Structure Abstraction and Genetic Programming

Tina Yu
Department of Computer Science
University College London
Gower Street, London WC1E 6BT, U. K.
T.Yu@cs.ucl.ac.uk
http://www.cs.ucl.ac.uk/staff/t.yu

Abstract- The selection of program representation can have strong impact on the performance of Genetic Programming. Previous work has shown that a particular program representation which supports structure abstraction is very effective in solving the general even-parity problem. We investigate program structures and analyze all perfect solutions in the search space to provide explanation of why structure abstraction is so effective with this problem. This work provides guidelines for the application of structure abstraction to other problems.

1 Introduction

Genetic Programming (GP) evolves problem solutions represented as program trees using a subtree crossover operator [Koza, 1992]. Recently, researchers have shown that such a search operator does not provide advantages in solving the artificial ant problem [Langdon and Poli, 1998a]. Chellapilla [1997] has devised search operators which provide better performance on this problem. Yet, according to the No Free Lunch Theorems [Wolpert and Macready, 1997], his operators will perform poorly on some other problems. There have also been attempts to design better program representations for GP [Koza, 1994; Angeline, 1994; Rosca and Ballard, 1996; Spector, 1996; Yu and Clack, 1998b]. A GP program can be represented in many different ways depending on the functions and terminals selected and the incorporation of module definition. Reports about the impact different functions and terminals have on GP [Haynes, 1998] and the advantages/disadvantages of Automatically Defined Functions (ADFs) on different problems [Koza, 1994] suggest that program representation is also problem dependent. It is only when the appropriate choices are made that the power of an evolutionary algorithm shows.

Previously, a program representation which incorporated a higher-order function `foldr` has been shown to be very effective in solving the general even-parity problem [Yu and Clack, 1998b]. This program representation contains a particular structure pattern named “structure abstraction” which we suggested to be the cause of the outstanding performance. However, when we used the same kind of program representation (with a different higher-order function) to solve the artificial ant problem [Koza, 1992], no performance advantage was found. This prompts us to investigate why structure abstraction is particularly effective with the general even-parity problem.

This paper starts by providing a formal definition of structure abstraction (Section 2). The application of structure abstraction to the general even-parity problem is summarized in Section 3. In Section 4, a systematic analysis of program structures in the search space is presented. Section 5 investigates all perfect solutions in the search space. Based on the analysis, the “effort” [Koza, 1992, Ch. 8] required to find a solution is estimated. This result indicates that due to structure abstraction, the complexity of the general even-parity problem is reduced to that of a much simpler Boolean function problem.

The experiments and their results are consistent with our analysis (Section 6). Indeed, structure abstraction provides a mechanism of hierarchical processing in problem solving, hence enables the solution to be found very efficiently (Section 7). As there is a trend in developing problem-specific evolutionary algorithms [Leonhardi *et al.*, 1998], we provide guidelines for the application of structure abstraction to other problems (Section 8). Finally, Section 9 presents our conclusion and future work.

2 Structure Abstraction in Program Evolution

The structure evolved by GP is a program tree[1] where the internal nodes are labelled with functions and the external nodes (leaves) are labelled with terminals. During program evolution, each subtree and leaf node acts as an independent unit and can be replaced by any other subtree and leaf node. It is hoped that the evolutionary process will promote “good” subtrees/leaf nodes to compose the program which solves the target problem.

When modules are incorporated in the program representation, some additional dynamics are introduced to the evolutionary process. A module is a block of code represented as a single tree. It might or might not have any input arguments. It normally, but not always, produces outputs. With ADFs [Koza, 1994], a module is provided with a name. Like

1. GP also evolves other types of structures such as linear sequences of instructions or graphs. This research focus on the evolution of program trees.

other functions and terminals, the names of ADFs can be used to compose the main program. In other words, an ADF can appear anywhere in the main program tree. During GP runs, both ADFs and the main program are evolved simultaneously.

Modules can also be incorporated into program representation through the use of higher-order functions [Yu and Clack, 1998b]. A higher-order function is a function which takes other functions as arguments or returns functions as outputs. By including higher-order functions in the function set, the evolved program trees contain subtrees which represent the function arguments. In our work, these function arguments are represented as λ abstractions.

λ abstractions are local function definitions, similar to function definitions in a conventional language such as C. The following is an example λ abstraction together with an equivalent C function.

```
(λ x (+ x 1))              (λ abstraction)
Inc (int x)                (C function)
{return (x+1);}
```

Similar to ADFs, λ abstractions are modules which evolve during GP runs. However, unlike ADFs, λ abstractions can only be positioned in a certain way in the program trees. As an argument to a higher-order function, a λ abstraction is constrained to sit underneath the higher-order function in the program tree hierarchy (See Figure 1). Consequently, a higher-order function and its function arguments group into a two-layer-hierarchy in the program trees. We term this two-layer-hierarchy "structure abstraction" as it groups into one structural unit during program creation and evolution. Although the contents of a grouping can change, i.e. λ abstractions also evolve, the two-layer-hierarchy structure of higher-order functions and its function arguments holds throughout the evolutionary process. Structure abstraction is a common property for any program representation using higher-order functions.

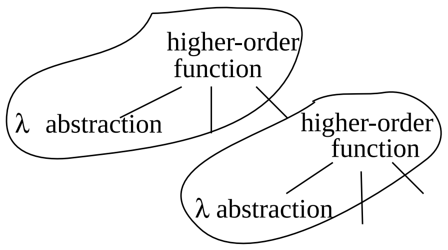

Figure 1. Structure abstraction in program tree hierarchy.

3 Applying Structure Abstraction to the Even-Parity Problem

The even-parity has been used by various researchers as a benchmark problem for GP to solve [Koza, 1992; Chellapilla, 1997; Poli, Page and Langdon, 1999]. This problem may have different number of inputs. For *N* number of inputs, the solution of the problem returns True if an even number of inputs are True. Otherwise, it returns False. This problem uses the following function and terminal sets:

- Function set: {`and,or,nand,nor`}. These are standard logic functions and are logically complete.
- Terminal set: {b_0, b_1, ..., b_{N-1}}, *N* Boolean inputs.

The test cases consists of all the 2^N possible combinations of *N* inputs. As the value of *N* increases, the problem becomes more difficult. Koza has experimented different value of *N* for this problem. Using the standard GP, he was able to solve the problem up to *N*=5. When *N*=6, none of his 19 runs found a 100%-correct solution [Koza, 1992]. Moreover, his results indicate that the number of evaluation required for the standard GP to find the solutions increases by about an order of magnitude for each increment of *N*.

The goal of this work is to evolve a *general solution* to the even-parity problem which works for any value of *N*, i.e. solving the general even-parity problem.

3.1 Program Representation with Structure Abstraction

The higher-order function selected to support structure abstraction for this problem is `foldr`. This function takes 3 arguments: the first one is a function, the second one is a value and the third one is a list. It returns a single value. To evaluate the `foldr` function, its first function argument is placed between each items in the list argument and the second argument is appended at the end. The resulting expression is then evaluated with parenthesis associated to the right. For example:

```
foldr (and) F [T,T,T]
= T and (T and (T and F))
= T and (T and F)
= T and F
= F
```

In the program tree, the function argument is represented as a λ abstraction. For the above example, the equivalent program tree is presented in Figure 2. The #1 and #2 stand for the first and the second arguments to the λ abstraction.

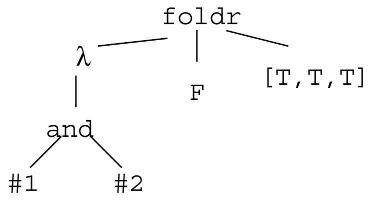

Figure 2. Program tree with `foldr` and λ abstraction.

It is important to notice that `foldr` reuses its function argument. This form of reuse is known as "implicit recursion" where the recursive calls of the function argument are carried out by the higher-order functions. For problems whose solutions contain regularities such as the general even-parity, this higher-order function is an ideal candidate to provide structure abstraction.

The inputs to the problem are given through a list named `L`, i.e. `L` is a list of *N* Boolean values. To allow the processing of each input item in the list, two more functions are included in the function set: `head` and `tail`. The function `head` takes a list as argument and returns the first element of the list. The `tail` function takes a list as argument and returns the list with its first element removed.

3.2 Type System

A type system is used to preserve the structure abstraction grouping in the program trees. The details of the type system is described in [Yu and Clack, 1998a]. In brief, the `foldr` function is specified with the following type:

```
foldr::(bool->bool->bool)->bool->
        [bool]->bool
```

This indicates that `foldr` has three arguments: the first one is a function which takes 2 Boolean values and returns a Boolean value. The second one is a Boolean value and the third one is list of Boolean values. It returns a Boolean value. During the generation of the initial population, each time `foldr` is selected to construct the program, a 2-input 1-output λ abstraction is generated as its function argument.

During program evolution, the structure abstraction grouping is maintained by the type system. Each node in the program tree is annotated with a type. A node is first selected from the first parent program; depending on the source of the node (the main program or λ abstractions) a node with the same source and the same type is selected from the second parent program. Crossover is then performed on these two selected nodes. In other words, λ abstractions can only crossover with λ abstractions. In the case of mutation, a new subtree is generated to replace the selected subtree from the first parent. The new subtree has a root node which has the same type as the root node of the subtree it replaces.

4 Program Structures in the Search Space

The search space of GP consists of all program trees that can be composed using the available functions and terminals. The number of such trees increases rapidly as the number of tree nodes increases. This fast growth is due to 1) the substantial number of different tree structures and 2) the enormous number of permutations in labelling the internal and external nodes using the provided functions and terminals. Without any restriction on its program trees, the GP search space is infinite. In practice, however, most implementations either restrict the number of tree nodes or the tree depth to prevent the otherwise explosive computation. This work applied a tree depth restriction of 4 to evolve the solutions to the general even-parity problem.

We analyze program structures in the search space. The term "program structure" is used to describe a program where a λ abstraction module is considered as one partial solution unit. Hence, only the semantics (outputs) of λ abstractions are considered in the program structures. The results of this study will be combined with the analysis of λ abstraction partial solution to estimate the effort required to solve the problem (see Section 5).

At first, the program structure of `foldr` is investigated because it is more complicated than the rest of the functions and terminals. A `foldr` program tree can only be constructed in some restricted ways due to the type constraints specified by the functions and terminals (Table 1).

Table 1: Functions and terminals with their types.

Name	Type
`and`	bool->bool->bool
`or`	bool->bool->bool
`nand`	bool->bool->bool
`nor`	bool->bool->bool
`foldr`	(bool->bool->bool)-> bool->[bool]->bool
`head`	[bool]->bool
`tail`	[bool]->[bool]
`L`	[bool]

The first argument of `foldr` is specified with a function type (bool->bool->bool). This function argument is constructed as a λ abstraction which takes 2 inputs of Boolean type and produces one output of Boolean type. The terminal set used to construct a λ abstraction is designed such that only arguments of the λ abstraction are included. Since both arguments are Boolean values, there is no terminal with the type of Boolean list that can be used to construct the λ abstraction. Consequently, those functions (`head`, `tail` and `foldr`) which require Boolean list type argument can never appear in a λ abstraction. Only the four Boolean operators (`and`, `or`, `nand` and `nor`) can be used to construct the internal nodes of λ abstractions. With two Boolean inputs and one Boolean output (each of them can be either True or False), there are total of 16 possible Boolean functions that a λ abstraction can generate.

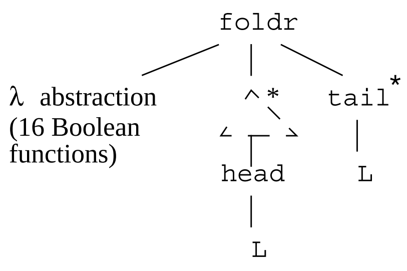

Figure 3. The `foldr` program tree structure.

The second argument of `foldr` is of type Boolean. There are six functions which return Boolean value hence can be used to build the internal nodes of the subtree. However, there is only one terminal `L` which can be used to construct the leaves of the subtree. As specified, `L` has type of Boolean list. To bridge between the leaf type (Boolean list) and the subtree root node type (Boolean), `head` function is used. Consequently, the subtree (`head L`) occupies the fringes of the subtree representing the second argument. The third argument of `foldr` is specified with Boolean list type. Among the functions and terminal, only `tail` and `L` return Boolean list type. Moreover, `tail` also requires its argument to be of Boolean list type. Consequently, the subtree representing the third argument can only be constructed using `tail` and `L`. Figure 3 shows the program structure for `foldr`. An * on the node indicates the node is optional.

The program structures in the search space is analyzed according to the number of `foldr` in the program trees. Figure 4 shows all the possible program structures containing 2 `foldr`. Figure 4(a) represents 1,536 program structures (Each of the two λ abstractions can generate 16 Boolean functions and there are 3 optional nodes in the tree). Figure 4(b) represents 4,096 program structures. In total, there are 5,632 program structures containing 2 `foldrs`.

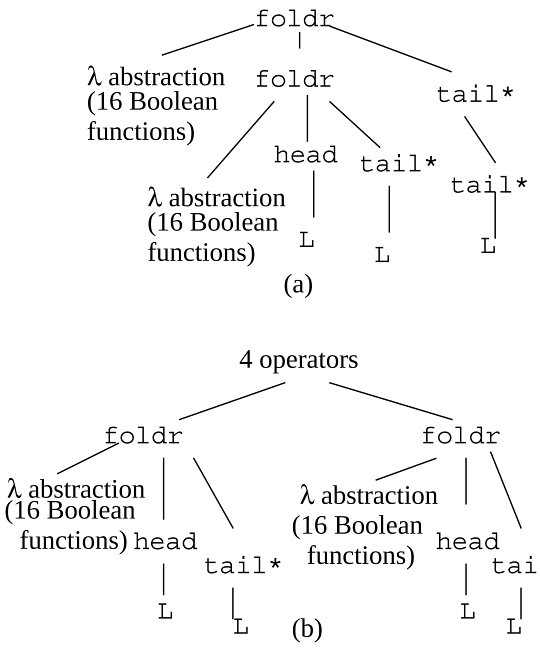
foldr
λ abstraction (16 Boolean functions)
foldr
tail*
head
tail*
λ abstraction (16 Boolean functions)
tail*
L
L
L
(a)
4 operators
foldr
foldr
λ abstraction (16 Boolean functions)
head
tail*
λ abstraction (16 Boolean functions)
head
tail*
L
L
L
L
(b)

Figure 4. Program structures with 2 `foldrs`.

Figure 5 shows program structures with 1 `foldr`. Figure 5(a) represents 192 program structures. Figure 5(b) represents 1,024 program structures (The two subtrees can be swapped with each other). Figure 5(c) represents 96 program structures and Figure 5(d) represents 512 program structures. In total, there are 1,824 program structures with 1 `foldr`.

Figure 6 shows program structures without `foldr`. Figure 6(a) represents 64 program structures. Figure 6(b) represents 64 program structures. Figure 6(c) represents 3 pro-

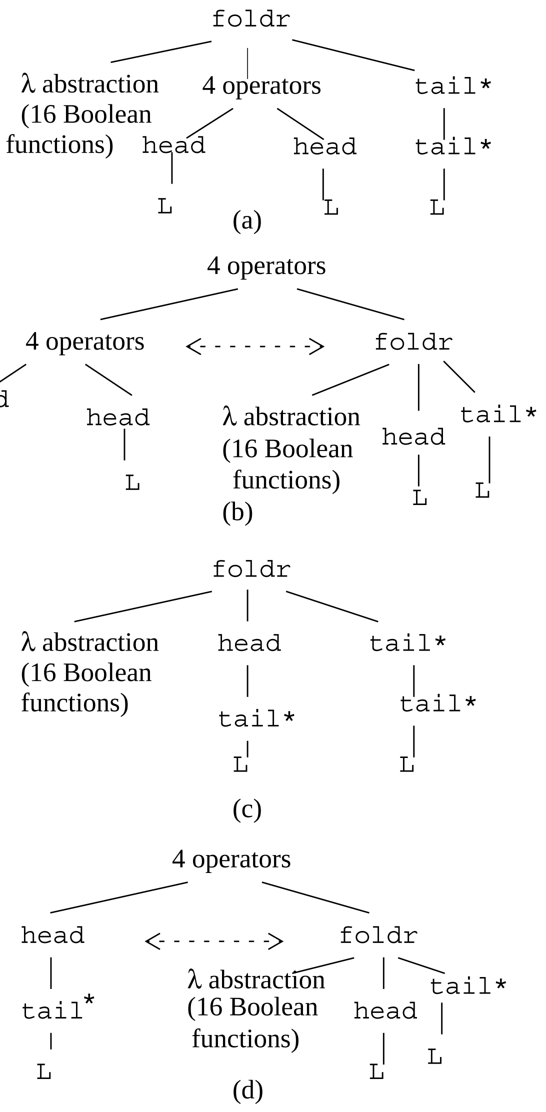

Figure 5. Program structures with 1 `foldr`.

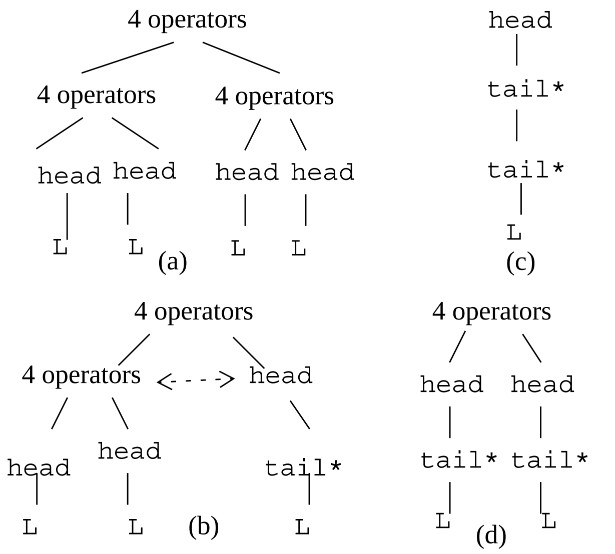

Figure 6. Program structures without `foldr`.

gram structures and Figure 6(d) represents 16 program structures. In total, there are 147 program structures without `foldr`.

In summary, there are 7,603 program structures in the search space. Among them, 5,632 with 2 `foldrs`; 1,824 with 1 `foldr` and 147 without `foldr`.

4.1 Fitness Distribution

The 7,603 program structures in the search space are capable of solving the general even-parity problem in different degree. A general solution to this problem has to be able to handle any number of inputs, i.e. *N* can be of any value. To evaluate how well each program structure is in solving the problem, we use even-2-parity and even-3-parity as test cases. This decision is based on the fact that the value of *N* has to be either even or odd. The test cases of even-2-parity help GP to learn to handle an even number of inputs while the even-3-parity test cases train GP to work on an odd number of inputs. Hence, there are a total of $2^2 + 2^3 = 12$ test cases. A program structure receives one point for each correctly handled test case. A perfect solution to the problem receives a fitness value of 12.

Figure 7 shows the fitness distribution in the search space. More than 60 percent of the program structures are of fitness 6. The number of program structures with other fitness falls dramatically away either side of the peak. There are only 29 program structures which score 12 and are prefect solutions to the general even-parity problem.

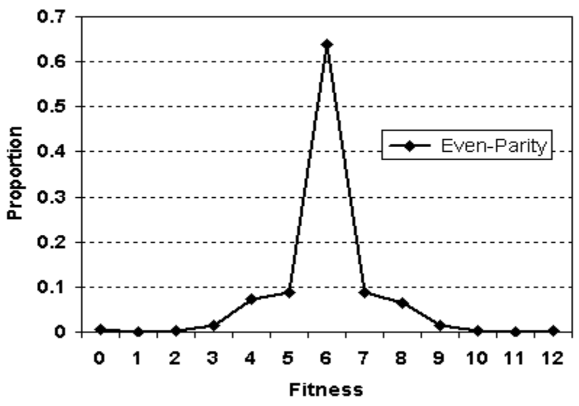

Figure 7. Fitness distribution in the search space.

5 Solutions in the Search Space

The 29 solutions in the search space are split into 11 groups (Tables 3 - 13). Programs in each group have the same structure but different λ abstraction values (Boolean functions). We calculate the probability to find each solution based on the program structure and its Boolean functions.

To successfully find a solution, it requires to find both its correct program structure and the correct Boolean functions. Based on the analysis of program structures in Section 4, the probability to find each program structure can be estimated. Meanwhile, the search of the 16 Boolean functions has been studied by Koza [Koza, 1992, page 228]. For each of the 16 Boolean functions, Koza did random search using a program tree with 31 nodes. The results are summarized in Table 2.

Table 2: 16 Boolean rules found using random search.

Rule	Random Search	Rule	Random Search
15	4.8	13	31.9
00	4.8	11	32.0
10	7.8	04	32.0
05	7.8	03	32.0
14	28.8	02	32.1
08	28.8	12	32.2
07	28.9	09	821.0
01	29.0	06	846.0

We use this table to measure the probability to find each of the Boolean functions (Koza called them Boolean rules). For example, rule 15 can be generated by random creation of 4.8 programs. The probability of success to this rule is therefore 1/4.8. Table 3 to Table 13 present the probability to find the 29 solutions. The caption indicates the solution. Columns 1 & 2 are the Boolean rules for the λ abstractions. Columns 3 & 4 are the probability to find each of the Boolean rules. Column 5 is the probability to find the program structure. Finally, column 6 is the probability to find the solution. It is the result of multiplying columns 3, 4 and 5. Table 13 is slightly different in that the solution only contains one λ abstraction. However, the method to measure the probability to find the solution is the same.

Table 3: nand (foldr λ_1(head L) L)(foldr λ_2 (head L) (tail L))

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r13	r6	1/31.9	1/846	256/7603	1.25e-6
r14	r6	1/28.8	1/846	256/7603	1.38e-6
r15	r6	1/4.8	1/846	256/7603	8.29e-6

Table 4: nand (foldr λ_1(head L)(tail L))(foldr λ_2 (head L) L)

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r6	r13	1/846	1/31.9	256/7603	1.25e-6
r6	r14	1/846	1/28.8	256/7603	1.38e-6
r6	r15	1/846	1/4.8	256/7603	8.29e-6

Table 5: nand (foldrλ_1(head L)(tail L))(foldrλ_2(head L)(tail L))

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r6	r6	1/846	1/846	256/7603	4.70e-8
r6	r14	1/846	1/28.8	256/7603	1.38e-6
r6	r15	1/846	1/4.8	256/7603	8.29e-6
r14	r6	1/28.8	1/846	256/7603	1.38e-6
r15	r6	1/4.8	1/846	256/7603	8.29e-6

Table 6: nor (foldr λ_1(head L) L)(foldr λ_2 (head L) (tail L))

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r0	r6	1/4.8	1/846	256/7603	8.29e-6
r4	r6	1/32	1/846	256/7603	1.24e-6

Table 7: nor (foldr λ_1(head L)(tail L))(foldr λ_2 (head L) L)

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r6	r0	1/846	1/4.8	256/7603	8.29e-6
r6	r4	1/846	1/32	256/7603	1.24e-6

Table 8: nor (foldrλ_1(head L)(tail L))(foldrλ_2 (head L)(tail L))

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r0	r6	1/4.8	1/846	256/7603	8.29e-6
r4	r6	1/32	1/846	256/7603	1.24e-6
r6	r6	1/846	1/846	256/7603	4.70e-8
r6	r4	1/846	1/32	256/7603	1.24e-6
r6	r0	1/846	1/4.8	256/7603	8.29e-6

Table 9: foldr λ_1(foldr λ_2 (head L)(tail L))(tail(tail L))

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r3	r9	1/32	1/821	256/7603	1.28e-6

Table 10: foldr λ_1(foldr λ_2 (head L)(tail L)) L

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r3	r9	1/32	1/821	256/7603	1.28e-6
r6	r15	1/846	1/4.8	256/7603	8.29e-6

Table 11: foldr λ_1(foldr λ_2 (head L) L)(tail L)

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r6	r5	1/846	1/7.8	256/7603	5.10e-6
r9	r3	1/821	1/32	256/7603	1.28e-6

Table 12: foldr λ_1(foldr λ_2 (head L) L) L

λ_1	λ_2	P(λ_1)	P(λ_2)	P(str)	P(s)
r6	r15	1/846	1/4.8	256/7603	8.29e-6
r6	r13	1/846	1/31.9	256/7603	1.25e-6

Table 13: foldr λ_1(fun (head L)(head L))(tail L)

λ_1	fun	P(λ_1)	P(str)	P(s)
r6	nand	1/846	16/7603	2.48e-6
r6	nor	1/846	16/7603	2.48e-6

The probability to find a solution to the general even-parity problem is the summation of the probability to find each of the 29 solutions:

$$P(S) = \sum_{i=1}^{29} P(s_i) = 0.000111$$

Using the probability of finding a solution, the number of program evaluations required to find a solution can be estimated. This is the “Effort” defined in [Koza, 1992, page 194].

$$E = \frac{\log(1-z)}{\log(1-P)} \text{, where z = 99\%} \qquad \textit{(EQ 1)}$$

$$E \approx -\frac{\log(1-z)}{P} \approx 18,000$$

The results of our analysis have two important implications:

- Structure abstraction enables the general even-parity problem to be solved very efficiently. Related work using different techniques requires the evaluation of a much bigger number of programs before a solution can be found [Koza, 1992; Chellapilla, 1997; Poli, Page and Langdon, 1999]. Moreover, their solutions only work for a particular value of *N* and are not general solutions.
- Most of the effort used to find a solution is devoted to the search of the correct Boolean rules. This is based on the observation that *P*(*str*) is insignificant compared to $P(\lambda_1) \cdot P(\lambda_2)$, hence $P(s) \approx P(\lambda_1) \cdot P(\lambda_2)$. This means that the complexity of the even-parity problem is approximately the same as that of the Boolean functions with two arguments problem.

6 Experiments and Results

To verify our analysis, we conduct experiments with the following implementations:

- The function and terminal sets are as shown in Table 1.
- The test cases and fitness evaluation are as described in Section 4.1.
- Fitness-proportionate selection is used to choose parents for reproduction.
- The population size is 500; maximum number of generation is 50; crossover rate is 100%. The tree depth limit is 4 for the main program and is 5 for the subtrees representing λ abstractions.

Figure 8 shows the performance curves based on 50 runs. All the runs find perfect solutions. The probability of success curve, P(M,i), reaches 100% at generation 26. The "effort" curve, I(M,i,z), gives the number of program evaluations required at each generation to find a solution. It is calculated using EQ 1. According to the experiment results, a solution to the general even-parity problem can be found by evaluating 15,500 programs which are randomly generated at generation 0. This is very close to our estimate of 18,000 program evaluations. Consequently, the 2 implications of our analysis are asserted.

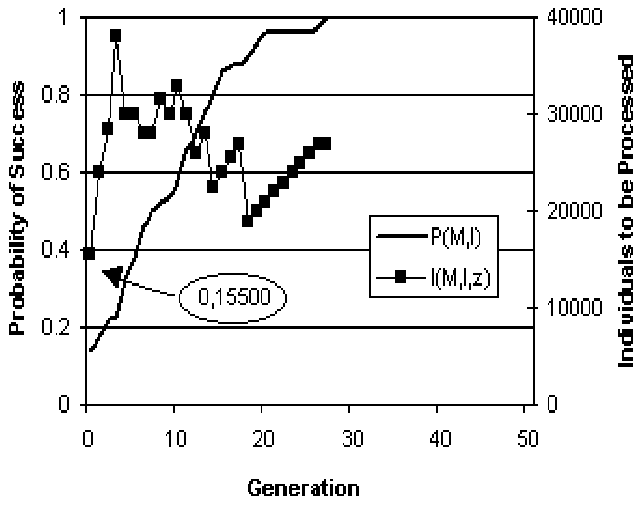

Figure 8. Performance curves for general even-parity problem.

7 Analysis and Discussion

With a program representation which supports structure abstraction, the general even-parity problem can be solved by random generation of 15,500 program trees. This result raises two important questions:

- Why structure abstraction makes the general even-parity an easy problem?
- Why random search outperforms GP search on this problem?

We address these two questions in the following subsections.

7.1 Impacts of Structure Abstraction

Structure abstraction has enabled the general even-parity problem to be solved very efficiently. Yet, there are only 29 solutions in the search space of 7,603 program structures (the density of the solution is 0.38%). This proves that the difficulty of a problem is independent of the density of the solution in the search space. Instead, it rely on how easy these solutions can be found. Our analysis indicates that the effort required to find a solution to the problem is approximate the same as that to find the Boolean rules partial solutions. This means that the module mechanism of λ abstraction, which allows partial solution to be evolved, is very important to the search of the solution. However, provided with partial solutions alone, [Langdon and Poli, 1998b] has shown that the generation of the overall solution is still not able to be achieved. An additional ingredient is the method to manipulate the partial solutions. The structure abstraction supported by `foldr` provides both ingredients, hence enables the solutions to be found easily:

- The bottom level of the structure abstraction hierarchy (λ abstraction) supports the evolution of partial solution (Boolean rules).
- The top level of the structure abstraction hierarchy (`foldr` higher-order function) provides mechanism to manipulate the partial solutions (reuse the Boolean rules) and the specification of the inputs order (as a list).

In other words, structure abstraction, provides a mechanism of hierarchical processing in problem solving: partial solutions are evolved and assembled to form a bigger solution. This hierarchical processing is shown to be very effective in solving the general even-parity problem.

7.2 Random Search Vs. GP Search

The fitness distribution (Figure 7) shows that program structures with fitness 6 occupies more than 60% of the search space while program structures with other fitness are sparse. This means that the search space does not contain gradient information, which is of great importance for progressive search algorithms, such as GP, to work. Consequently, GP is not able to outperform random search on this problem.

However, structure abstraction does not always create this kind of search space. With different problems, we anticipate that structure abstraction will generate search space that allows GP to shine.

8 Guidelines to Apply Structure Abstraction

Although an engine of hierarchical processing, structure abstraction must be applied properly in order to receive the benefits. This is the lesson learned from our experiences with the artificial ant problem. We have made the first-step to formulate guidelines for the application of structure abstraction to other problems:

Design a higher-order function which provides the following:

- function arguments to allow partial solutions to be evolved.
- methods to manipulate the partial solutions in constructing a bigger solution.
- specifications of the order of the inputs that partial solution can apply.

These guidelines will evolve as more experiences are gained.

9 Conclusion and Future Work

By selecting an appropriate program representation (higher-order function `foldr`, functions `head` & `tail` and terminal `L`), we present a successful example of using an evolutionary algorithm to solve a difficult problem. The important property of such a program representation is "structure abstraction", which can be obtained by including a higher-order function in the function set. When an appropriate higher-order function is selected, structure abstraction provides a mechanism of hierarchical processing in problem solving. Consequently, the effort required to find a solution is reduced. The general even-parity problem presented is an example of good use of structure abstraction. As there is a trend in developing problem-specific evolutionary algorithms [Leonhardi *et al.*, 1998], we provide guidelines for the application of structure abstraction to other problems.

We continue the design of a suitable higher-order function to provides structure abstraction for the artificial ant problem. We are also investigating structure abstraction with multiple-layer-hierarchy. It is hoped that this would provide hierarchical processing for program evolution even farther.

Acknowledgments

I like to thank Bill Langdon for his suggestions and comments on the paper.

References

[Angeline, 1994] P. J. Angeline. Genetic programming and emergent intelligence. *Advances in Genetic Programming*, K. E. Kinnear, Jr. (ed.), MIT Press, pp. 75-97, 1994.

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

[Haynes, 1998] T. Haynes. Perturbing the representation, recoding, and evaluation of chromosomes. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), Morgan Kaufmann, pp. 122-127, 1998.

[Koza, 1992] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[Koza, 1994] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, MA, 1994.

[Langdon and Poli, 1998a] W. B. Langdon and R. Poli. Why ants are hard. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), Morgan Kaufmann, pp. 193-201, 1998.

[Langdon and Poli, 1998b] W. B. Langdon and R. Poli. Why "building blocks" don't work on parity problems. Technical Report CSRP-98-17. The University of Birmingham, U.K. 1998.

[Leonhardi *et al.*, 1998] A. Leonhardi, W. Reissenberger, T. Schmelmer, K. Weicker and N. Weicker. Development of problem-specific evolutionary algorithms. *Fifth International Conference on Parallel Problem Solving from Nature*. A. E. Eiben, T. Bäck, M. Schoenauer and H.-P. Schwefel (eds.). Springer. pp. 388-397, 1998.

[Poli, Page and Langdon, 1999] R. Poli, J. Page and W. B. Langdon. Solving even-12, -13, -15, -17, -20 and -22 boolean parity problems using sub-machine code GP with smooth uniform crossover, smooth point mutation and demes. Technical report CSRP-99-2. The University of Birmingham, U.K. 1999.

[Rosca and Ballard, 1996] J. P. Rosca and D. H. Ballard. Discovery of subroutines in genetic programming. *Advances in Genetic Programming II*, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press, Cambridge, MA, pp.177-201, 1996.

[Spector, 1996] L. Spector. Simultaneous evolution of programs and their control structures. *Advances in Genetic Programming II*, P. J. Angeline and K. E. Kinnear, Jr. (eds.), MIT Press, Cambridge, MA, pp.137-154, 1996.

[Wolpert and Macready, 1997] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67-82, April 1997.

[Yu and Clack, 1998a] T. Yu and C. Clack. PolyGP: a polymorphic genetic programming system in haskell. *Genetic Programming 1998: Proceedings of the Third Annual Conference.* J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), Morgan Kaufmann, pp. 416-421, 1998.

[Yu and Clack, 1998b] T. Yu and C. Clack. Recursion, lambda abstractions and genetic programming. *Genetic Programming 1998: Proceedings of the Third Annual Conference*. J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba and R. Riolo (eds.), Morgan Kaufmann, pp. 422-431, 1998.