatically defined functions are being used, a program in the population consists of a hierarchy of one (or more) reusable function-defining branches (i.e., ADFs) along with a main result-producing branch. Usually, ADFs contain one or more formal parameters where the ADFs are reused with different instantiations of the parameters by the main result-producing branch.

When automatically defined functions are used, it is necessary to determine the architecture of the evolved programs. The specification of the architecture consists of (1) The number of function-defining branches in the overall program, (2) The number of parameters possessed by each function-defining branch, and (3) If there is more than one function-defining branch, the nature of the hierarchical references allowed between them.

The use of ADFs allows for automatic problem decomposition when solving a problem by GP. It also allows for the reusability of a solution to a sub-problem within the overall problem the same sub-program can be reused multiple times within the overall solution program. These advantages of ADFs are useful for evolution because problems as well as their solution programs may be compositional in nature: Automatic discovery of subprograms facilitates the discovery of a sub-problem within the overall problem and the use of reusable subprograms cuts down the need and effort for re-doing work and rediscovery of partial solutions to a problem. Hence, ADFs are highly desirable in GP.

Random Sampling Fitness Cases in GP

As the number of fitness cases for a problem can be potentially large, it would be desirable to use only a relatively small subset of fitness cases when evaluating the fitness of an individual in the population. For example, in solving the even-N-parity problem (s), the number of fitness cases amounts to O (2M) when N = 21. When N = 30, the number of fitness cases amounts to O (1B). It takes a lot of computation to fully evaluate the fitness of an individual program over an entire set of fitness cases. However, a solution program that has solved only a subset of all the fitness cases may be able to solve a large fraction of or even all of the fitness cases. Since evaluating the fitness of an individual only on a subset of fitness cases can dramatically cut down on the amount of computation needed, it is desirable to explore the use of random sampling of fitness cases in GP as such an approach can boost the efficiency of GP. In addition, using random sampling of fitness cases may make evolution more resistant to bloating (Harper, 2012). The technique is also useful for avoiding overfitting (Gonçalves and Silva, 2011) and in finding novel solutions (Klein and Spector, 2008).

A number of random sampling techniques had been used and experimented with. One method is called interleaved sampling, which is a deterministic-based sampling method (Gonçalves and Silva, 2011). It uses the entire training set to compute fitness in some generations and uses a single fitness case in others. The approach was motivated by the idea of balancing learning and overfitting through the interleaving of fitness cases, which attempts to avoid local optima. An earlier method, called historical subset selection, uses misclassified instances from previous GP runs at each generation using the best individual of the population (Gathercole and Ross, 1994). The subset contains not only difficult cases but also easy ones that are collected at earlier generations. In fixed random selection, the selection of fitness cases is based on a uniform probability among the training subset (Zhang and Joung, 1999). A fixed number of cases are selected for every generation.

Our approach to random sampling is very similar to that of fixed random selection. More precisely, a fixed number of fitness cases are selected for each generation to be used in the evaluation of the fitness of the individuals in the population. However, we employ multi-core threading in our implementation of the GP kernel. In some of the experiments in solving the even-N-parity problem, we employ fixed random selection for each sub-population at each generation.

Results

We have attempted seven different experiments in learning Boolean even-N-parity functions. More precisely, we tackled the problem by GP for N = 8, 9, 11, 13, 15, 17, and 20. In solving the even-N-parity problem for each N, we ran GP for four independent trials and selected the best run out of the trials. Our GP kernel was able to learn the target function for each target N in an even-N-parity problem.

In solving a particular even-N-parity problem, we provided two kinds of background knowledge: (1) Primitive Boolean functions such as AND, OR, and XOR; (2) Some even-k-parity functions for k < N such as the even-3-parity function in solving the even-8parity problem. We have used two automatically defined functions each has an arity of five. In a random sampling of fitness cases, we chose a fitness case subset of size 500 at each generation. The fitness evaluation method used was tournament selection (Fang and Li, 2010).

For each even-N-parity problem, we evaluated the size of the population needed to find the target Boolean function and also the total number of individuals processed by GP to find the target function.

Experimental Results

Generally speaking, as N increases, the size of the population needed to find the target Boolean function and the total number of individuals processed by GP (in finding the target function) both increase in numbers.

In Fig. 1, N (the x-axis) is plotted against the size of the population needed to find the target Boolean function (the y-axis).

As we can see from Fig. 1, generally, the population size needed for finding the target Boolean function increases with respect to the arity N in an even-N-parity problem.

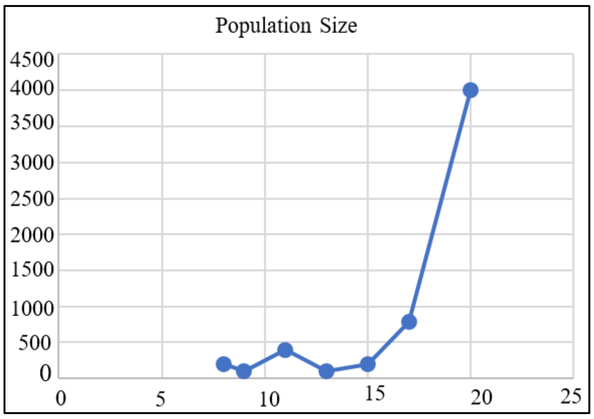


Fig. 1: How N (x-axis) varies with population size (y-axis) needed to find the target Boolean function in an even-N-parity problem

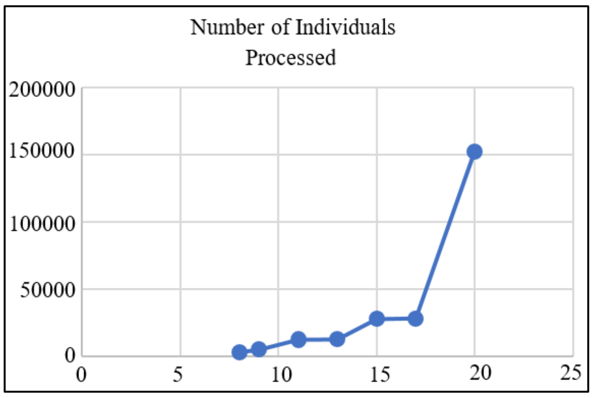


Fig. 2: How N (x-axis) varies with the total number of individuals processed by GP (y-axis) to find the target Boolean function in an even-N-parity problem

This is because as the arity of a problem becomes bigger, the number of terminals (i.e., variables) involved in instantiating the parameters of the background functions in the function set also increases. Hence, we have a larger hypothesis space as N increases (i.e., as the number of terminals increases). Therefore, as N increases, we need larger populations of individuals to increase the probability that a target Boolean function can be generated within the population.

In Fig. 2, N (the x-axis) is plotted against the total number of individuals processed by GP to find the target Boolean function (the y-axis).

As we can see from Fig. 2, generally, the total number of individuals processed by GP to find the target Boolean function increases with respect to the arity N in an even-N-parity problem. As the arity N of an even-N-parity problem becomes bigger, the hypothesis space becomes larger as well since the expected arity of a background function also becomes larger if we want to carry out large-scale automatic problem decomposition.

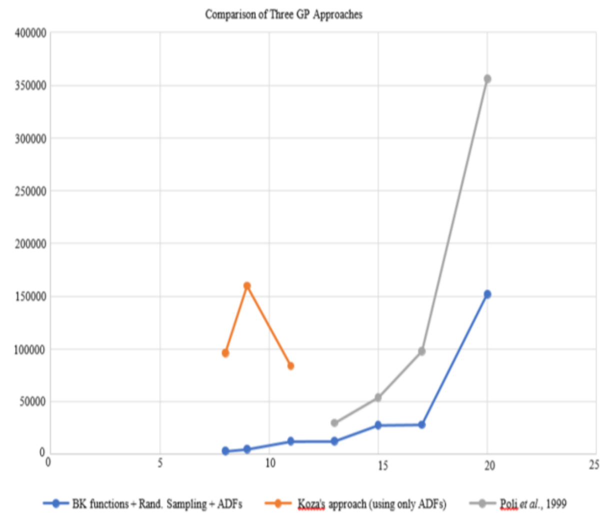


Fig. 3: An experimental comparison of (1) Our approach in GP, (2) Koza's approach in GP, and (3) (Poli *et al*., 1999) approach in GP in terms of the total number of individuals processed by GP versus the arity N of an even-N-parity problem

Table 1: An experimental comparison of (1) Our approach in GP, (2) Koza's approach in GP, and (3) (Poli *et al*., 1999) approach in GP in terms of the total number of individuals processed by GP versus the arity N of an even-N-parity problem

Approach \Arity	7	8	11	13	15	17	20
Koza's	96000	160000	84000	N/A	N/A	N/A	N/A
Poli *et al.* (1999)	N/A	N/A	N/A	30000	54200	98000	356400
Ours	8000	12000	16000	15000	31000	30000	151000

More precisely, GP needs to exchange and/or process more genetic materials in the population across the generations by performing more crossover operations as N increases so that a sufficient amount of processing of genetic materials is done to produce a target Boolean function for a given even-N-parity problem.

In Fig. 3 and in Table 1, our approach of using background functions, random sampling of fitness cases, and automatically defined functions are compared against Koza's approach using only ADFs and (Poli *et al*., 1999) approach of using smooth uniform crossover, submachine code GP and demes.

It can be seen that our approach needs to process a significantly lower total number of individuals in order to find a target Boolean function for a given even-N-parity problem. More precisely, when solving even-N-parity problems for N = 8, 9, and 11, Koza's approach of using only ADFs needs to process a total of 96000, 160000, and 84000 individuals respectively. When solving even-N-parity problems for N = 13, 15, 17, and 20, the (Poli *et al*., 1999) approach needs to process a total of 30000, 54200, 98000, and 356400 individuals respectively.

Our approach is processing a significantly smaller total number of individuals when learning the target Boolean function for a given even-N-parity problem mainly because of the following reasons:

- Using background functions such as even-k-parity for k < N facilitates problem decomposition in the course of learning a target Boolean function. More precisely, a lot of function compositions otherwise using primitive Boolean functions such as AND, OR and XOR can be saved by using background functions. This can reduce the amounts of genetic materials that need to be processed by the crossover operation; hence, the total number of individuals that need to be processed can be significantly reduced as potentially "big chunks" of primitive functions inter-composed can be succinctly represented by background functions. In other words, a lot of crossover operations needed for composing a sub-concept in an even-N-parity problem can be saved by the use of a sub-concept directly. Furthermore, as background functions are reusable by the main result-producing branch in a program, the total number of individuals needed to be processed can be exponentially reduced by repeated use of the same background functions: The effect is similar to the use of caching in computation (i.e., unnecessary repeated computational steps can be avoided). As we can see, Koza's use of ADFs cannot reinvent sub-concepts represented directly by background functions. This suggests that a sub-concept represented by a background function can potentially possess a level of functional complexity beyond the capacity of function invention using ADFs alone. Poli *et al*. (1999) approach mainly uses uniform crossover to increase the diversity of genetic materials chosen in a crossover operation. While such an approach may increase the probability of finding a target individual with a bias of using a diverse set of primitive Boolean functions, it is not as effective in "short-cutting" the composition of a sub-concept that can be directly represented by a background function. Hence, their approach is processing a larger total number of individuals when learning a target Boolean function
- Using random sampling for selecting a subset of fitness cases can increase the diversity of genetic materials chosen in the crossover from generation to generation due to diversity in fitness cases. More precisely, such an approach promotes the retainment of diverse genetic materials in the population from generation to generation. Hence, premature convergence to a sub-optimal solution can be avoided and convergence to a global solution can be facilitated

- Using background functions together with ADFs can create synergy between the kind of problem decomposition employed by ADFs and the kind induced by the use of background functions. More precisely, ADFs can now be created to "bridge the gap" in connecting different sub-concepts represented by background functions. For example, an ADF may be created by using particular primitive Boolean functions to represent a sub-concept not expressed by but needed by the use of specific background functions. In other words, problem decomposition by using ADFs is shaped by the use of specific background functions and it is carried out more purposefully. Therefore, using background functions together with ADFs enhances the effectiveness of problem decomposition in learning a target concept by exploiting the synergy between the two kinds of problem decomposition strategies. The increase in effectiveness in problem decomposition reduces the total number of individuals needed to be processed by GP since a sub-concept invented and/or employed is reusable by the main result-producing branch in an evolving program

Discussion

When using background functions together with automatically defined functions, we can see that GP would employ the following strategy for solving a particular even-N-parity problem. An ADF would be created to represent a sub-concept by composing primitive Boolean functions over a subset of its parameters (e.g., (AND (NOT A0) A1)) where A0 and A1 are two particular parameters of the ADF. Such a sub-function is then employed in composing with other background functions (e.g., even-3-parity). A background function in the main result-producing branch of an evolving program can be called compositionally with ADFs and/or other background functions (e.g., even-8-parity can be called even-3-parity in one of its arguments). A target Boolean function (e.g., even-11-parity) is created by inter-composing ADFs and/or background functions so that all parameters in the target Boolean function are used by some functions within the tree of composing functions. Hence, a target concept is expressed by a hierarchy of sub-concepts, which makes for an alternative definition of a target Boolean function.

In Fig. 4, we have the learned definition of even-11-parity. To create such an alternative definition of the target Boolean function, GP exploits the following strategies in evolution:

- Repeated use of the same background function (e.g., even-10-parity)
- Calling an ADF by a background function (e.g., even-10-parity calls ADF1)
- An ADF may call a background function (e.g., ADF1 calls even-3-parity)
- Using the same term within a function (e.g., D2 appears as two different arguments in even-8-parity), creates a specific context of terminals (i.e., variables) in which an even-k-parity function can be true or false over the context
- Relatively higher arity background function(s)
- (e.g., even-10-parity) within the target
- Boolean functions (e.g., even-11-parity) are used to define the overall basic structure of the target solution
- All terminals in the terminal set are used by some functions within the entire learned definition

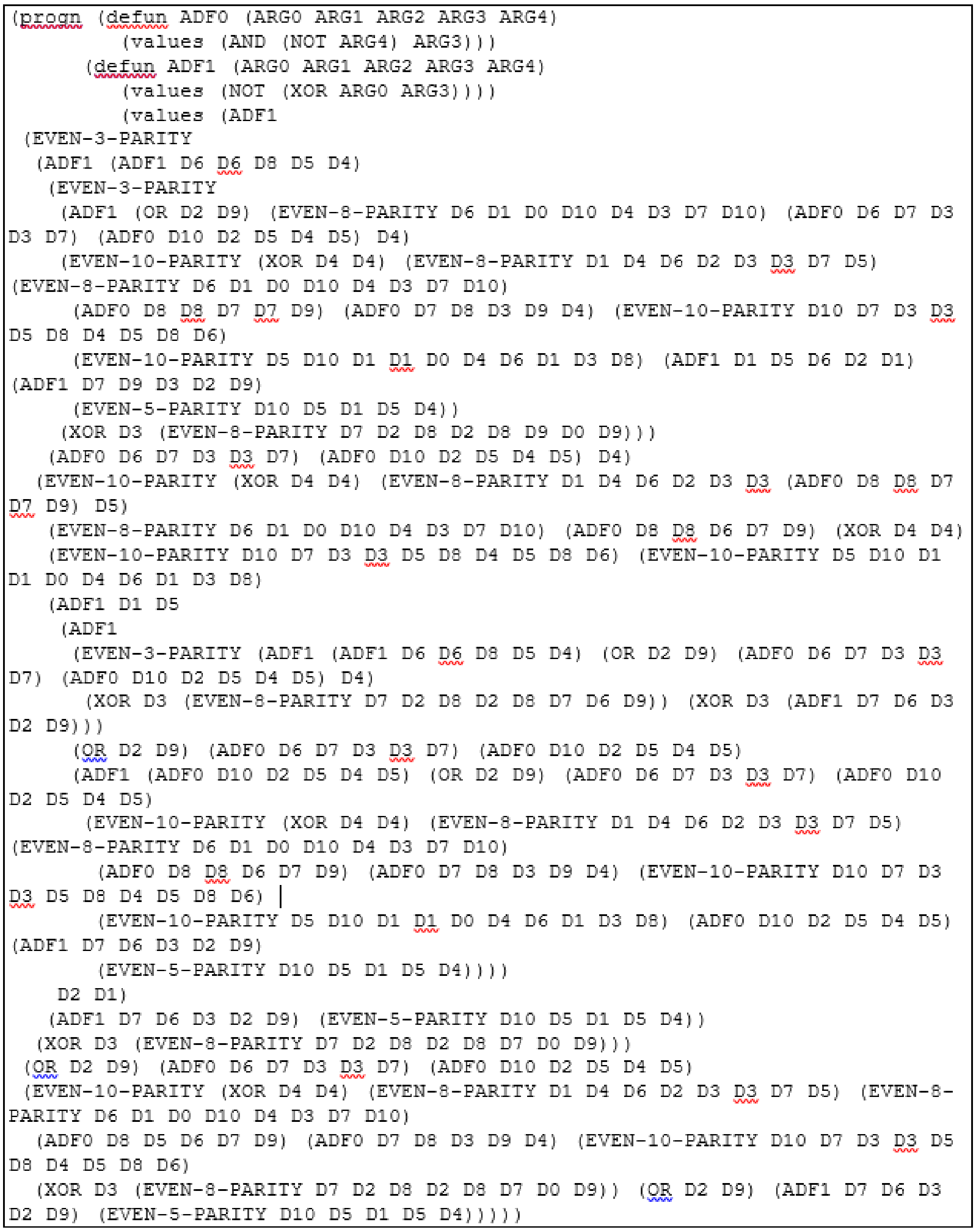
In Fig. 5, we can see a learned alternative definition for the target Boolean function even-20-parity. Again, we can see a similar strategy used by GP to learn the definition. More precisely, even-k-parity functions ($k < N$) of relatively big k are used to define the overall basic structure of the target solution. In this case, even-19-parity, even-17-parity, and even-15-parity are used for defining the basic layout of the target solution they define the basic organization of the solution tree.

Again, ADFs and primitive Boolean functions (e.g., XOR) are used for "gluing" together the entire definition. The same terminal can appear as different arguments to a background function and the same background functions are called multiple times within the entire definition. All the terminals are instantiated as arguments to some functions within the entire definition.

Our experience in conducting the experiments is that the larger the arity of an even-N-parity problem, the bigger would be the requirement on the population size for successfully finding a target solution.

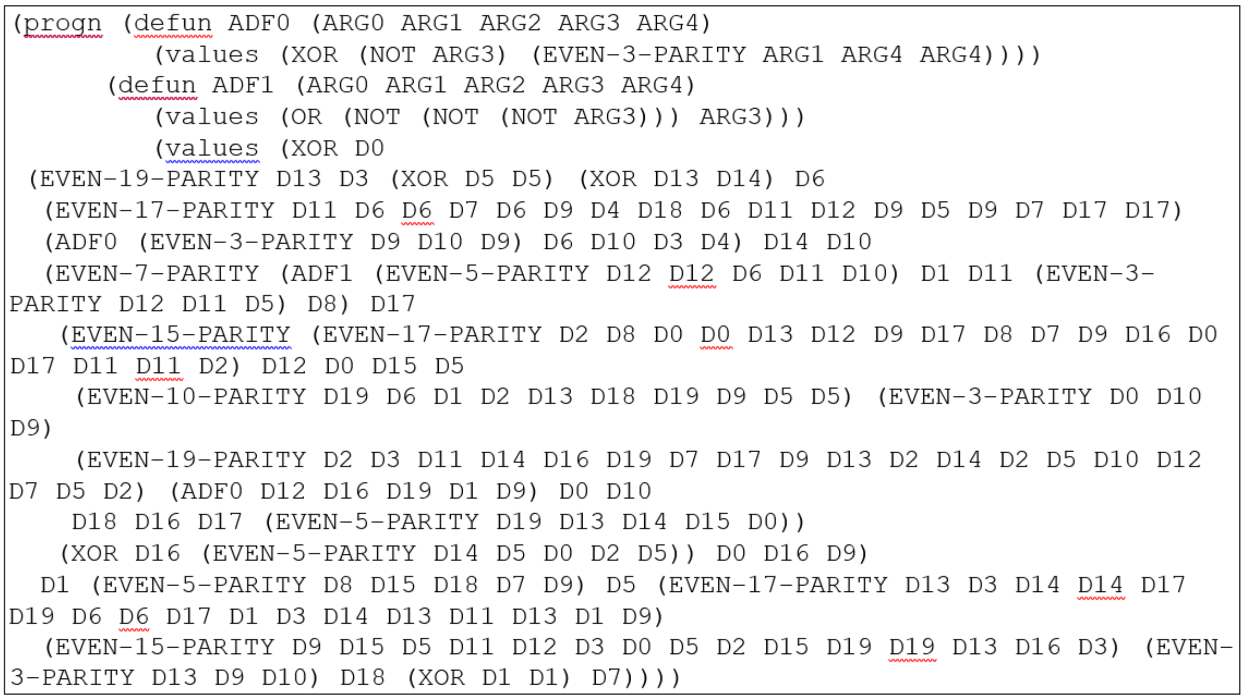
Random sampling serves to create a subset of fitness cases to use with fitness evaluation at each generation. Exploiting this strategy, GP is much more efficient in its computation since a subset of O (10×) to O (10000×) smaller in size is instead used in fitness evaluation for an individual program. Hence, GP has a speedup of O (10×) to O (10000×) in its computation as a result of a dramatic reduction in the number of fitness cases employed in fitness evaluation. This makes an intractable problem of evaluating O (1M) fitness cases repeatedly a tractable problem.

The strategy employed by GP when using random sampling is that it progressively retains a more and more diverse set of genetic materials from generation to generation until a target solution is found. This is because the genetic materials retained by crossover are dependent upon the fitness cases chosen for a particular generation. Varying the fitness cases from generation to generation, thus, allows GP to retain genetic materials from a variety of individuals within the population. In other words, genetic materials are "recruited" from different individuals from one generation to the next and the best overall individual is built up incrementally over time until it would become the target solution.



```
(progn (defun ADF0 (ARG0 ARG1 ARG2 ARG3 ARG4)
         (values (AND (NOT ARG4) ARG3)))
      (defun ADF1 (ARG0 ARG1 ARG2 ARG3 ARG4)
         (values (NOT (XOR ARG0 ARG3))))
         (values (ADF1
 (EVEN-3-PARITY
  (ADF1 (ADF1 D6 D6 D8 D5 D4)
   (EVEN-3-PARITY
    (ADF1 (OR D2 D9) (EVEN-8-PARITY D6 D1 D0 D10 D4 D3 D7 D10) (ADF0 D6 D7 D3
D3 D7) (ADF0 D10 D2 D5 D4 D5) D4)
    (EVEN-10-PARITY (XOR D4 D4) (EVEN-8-PARITY D1 D4 D6 D2 D3 D3 D7 D5)
(EVEN-8-PARITY D6 D1 D0 D10 D4 D3 D7 D10)
     (ADF0 D8 D8 D7 D7 D9) (ADF0 D7 D8 D3 D9 D4) (EVEN-10-PARITY D10 D7 D3 D3
D5 D8 D4 D5 D8 D6)
     (EVEN-10-PARITY D5 D10 D1 D1 D0 D4 D6 D1 D3 D8) (ADF1 D1 D5 D6 D2 D1)
(ADF1 D7 D9 D3 D2 D9)
     (EVEN-5-PARITY D10 D5 D1 D5 D4))
    (XOR D3 (EVEN-8-PARITY D7 D2 D8 D2 D8 D9 D0 D9)))
   (ADF0 D6 D7 D3 D3 D7) (ADF0 D10 D2 D5 D4 D5) D4)
  (EVEN-10-PARITY (XOR D4 D4) (EVEN-8-PARITY D1 D4 D6 D2 D3 D3 (ADF0 D8 D8 D7
D7 D9) D5)
   (EVEN-8-PARITY D6 D1 D0 D10 D4 D3 D7 D10) (ADF0 D8 D8 D6 D7 D9) (XOR D4 D4)
   (EVEN-10-PARITY D10 D7 D3 D3 D5 D8 D4 D5 D8 D6) (EVEN-10-PARITY D5 D10 D1
D1 D0 D4 D6 D1 D3 D8)
   (ADF1 D1 D5
    (ADF1
     (EVEN-3-PARITY (ADF1 (ADF1 D6 D6 D8 D5 D4) (OR D2 D9) (ADF0 D6 D7 D3 D3
D7) (ADF0 D10 D2 D5 D4 D5) D4)
      (XOR D3 (EVEN-8-PARITY D7 D2 D8 D2 D8 D7 D6 D9)) (XOR D3 (ADF1 D7 D6 D3
D2 D9)))
     (OR D2 D9) (ADF0 D6 D7 D3 D3 D7) (ADF0 D10 D2 D5 D4 D5)
     (ADF1 (ADF0 D10 D2 D5 D4 D5) (OR D2 D9) (ADF0 D6 D7 D3 D3 D7) (ADF0 D10
D2 D5 D4 D5)
      (EVEN-10-PARITY (XOR D4 D4) (EVEN-8-PARITY D1 D4 D6 D2 D3 D3 D7 D5)
(EVEN-8-PARITY D6 D1 D0 D10 D4 D3 D7 D10)
       (ADF0 D8 D8 D6 D7 D9) (ADF0 D7 D8 D3 D9 D4) (EVEN-10-PARITY D10 D7 D3
D3 D5 D8 D4 D5 D8 D6) |
       (EVEN-10-PARITY D5 D10 D1 D1 D0 D4 D6 D1 D3 D8) (ADF0 D10 D2 D5 D4 D5)
(ADF1 D7 D6 D3 D2 D9)
       (EVEN-5-PARITY D10 D5 D1 D5 D4))))
    D2 D1)
   (ADF1 D7 D6 D3 D2 D9) (EVEN-5-PARITY D10 D5 D1 D5 D4))
  (XOR D3 (EVEN-8-PARITY D7 D2 D8 D2 D8 D7 D0 D9)))
 (OR D2 D9) (ADF0 D6 D7 D3 D3 D7) (ADF0 D10 D2 D5 D4 D5)
 (EVEN-10-PARITY (XOR D4 D4) (EVEN-8-PARITY D1 D4 D6 D2 D3 D3 D7 D5) (EVEN-8-
PARITY D6 D1 D0 D10 D4 D3 D7 D10)
  (ADF0 D8 D5 D6 D7 D9) (ADF0 D7 D8 D3 D9 D4) (EVEN-10-PARITY D10 D7 D3 D3 D5
D8 D4 D5 D8 D6)
  (XOR D3 (EVEN-8-PARITY D7 D2 D8 D2 D8 D7 D0 D9)) (OR D2 D9) (ADF1 D7 D6 D3
D2 D9) (EVEN-5-PARITY D10 D5 D1 D5 D4))))))
```

Fig. 4: The learned definition of even-11-parity. The learned alternative definition for the target Boolean function is composed of relatively higher arity even-k-parity functions (k < N) with ADFs and primitive Boolean functions serving to "gluing" the overall definition together



```
(progn (defun ADF0 (ARG0 ARG1 ARG2 ARG3 ARG4)
         (values (XOR (NOT ARG3) (EVEN-3-PARITY ARG1 ARG4 ARG4))))
      (defun ADF1 (ARG0 ARG1 ARG2 ARG3 ARG4)
         (values (OR (NOT (NOT (NOT ARG3))) ARG3)))
         (values (XOR D0
 (EVEN-19-PARITY D13 D3 (XOR D5 D5) (XOR D13 D14) D6
  (EVEN-17-PARITY D11 D6 D6 D7 D6 D9 D4 D18 D6 D11 D12 D9 D5 D9 D7 D17 D17)
  (ADF0 (EVEN-3-PARITY D9 D10 D9) D6 D10 D3 D4) D14 D10
  (EVEN-7-PARITY (ADF1 (EVEN-5-PARITY D12 D12 D6 D11 D10) D1 D11 (EVEN-3-
PARITY D12 D11 D5) D8) D17
   (EVEN-15-PARITY (EVEN-17-PARITY D2 D8 D0 D0 D13 D12 D9 D17 D8 D7 D9 D16 D0
D17 D11 D11 D2) D12 D0 D15 D5
    (EVEN-10-PARITY D19 D6 D1 D2 D13 D18 D19 D9 D5 D5) (EVEN-3-PARITY D0 D10
D9)
    (EVEN-19-PARITY D2 D3 D11 D14 D16 D19 D7 D17 D9 D13 D2 D14 D2 D5 D10 D12
D7 D5 D2) (ADF0 D12 D16 D19 D1 D9) D0 D10
    D18 D16 D17 (EVEN-5-PARITY D19 D13 D14 D15 D0))
   (XOR D16 (EVEN-5-PARITY D14 D5 D0 D2 D5)) D0 D16 D9)
  D1 (EVEN-5-PARITY D8 D15 D18 D7 D9) D5 (EVEN-17-PARITY D13 D3 D14 D14 D17
D19 D6 D6 D17 D1 D3 D14 D13 D11 D13 D1 D9)
  (EVEN-15-PARITY D9 D15 D5 D11 D12 D3 D0 D5 D2 D15 D19 D19 D13 D16 D3) (EVEN-
3-PARITY D13 D9 D10) D18 (XOR D1 D1) D7))))
```

Fig. 5: The learned definition of even-20-parity. Again, relatively higher arity background functions (e.g., even-19parity, and even-17-parity) define the overall basic structure of the target solution. ADFs and primitive Boolean functions (e.g., XOR) are used for "gluing" the overall definition together

Conclusion

Learning Boolean functions for the even-N-parity problem is a benchmark problem because it is a difficult problem for evolutionary computation. Most approaches proposed have been focusing on the representation of individuals and/or the effectiveness of crossover operation. Our proposed approach involves the use of high-arity background functions, automatically defined functions, and random sampling of fitness cases. Such an approach has the advantages of (1) Facilitating automatic large-scale problem decomposition in finding a target solution and (2) Promoting diversity in the retainment of genetic materials in the course of evolution. A major advantage of using high-arity background functions is that a lot of steps of crossover involved in function composition using primitive Boolean functions can be saved by the direct use of a sub-concept or a sub-function. In other words, the total number of individuals to be processed can be reduced by saving crossover operations that would have been carried out over function composition using primitive Boolean functions-instead, a sub-function, or a sub-concept is used directly. Secondly, random sampling of fitness cases promotes the use of a diverse set of genetic materials across the population from generation to generation. This facilitates the convergence of evolution to a global optimum due to diversity in evolution. Our experimental results show that the proposed "three-pronged" approach can significantly reduce the total number of individuals needed to be processed by GP. Although our approach in complexity, as per Fig. 3, is not linear with respect to N, it is growing less dramatically compared to the (Poli *et al*., 1999) approach (the state of the art in learning Boolean even-N-parity functions). In the future, we plan to experiment with our approach to solving the even-N-parity problem for much larger values of N (e.g., up to 26 and/or 30) to further investigate computational principles that might be involved in Genetic Programming in solving very large even-N-parity problems.

Acknowledgment

We are grateful to Koza for providing the coding of the genetic programming kernel using automatically defined functions. We are also grateful to the community that developed the publicly available framework gplearn for symbolic regression by genetic programming.

Funding Information

The authors have not received any financial support or funding to report the work.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that there were no ethical dilemmas.

References

Elprocus. (2013). What is Parity Generator and Parity Checker: *Types & Its Logic Diagrams*. https://www.elprocus.com/what-is-parity-generator-and-parity-checker-types-its-logic-diagrams/

Fang, Y., & Li, J. (2010). A review of tournament selection in genetic programming. In Advances in Computation and Intelligence: *5th International Symposium, ISICA 2010, Wuhan, China, October 22-24, 2010. Proceedings 5*, (pp. 181-192). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-16493-4_19

Gathercole, C., & Ross, P. (1994). Dynamic training subset selection for supervised learning in genetic programming. In *Parallel Problem Solving from Nature-PPSN III: International Conference on Evolutionary Computation The Third Conference on Parallel Problem Solving from Nature Jerusalem, Israel, October 9-14, 1994 Proceedings 3* (pp. 312-321). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-58484-6_275

Gonçalves, I., & Silva, S. (2011, October). Experiments on controlling overfitting in genetic programming. In *15th Portuguese Conference on Artificial Intelligence (EPIA 2011)*, (pp. 10-13).

Harper, R. (2012, July). Spatial co-evolution: quicker, fitter and less bloated. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, (pp. 759-766). https://doi.org/10.1145/2330163.2330269

Hirschberg, D. S., Pazzani, M. J., & Ali, K. M. (1994). Average Case Analysis of k-CNF and k-DNF learning algorithms. *Computational Learning Theory and Natural Learning Systems*, *2*, 15-28. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=f1f517fd4e43de9b23e03f8d7a44b816bcc32742. ISBN: 0262111896.

Klein, J., & Spector, L. (2008). Genetic programming with historically assessed hardness. In *Genetic Programming Theory and Practice VI*, (pp. 1-14). Boston, MA: Springer US. https://doi.org/10.1007/978-0-387-87623-8_5

Koza, J. R. (1992). Genetic Programming, On the Programming of Computers by Means of Natural Selection. A Bradford Book. *MIT Press*. ISBN: 9780262111706.

Koza, J. R. (1994). *Genetic programming II: Automatic Discovery of Reusable Programs*. MIT press.

Koza, J. R. andre, D., Bennett III, F. H., & Keane, M. A. (1996, July). Use of automatically defined functions and architecture-altering operations in automated circuit synthesis with genetic programming. In *Proceedings of the First Annual Conference on Genetic Programming*, (pp. 132-140). Stanford University MIT Press, Cambridge, MA. http://www.genetic-programming.com/jkpdf/gp1996adfaa.pdf

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*, 115-133. https://doi.org/10.1007/BF02478259

Miller, J. F., & Harding, S. L. (2008, July). Cartesian genetic programming. In *Proceedings of the 10th annual conference companion on Genetic and evolutionary computation* (pp. 2701-2726). https://doi.org/10.1145/1388969.1389075

Muggleton, S. (1991). Inductive logic programming. *New generation computing*, *8*, 295-318. https://doi.org/10.1007/BF03037089

Oltean, M. (2003, September). Solving even-parity problems using multi expression programming. In *Proceedings of the 5th International Workshop on Frontiers in Evolutionary Algorithms, The 7th Joint Conference on Information Sciences* (pp. 26-30). https://www.tcreate.org/oltean_fea2003_2.pdf

Oltean, M. (2004, June). Solving even-parity problems using traceless genetic programming. In *Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No. 04TH8753)*, (Vol. *2*, pp. 1813-1819). IEEE. https://doi.org/10.1109/CEC.2004.1331116

Poli, R., Page, J., & Langdon, W. B. (1999, July). Smooth uniform crossover, sub-machine code GP and demes: A recipe for solving high-order boolean parity problems. In *Proceedings of the Genetic and Evolutionary Computation Conference* (Vol. *2*, pp. 1162-1169).

Srinivasan, A., King, R. D., & Bain, M. E. (2003). An empirical study of the use of relevance information in inductive logic programming. *The Journal of Machine Learning Research*, *4*, 369-383. https://www.jmlr.org/papers/volume4/srinivasan03a/srinivasan03a.pdf

Veness, J., & Hutter, M. (2014). Online learning of K-CNF boolean functions. *arXiv preprint arXiv:1403.6863*. https://doi.org/10.48550/arXiv.1403.6863

Zhang, B. T., & Joung, J. G. (1999, July). Genetic programming with incremental data inheritance. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2*, (pp. 1217-1224). http://gpbib.cs.ucl.ac.uk/gecco1999/GP-460.pdf