Addressing the Even-n-parity problem using Compressed Linear Genetic Programming

Johan Parent
Vrije Universiteit Brussel
Faculty of Engineering, ETRO
Pleinlaan 2
1050 Brussel, BELGIUM
johan@info.vub.ac.be

Ann Nowé
Vrije Universiteit Brussel
Faculty of Science, COMO
Pleinlaan 2
1050 Brussel, BELGIUM
asnowe@info.vub.ac.be

Anne Defaweux
Vrije Universiteit Brussel
Faculty of Science, COMO
Pleinlaan 2
1050 Brussel, BELGIUM
adefaweu@vub.ac.be

ABSTRACT

Compressed Linear Genetic Programming (cl-GP) uses substring compression as a modularization scheme. Despite the fact that the compression of substrings assumes a tight linkage between alleles, this approach improves the GP search process. The compression of the genotype, which is a form of linkage learning, provides both a protection mechanism and a form of genetic code reuse. This text presents the results obtained with the cl-GP on the Even-n-parity problem. Results indicate that the modularization of the cl-GP performs better than a normal l-GP as it allows the cl-GP to preserve useful gene combinations. Additionally the cl-GP modularization is well suited for problems where the problem size is adjusted in a co-evolutionary setup, the problem size increases each time a solution is found.

Categories and Subject Descriptors

[**Genetic Programming**]

Keywords

representation, linear GP, genotype, substitution

1. INTRODUCTION

Genetic algorithms (GA) and genetic programming (GP) search the space of possible solutions by manipulating the solution representation using genetic operators like crossover and mutation. Variable length representations allow the structure of the solution to evolve but are still restricted to the genetic primitives used to build the solutions. If GA and GP are to address more complex problems it becomes necessary to automatically adapt the representation to the problem. Modularization adapts the representation by extending the set of genetic primitives with problem specific functionalities. In this text we present a low level modularization technique for linear GP system based on compression. The presented compressed linear GP (cl-GP) attempts to identify usefull combinations of genes in the population and makes these available as new primitives. The search process benefit from these new functionalities to progress more quickly through the search space.

The assumption of tight linkage between individual genes (cfr substrings) was made with the use of a linear representation in mind. The representation of a linear GP system is similar to that used by a GA [4][11]. Following the approach used by [11] the cl-GP loop consists of a standard (generational) GA on top of which a problem specific evaluator is placed to simulate the program execution.

This paper is structured as follows Section 2 describes related work, In Section 3 the compression/substitution scheme is presented in detail. In Section 4 the experiments used to evaluate the impact different parameter settings of our algorithm is presented. Results are presented in Section 5.

2. RELATED WORK

The benefits of modularization for the GP search process are well known [9][1][7]. Modularization fosters code reuse on one hand, and on the other hand allows the GP system to identify and use high level functionalities. Different modularization strategies for tree based GP have been explored. Automatically Defined Functions (ADF) is the most prominent approach. It consists of a main tree which evolves together with a predefined number of additional trees. Those additional trees can be called from the main tree and, as such, complement the set of primitives available to the GP system. An alternative method is *encapsulation*. It replaces a subtree by a new symbol, this symbol is added to the primitive set of the system [6][9]. The symbol created in this way corresponds to a terminal/leaf node as it does not have any arguments. A third method, *module acquisition*, works in a similar fashion, but can create both function nodes (modules) and leaf nodes. If the depth of the selected subtree exceeds a certain threshold module acquisition removes the subtrees below this level. In this case a function node will be created by adding an argument for each removed subtree [2][3]. Although modularization has mainly been of interest to the GP community it is of course related to the search of building blocks in a GA. In [5] a *module acquisition algorithm* is presented which exploits modularity and hierarchy. Despite the use of a variable length representation, the problem of modularity is approached purely from a GA point of

```
0.  Choose initial population
1.  Evaluate each individual's fitness
2.  Repeat
3.      Compress individuals
4.      Select best-ranking individuals to reproduce
5.      Mate pairs at random
6.      Apply crossover operator
7.      Apply mutation operator
8.      Decompress individuals
9.      Evaluate each individual's fitness
10.  Until terminating condition
```

Figure 1: The pseudo code of a standard GA with 2 extra steps. Step 3 adds the compression of the individual prior to the creation of the next generation. Step 8 decompresses the individuals so that their fitness can be evaluated.

view. A *module* is defined as a combination of gene values that maximize the fitness. For example, if for the genes g_1 and g_2 the combination $g_1 = v_8$ and $g_2 = v_3$ dominates all other combinations of genes value, then it is considered a module. As in the work of [5] our algorithm relies on a substitution scheme but with notable differences in the substitution strategy. Another difference is that the compressed genotype scheme of the cl-GP was imagined to be used in a linear GP system. The linear encoding of a GP program exhibits a much tighter linkage than is the case for standard GA problems. This assumption is important as it justifies the use of the simple compression scheme which supposes a strong dependency between adjacent values.

3. COMPRESSED LINEAR GP

Our cl-GP algorithm consist of the compressed genetic algorithm (cGA) together with problem specific evaluator. This section presents the cGA which consists of a GA extended with a substitution/compression based modularization mechanism. Figure 1 contains the pseudo code for cGA loop. The cGA adds compression and decompression steps in the GA loop. Compression is applied to the genotype of individuals in the population prior to the creation of the next generation. As a result the search process occurs using a compressed representation. After the creation of the next generation the individuals are decompressed so that their fitness can be evaluated.

The schema theory models the probability of disruption by crossover is proportional to the defining length of the building block. Compression can shorten the representation of the building blocks thus reduce the chance of disruption by crossover. The cGA uses a substitution coder to compress the genotype in an attempt to protect substrings that represent building blocks. Protecting the building blocks guarantees the preservation of good allele combinations and is beneficial to the search process. In Subsection 3.1 the basics of the substitution coder used by the cGA are presented. The compression of the individuals is a two stage stochastic process: build a dictionary and select individuals for compression. Subsection 3.2 explains how the dictionary used for the compression is built. Subsection 3.4 describes how compression is applied to the population of the cGA.

```
0. D = build_dictionary();
1. current = in; /* in input string */
2. out = ""; /* Empty output */
3. For e in D /* Loop A */
4.    For pos = 1 to |in| /* Loop B */
5.       If match(pos, current, e.str) Then
6.          out[pos] = current[pos];
7.       Else
8.          out[pos] = e.ref;
9.          pos = pos + |e.str|;
10.       End;
11.   End
12.   current = out;
13.End
```

Figure 2: The pseudo code for the substitution step of the coder used in the cGA. Given are the dictionary $\mathcal{D}$ and the input string in. The coder searches for a match for each dictionary entry (loop A). If a match is found the reference is placed in the output, otherwise the original symbol. The inner loop (loop B) can be replaced by the more efficient KMP string matching algorithm with a linear runtime complexity.

3.1 Substitution coder

A substitution coder is a lossless[1] compression algorithm. Compression is obtained by replacing substrings in the input by a shorter reference. A substitution coder uses a *dictionary* which contains the substrings that need to be substituted and associates a placeholder symbol with each entry. Compression involves two phases: 1) building a dictionary and 2) performing the substitutions. Figure 2 represents the pseudo code of a substitution coder. The content of the dictionary is critical for the compression performance. Much of the research in data compression concerns the development of algorithms to build the dictionary. Section 3.2 describes how the dictionary is built in the cGA. The substitution step is computationally expensive as it involves searching for a match for each dictionary entry in the input string. If a match for substring s is found its placeholder symbol is placed in the output instead of the original symbols. The decompression step involves the replacing the placeholder symbols by the original strings. Decompression is much faster since it does not involve any search.

Suppose a dictionary $\mathcal{D} = \{$"101" $\rightarrow \alpha$,"00" $\rightarrow \beta$,"11" $\rightarrow \gamma\}$ then a substitution coder would compress the individual "1010001011101" to "$\alpha\beta0\alpha1\alpha$". One can observe that the order in which the dictionary entries are ordered is important. If the opposite order had been used the result would have been $\{\alpha\beta010\gamma\alpha\}$.

3.2 Building the dictionary

This section explains the stochastic algorithm used in the cGA to build the dictionary as it differs from the algorithm(s) used for *pure* data compression applications. We considered other criteria to build the dictionary since reducing the memory needed to represent the individuals is not our goal. We seek to build a dictionary containing building

[1]Lossless compression means that the original data is restored after compression.

blocks. The problem of identifying building blocks is recurrent for all the modularization algorithms. In [5] the identification of a set of alleles as a module involves additional fitness evaluations. These are used to determine whether an alternative substring with a better fitness exists. If no such string can be found then the substring will be used as a module. Another approach, used by [10], uses a separate *block fitness function* to evaluate the merits of a subpart of a genetic program. This function is presented as scaled down version of the fitness function for a smaller version of the original problem. The strategy adopted in this work is to see a GA as the building block discovery process. Since by definition building blocks are contributing in a positive way to the fitness of an individual, they are bound to be present in the better individuals of a population. We believe that this approach is more general as it 1) relies on the information already present in the population, 2) avoids additional computations and 3) does not require additional fitness functions to be engineered. Especially the latter two unacceptable when it comes to real world applications.

The cGA builds its dictionary in two stages. First, a pool of M individuals is selected stochastically from the population. The genotype of these individual will be used to create the dictionary in the second stage. These individuals are selected using fitness proportional selection. The pool consist mainly of above average individuals, yet with a minimum of diversity. Once the pool has been created every substring of length l of each individual is added to the dictionary. The dictionary only contains strings of the same length[2]. For example, suppose a non-binary alphabet $\{a, b, c, d\}$ and the substring length l equal to 3. In this case the individual "$abccdabc$" would add the substring (and their respective placeholder) "abc" $\rightarrow \alpha$, "bcc" $\rightarrow \beta$, "ccd" $\rightarrow \gamma$, "cda" $\rightarrow \delta$ and "dab" $\rightarrow \sigma$ to the dictionary. The cGA dictionary contains next to the substrings and their placeholder symbol a counter of the occurrence of every substring in the pool. This counter is used to order entries of the dictionary. We have chosen to sort the dictionary entries based on the occurrence of each substring, the most frequently occurring substring(s) will be substituted first.

The cGA rebuilds the dictionary for each generation. This allows to update the content of the dictionary with information that reflects the evolution of the population. This differs from [5] where the *module formation algorithm* is applied periodically. This also excludes the presence of dictionary entries containing references to other entries.

3.3 Modularization

Applying compression to the genotype can protect building block. But it does not by itself provide modularization comparable to that of ADFs, encapsulation or module acquisition. Modularization implies that a somehow code reuse should be present. Code reuse is achieved by the cGA by adhering to several conditions. First of all, the representation used by the cGA is adapted during the search. A new symbol is added to the alphabet of the genetic system for each potential building block identified by the cGA. Second, the genetic operator must be unrestricted: no distinction is made between compressed symbols and symbols of the original alphabet. Mutation for instance can replace a compressed symbol by an *original* symbol and vice versa. The third condition is that no distinction is made between individuals with and without compressed genotype. As will be explained in section 3.4, all the individuals are not necessarily subjected to compression. By not discriminating between original genes and *compressed genes*, the cGA can make full use of the genetic combinations that where identified as possible building blocks. It then becomes possible for the cGA to address problems where repetition is present. Crossover and mutation operators serve as natural mechanism to obtain this form of *translocation* of genetic information. Translocation occurs when a part of a chromosome is detached and reattached at a position different from the its original position in the chromosome.

[2]In traditional substitution coders the dictionary can contain entries of different lengths

3.4 Compressing the population

Once the dictionary has been built the substitution coder (Subsection 3.1) can be used to compress the individuals in the population. The cGA applies compression in a stochastic fashion to a part of the population. A fraction κ ($\in [0, 1]$) of the population will be compressed. This fraction κ of individuals are selected using tournament selection. Since the compression setup is repeated anew for each generation this means that the cGA does not systematically compress the same individuals. This makes it possible to explore the benefit of the different modules (dictionary entries) in different contexts (i.e.. allele combinations).

3.5 Lossless and stochastic

The cGA applies a deterministic transformation, compression, to some of the individuals in the population. Since the compression is lossless the (genetic) information present in the population of the cGA or a GA is exactly the same. Yet, as described above, a stochastic component has been added to the overall population compression process. The creation of the dictionary and the selection of the individuals for compression are non-deterministic processes. This stochastic component is meant to counter balance the effect of the substitutions on the search process. The protection against crossover provided by the compression has a negative impact on the population diversity. Several factors explain this phenomenon. First the search efficiency of the crossover operator is reduced by the compression of the genotype. Second, the compression of the individuals is detrimental for the sampling of schemata. A last reason is that the dictionary entries are not guaranteed to correspond to building blocks. This situation is especially exacerbated during the first generation as the population still needs to discover promising gene combinations. This was illustrated by experiments where the entire population underwent compression. These experiments have invariably lead to premature convergence and consequently suboptimal results. The parameter κ is a way to limit the decrease in the population diversity. The other non-deterministic steps were introduced for the same reason as they maintain a minimum of diversity at the substring level.

3.6 Further changes

Although the cGA is meant to be a minimal extension to the standard GA a few extra modification were required.

The biggest difference is an unavoidable transition to a variable length linear representation. This change is due to the compression scheme.

3.6.1 Variable length

Despite the use of a lossless compression algorithm and the fact that the cGA, like a standard GA, starts with a population of individuals of identical size, individuals of different sizes quickly emerge. This phenomenon, due to the use of compression, occurs through several mechanisms:

1. not all the individual are compressed
2. different individuals compress to different sizes
3. the genetic operators can create individuals of very different sizes

Since the cGA does not discriminate between placeholder symbols and *original* symbols, recombination and mutation can add or remove symbols representing a substring and vice versa. This can result in an important change in length. Depending on the situation individuals can exceed the initial individual size or be shorter. Consider the following example. Suppose the dictionary $\mathcal{D} = \{"xy" \rightarrow \alpha, "zz" \rightarrow \beta\}$ over the alphabet $\{x, y, z\}$. Mutating the second gene of the individual with genotype "xx" from x to α creates "$x\alpha$". The decompressed version of this individual's genotype is now "xxy". Similar scenarios exists for the crossover.

3.6.2 Recombination and mutation

As illustrated above the genetic operators can create individuals of very different sizes. As a result a maximum individual length has to be enforced to avoid bloat. Individuals created by crossover that exceed the maximum length are truncated. Yet in the case of both mutation and crossover one problem remains. As illustrated in the previous paragraph it is possible for an individual to exceed the maximum once it has been decompressed. In this case the cGA truncates the genotype after decompression. The mutation operator was modified to be able to cope with a variable alphabet sizes.

4. EXPERIMENT

The advantage of the modularization of the cl-GP, provided by the cGA, has been illustrated in different GP problems: symbolic regression, classification and a real world data application. This text presents the experiments using the Even-n-parity problem. This problem has been chosen since it is a standard GP problem. Furthermore the difficulty of this problem can be adjusted by modifying the number of input bits n. We performed two type of experiments.

The first experiments were intended to assess impact of the modularization of the cl-GP on the search process. The results of the l-GP and cl-GP are compared on the problem instances of different sizes. These experiments serve to illustrate the ability of the cl-GP to identify and reuse building block present in the population.

In a second set of experiments the size of the Even-n-parity problem was allowed to evolve. Each time a solution was found by the GP system the problem size was incremented making the problem increasingly more difficult. The approach inspired by co-evolutionary models provides a simple mechanism for increase the problem size. These experiments illustrate whether the cl-GP has to ability to generate increasingly complex solutions in a top-down fashion. Building blocks discovered while solving simpler problems can be reused to address larger problem.

4.1 Even-n-parity

The Even-n-Parity problem requires the correct classification of bit strings of length n having an even number of 1's. This classification is formulated as a boolean function returning the value *true* for an even number of 1's and *false* otherwise. The terminals and function set are $T = \{b_0, b_1, \cdots, b_n\}$ and $F = \{NOOP, AND, OR, NAND, NOR\}$. The NOOP instruction is not executed during the evolution. This instruction allows to represent programmas of variable length using a fixed length representation [11]. We believe the inclusion of this instruction is necessary in order to compare the l-GP, driven by a GA (fixed length), and the cl-GP driven by the cGA (variable length). The raw fitness (f_{raw}) is the ratio of correct classifications over the entire data set: $f_{raw} = 1 - \frac{\#errors}{2^n}$. The standard fitness is equal to the raw fitness for this problem. The evaluator used for this problem is described in the next section.

4.1.1 Boolean evaluator

The Even-n-parity individuals are postfix encoded and evaluated by a stack based virtual machine as described in [8]. Its instruction set provides the four boolean operators[3] and five terminals. The operations take their 2 operands from the stack and push their result onto it. The terminal instructions correspond to a push of the individual input bits on the stack. As for the numerical evaluator the result of the program is the value of the top of the stack after evaluation. The individual [b_0, NOR, NAND, b_0, b_4, OR, AND, b_2, OR, b_2, NOOP, NOR, NAND] corresponds to the boolean expression (b_0 OR b_4) NAND (b_2 NOR b_2).

5. RESULTS

All the results presented here are the average of 100 independent runs, using randomly seeded initial populations of 500 individuals. The genotype length is 32. Other settings were: crossover rate 80%, mutation rate 5% and the top 5% of the population was kept at every generation. Individuals were selection with tournament selection (size=2). Every experiment lasted 50 generations if not mentioned otherwise.

The following values were used for the runs using the cl-GP: $\kappa = 0.33$ (tournament size 4) and the length of the dictionary entries equals two. A pool of 10 individuals (fitness proportional selected) is used to build the dictionary.

5.1 cl-GP vs l-GP

This subsection compares the results for different Even-n-parity problem instances. The problem size n is fix during each experiment. Table 1 compares the fitness of the best of the population and the number of successful runs, between brackets, after 200 generations for different problem

[3]Following the observations of [8] the lazy version of the operators was implemented.

Problem size n	l-GP	cl-GP
5	0.905 (31)	0.996 (93)
6	0.818 (19)	0.963 (60)
7	0.724 (10)	0.962 (56)
8	0.666 (2)	0.926 (43)
9	0.598 (0)	0.758 (16)

Table 1: The raw fitness of the best of population and the number of successful runs for different problem sizes.

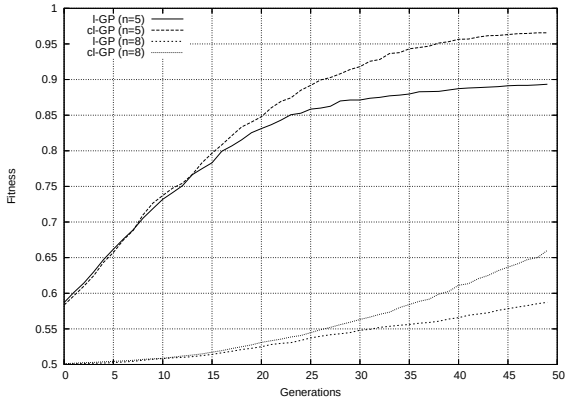

Figure 3: An example run for the Even-5-parity and Even-8-parity problem for population size of 500. The linear GP system driven by the cGA (cl-GP) outperforms the same system using the GA (l-GP).

instances. The use of substitution has a positive effect on the performance of the GP system. The unpaired Wilcoxon-Mann-Whitney test has been used to confirm that the difference in performance is indeed significant for all the presented problems.

Both the average score of the best of population and number of solved instances of cl-GP are higher than for the l-GP. This means that the cl-GP preforms better on average than the l-GP. But also that the cl-GP is able to find a solution more often than the l-GP. The linear GP driven by the GA (i.e. the l-GP) achieves a 31% probability of success after 200 generations. The same setup but using the cGA, the cl-GP, obtains a probability of success of 93%. For the Even-6-parity problem these numbers become 19% and 60% respectively.

Figure 3 illustrates the evolution the fitness of the best of population of both algorithms for the Even-5-Parity and Even-8-Parity problem. During the first 15 generations (on average) there is no significant difference in performance between the cl-GP and the l-GP. As the GP system discovers good combinations of genes the modularization of the cl-GP is enables it to reuse them. From that point on the cl-GP start to outperform the l-GP.

5.2 Co-evolving the problem size

In these experiments the problem size has been adjusted as soon as the GP system found a solution. Each experiment starts with a problem size of 3. Figure 4 illustrates the ability of the cl-GP to solve problems of increasing com-

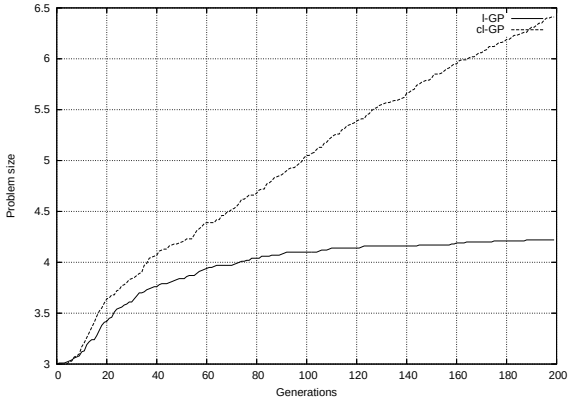

Figure 4: The evolution of the problem size starting from an initial problem size of 5. The cl-GP is able to solve problems of increasing size. The modularization allows gene combinations discovered while solving smaller problem instance to be reused.

cl-GP	F=3	4	5	6	7	8	9	10
I=3	1	8	11	33	32	14	1	0
5			10	20	28	22	2	0
7					24	23	11	3
9							16	0

Table 2: cl-GP: the number of runs that solved problems of size F after 200 generations as a function of the initial problem size I.

plexity. The modularization allows gene combinations discovered while solving smaller problem instances to be reused to solve larger problems. After 100 generations [4] the cl-GP successfully solves problems of size 5 and of size 6 after 160 generations. In a similar setup the l-GP manages to solve problems of size 4 but then stagnates.

To illustrate the influence of the initial problem size these experiments have been repeated, this time different initial problems sizes were compared. Tables 2 and 3 contain the number of runs that solved problem instances of size F (horizontally) after 200 generations starting from a problem instance of size I (vertically). One can see in table 2 that using 100 independent runs the cl-GP was able to solve 3 instances of the Even-10-parity when starting from a initial problem size of 7 (I=7, F=10).

[4]Averaged over 100 independent runs
[4]The size F corresponds to the **biggest** instance successfully solved.

l-GP	F=3	4	5	6	7	8	9	10
I=3	37	33	16	5	7	2	0	0
5			20	8	2	0	0	0
7					4	3	0	0
9							0	0

Table 3: l-GP: the number of runs that solved problems of size F after 200 generations as a function of the initial problem size I.

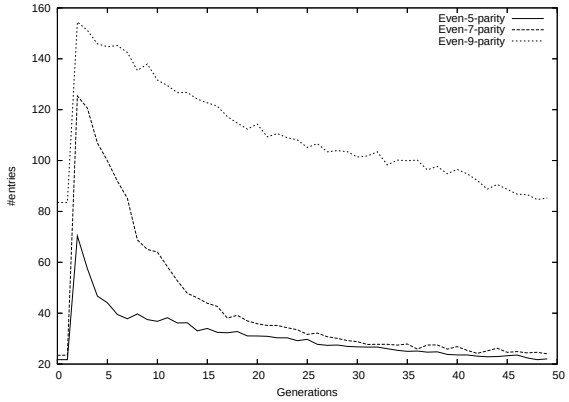

Figure 5: The evolution of the number of dictionary entries for different problem sizes.

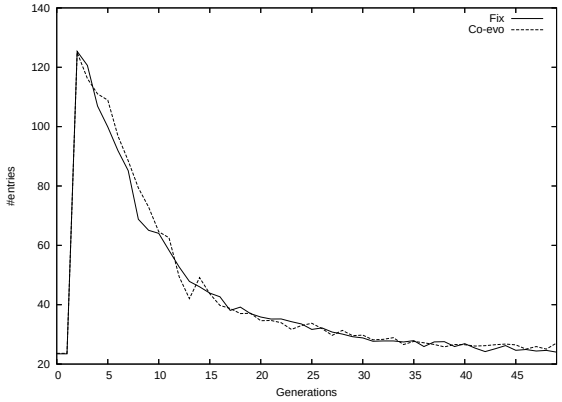

Figure 6: Comparison of the evolution of the number of dictionary entries for the fixed problem size and co-evolutionary approach (initial problem size 7).

Comparing the tables 2 and 3 with table 1 reveals that both the l-GP and the cl-GP benefit from the progressive increase of the problem difficulty. Both algorithms were able to find solutions for the Even-8-parity problem starting from smaller problems. The cl-GP benefits most from this approach since its modularization mechanism allows it to preserve building blocks. Building block discovered while solving smaller instances can be reused to addressed larger problems. In this setup the cl-GP was able to solve 61 (24+23+11+3) instances of size 7 or more. When keeping the problem size fixed the cl-GP only solved 56 instance of size 7. Using a larger initial, size 9, problem size does not allow the modularization of the cl-GP to be exploited (see section 6). For a problem size 9 the performance of th cl-GP is even lower than when using an initial size of 7.

5.3 Dictionary size of cl-GP

The size of the dictionary used by the cl-GP corresponds to the number of automatically created modules. Figure 5 depicts the evolution of the dictionary size for three problem instances for experiments using a *fixed* problem size. In the beginning of the search the dictionaries contain most entries. In the first generation the population still has to discover good genes combinations. Differences in the fitness between individuals are then mostly random. The individuals selected to form the pool used to build the dictionary at that point thus contains very different substrings. Each string will be added to the dictionary which explains why it is relatively large in the beginning of the search process. As the search progresses and the cl-GP discovers building blocks the number of entries starts to decrease quickly at first. At the end of the run the dictionary size(s) stabilize.

Interesting is the fact that for the smallest problem instances (size 5 and 7) the dictionary sizes at the end the run are approximately the same. This can indicate the presence of structure in the solutions of Even-n-parity problem. Building blocks used to solve the smaller instances can be reused to solve bigger ones. The evolution of the dictionary size for hardest problem instance (size 9) does not follow the same steep decrease as the other instances. This problem instance is more difficult to solve for both the l-GP and the cl-GP. The slow convergence to a good solutions makes it hard for the dictionary building algorithm of the cGA (used by the cl-GP) to identify good substrings.

Figure 6 compares the evolution of the dictionary size of the fixed and co-evolutionary problem instances of size 7. There is no significant difference for the two experiments despite the fact in the co-evolutionary setup the cl-GP is able to solve larger problem instances (up to Even-9-parity).

6. DISCUSSION

The cl-GP uses the cGA presented in this paper. The cGA provides a low level modularization mechanism based on compression. The benefit of the compression based modularization has been illustrated in the previous sections with two different experiments. It is important to point out that the presented results are averaged of 100 runs. The despite its generally better performance the cl-GP can fail to improve upon the l-GP. The analysis of such *unsuccessful* runs has revealed that in these cases the cl-GP did not identify any modules. This can be explained by the fact that the cGA, being an extension to the standard GA, capitalizes on the GA's ability to find good combinations. When the GA fails to identify building blocks it becomes impossible for the cGA to reuse these combinations to its advantage.

7. CONCLUSIONS

This paper explores the cl-GP algorithm in an co-evolutionary setup. The compression based modularization mechanism of the cl-GP implies a tight linkage between the genes in the representation which allows it to replace substrings in the genotype with a shorter reference. We performed an empirical study using several Even-n-parity problem instances in two different setups. In a first setup the problem size was fixed and was used to illustrate the advantage of the cl-GP over the l-GP which does not use modularization. The second co-evolutionary setup allowed the problem size to increase. In this setup the modularization of the cl-GP makes it possible to address larger problem instances by reusing previously acquired gene combinations. It is however important in this setting not to start the search using large problem instances as the cl-GP would then fail to discover any modules.

8. FUTURE WORK

Although substrings of length 2 seem to be best at this point, it may be interesting to allow this size to change during the course of evolution. One way to achieve this would be to allow recursive entries in the dictionary, i.e. entries containing references to other entries.

9. REFERENCES

[1] M. Ahluwalia and L. Bull. Coevolving functions in genetic programming. *Journal of Systems Architecture*, 47(7):573–585, July 2001.

[2] P. J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.

[3] P. J. Angeline and J. B. Pollack. Coevolving high-level representations. In C. G. Langton, editor, *Artificial Life III*, volume XVII of *SFI Studies in the Sciences of Complexity*, pages 55–71, Santa Fe, New Mexico, 15-19 June 1992 1994. Addison-Wesley.

[4] M. Brameier and W. Banzhaf. Effective linear genetic programming. Technical report, Department of Computer Science, University of Dortmund, 44221 Dortmund, Germany, 2001.

[5] E. D. de Jong and D. Thierens. Exploiting modularity, hierarchy, and repetition in variable-length problems. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Genetic and Evolutionary Computation – GECCO-2004, Part I*, volume 3102 of *Lecture Notes in Computer Science*, pages 1030–1041, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[6] D. Howard. Modularization by multi-run frequency driven subtree encapsulation. In R. L. Riolo and B. Worzel, editors, *Genetic Programming Theory and Practise*, chapter 10, pages 155–172. Kluwer, 2003.

[7] J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors. *Genetic Programming 1996: Proceedings of the First Annual Conference*, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[8] T. Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[9] S. C. Roberts, D. Howard, and J. R. Koza. Evolving modules in genetic programming by subtree encapsulation. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 160–175, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[10] J. P. Rosca and D. H. Ballard. Hierarchical self-organization in genetic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann, 1994.

[11] K. Stoffel and L. Spector. High-performance, parallel, stack-based genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 224–229, Stanford University, CA, USA, 28–31 July 1996. MIT Press.