Evolving Black-Box Search Algorithms Employing Genetic Programming

Matthew A. Martin
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, U.S.A.
mam446@mst.edu

Daniel R. Tauritz
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, U.S.A.
dtauritz@acm.org

ABSTRACT

Restricting the class of problems we want to perform well on allows Black Box Search Algorithms (BBSAs) specifically tailored to that class to significantly outperform more general purpose problem solvers. However, the fields that encompass BBSAs, including Evolutionary Computing, are mostly focused on improving algorithm performance over increasingly diversified problem classes. By definition, the payoff for designing a high quality general purpose solver is far larger in terms of the number of problems it can address, than a specialized BBSA. This paper introduces a novel approach to creating tailored BBSAs through automated design employing genetic programming. An experiment is reported which demonstrates its ability to create novel BBSAs which outperform established BBSAs including canonical evolutionary algorithms.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search; I.2.2 [**Artificial Intelligence**]: Automatic Programming–*program modification, program synthesis*

General Terms

Algorithms, Design

Keywords

Black-Box Search Algorithms, Evolutionary Algorithms, Genetic Programming

1. INTRODUCTION

An interpretation of the No Free Lunch (NFL) Theorem is that all non-repeating Black Box Search Algorithms (BBSAs) have the same average performance over all optimization problems [15]. This dooms the quest for a BBSA superior to all other BBSAs on all problems. However, restricting the class of problems we want to perform well on allows BBSAs specifically tailored to that class to significantly outperform more general purpose problem solvers. In contrast, the fields that encompass BBSAs, including Evolutionary Computing, are mostly focused on improving algorithm performance over increasingly diversified problem classes. By definition, the payoff for designing a high quality general purpose solver is far larger in terms of the number of problems it can address, than a specialized BBSA.

This paper introduces a novel approach to creating BBSAs through automated design employing genetic programming. It furthermore demonstrates that there are problem classes for which this approach generates BBSAs which significantly outperform established BBSAs including canonical Evolutionary Algorithms (EAs). While there have been previous attempts to automate the design of algorithms in terms of evolving operators and automating the selection of predefined operators, this work makes the next logical step and automates the design of algorithm *structure*. The proof of concept presented in this paper employs a limited set of relatively complex primitives extracted from existing canonical BBSAs for which experimental results are presented on the classic Deceptive Trap problem and compared to the performance of a steepest-ascent hill-climber and a canonical EA. A few selected evolved BBSAs demonstrating the abilities and drawbacks of this method are presented and analyzed.

2. RELATED WORK

Most previous work on employing evolutionary computing to create improved BBSAs, focused on tuning parameters [13] or adaptively selecting which of a pre-defined set of operators to use and in which order [12]. The latter employed Multi Expression Programming to evolve how, and in what order, the EA used selection, mutation, and recombination. This approach used four high level operations: Initialize, Select, Crossover, and Mutate. These operations were combined in various ways to evolve a better performing EA. Later this approach was also attempted employing Linear Genetic Programming (LGP) [4, 5, 11]. While this allowed the EA to identify the best combination of available selection, recombination, and mutation operators to use for a given problem, it was limited to a predefined structure.

A more recent approach to evolve BBSAs employed Grammatical Evolution (GE) [10] which uses a grammar to describe structure, but highly constrained to the standard EA model. No significant increase in result quality was reported.

GECCO'13 Companion, July 6–10, 2013, Amsterdam, The Netherlands.

Genetic Programming (GP) introduced the concept of evolving executable programs [9]. The first attempts at applying GP to the generation of BBSAs was to evolve individual EA operators. The primary effort has been to create improved EA variation operators [1, 6, 8, 17]. Some limited work has been done on evolving EA selection operators [14, 16]. Thus far the focus has been on evolving EA operators, rather than entire BBSAs of indiscrimate type. This paper takes the next logical step, namely evolving the *structure* of BBSAs to create novel and unexpected types of BBSAs.

3. METHODOLOGY

The specific focus of the research reported in this paper is to evolve BBSAs tailored to a specific problem class which can significantly outperform more general purpose BBSAs. GP was employed where fitness was based on the performance of an evolved BBSA with efficiency as tie-breaker.

3.1 Parse Tree

Instead of representing the entirety of an algorithm within a parse tree, the representation is a single iteration of a BBSA. A parse tree is used to represent the iteration for the evolutionary process such that standard GP operators will work effectively. The parse tree is evaluated in a pre-order fashion. Each non-terminal node will take one or more sets of solutions (including the empty set or a singleton set) from its child node(s), perform an operation on the set(s) and then return a set of solutions. The set that the root node returns will be stored as the 'Last' set which can be accessed in future iterations to facilitate population-based BBSAs. An example of a randomly generated BBSA represented as a parse tree is shown in Figure 1.

The terminal nodes are sets of solutions. These sets include the 'Last' set, as well as auxiliary sets which will be explained in Section 3.1.4. The non-terminal nodes that compose these trees are operations extracted from pre-existing algorithms. The nodes are broken down into selection nodes, variation nodes, set operation nodes, and other utility nodes. The following subsections describe the node type instances employed in the experiments reported in this paper.

3.1.1 Selection Operation Nodes

Two principal selection operations were employed in the experiments. The first of these is k-tournament selection with replacement. This node has two parameters, namely k and the number of solutions selected, the second is *count* which designates the number of solutions passed to the next node. The second selection operation employed is truncation selection. This operator takes the n best solutions from the set passed in, n being one of its parameters.

3.1.2 Variation Operation Nodes

For the experiments, four variation operations were used. The first variation operation is the standard binary uniform crossover for multiple parents. This variation operation returns n solutions, n being a parameter of the node. The second and third variation operations are the standard bit-flip mutation. The only difference between these two operations is that one creates a copy of each solution and then applies the mutation, while the other alters the solutions that were passed in. The last variation operator is diagonal crossover with multiple parents [7] which returns the same number of solutions as were passed in. This variation node has one pa-

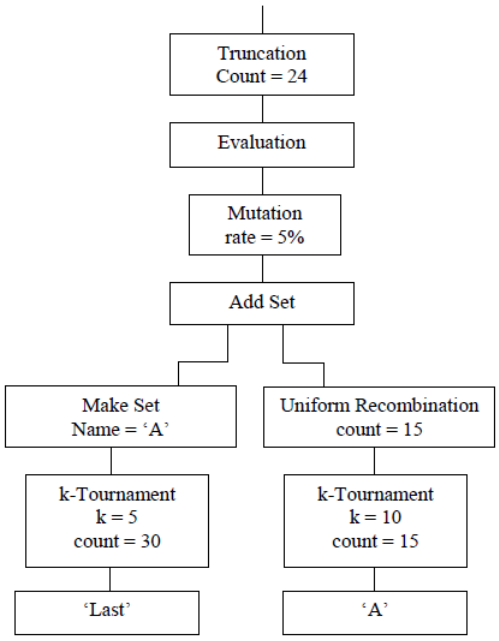

Figure 1: Example Parse Tree

```
initialize population
evaluate initial population
A = [ ]
while termination condition not met do
    X = kTournament(Last, k = 5,count =30)
    A = X
    Y = kTournament(A,k = 10, count = 15)
    Y = uniformRecombination(Y, count = 15)
    Z = X+Y
    Z = mutate(Z, rate = 5%)
    evaluate(Z)
    Last = truncate(Z, 24)
end while
evaluate(Last)
```

Figure 2: Example Parse Tree Generated Code

rameter, n, which determines the number of points employed by crossover.

All the experiments reported in this paper are on binary problems, thus the use of binary variation operators. However, this is not a general restriction and representation appropriate variation operators may be employed.

3.1.3 Set Operation Nodes

The experiments reported in this paper employed two distinct set operations. The first is the union operation named "Add Set". This node takes two sets of solutions and returns the union of the sets passed into it. The other operation is the save operation called "Make Set". This operation saves a copy of the set passed into it. This set can be used elsewhere in the algorithm as explained in Section 3.1.4.

3.1.4 Other Nodes

The last type of non-terminal node employed in this paper is the evaluation node. This node evaluates all of the solutions that are passed into it. Another option considered was, instead of having an evaluation node, to evaluate the solutions that were returned from the root node. This option was not selected to allow for more freedom in the structure of the algorithm.

The terminal nodes in this representation were sets of solutions. These sets could either be the 'Last' set returned by the previous iteration or a set that was created by the save operation. These saved sets persist from iteration to iteration such that if a set is referenced before it has been saved in a given iteration, it will use the save from the last iteration. At the beginning of each run, these sets are set to the empty set.

3.2 Meta-Algorithm

A customized GP was employed to meta-evolve the BBSAs. The two primary variation operators employed were the standard sub-tree crossover and sub-tree mutation. An alteration to the standard sub-tree mutation was made. The maximum number of nodes being added in this mutation is from 1 to a user defined value. Another mutation operation was added to this algorithm that selects a random node from the parse-tree and randomizes the parameters if it has any. To ensure that the GP has a good initial population, when creating the initial population each BBSA must have a non-zero fitness value. This discards the BBSAs that do not evaluate any solutions that they are given.

3.2.1 Black-Box Search Algorithm

Each individual in the GP's population is a BBSA. To evaluate the fitness of an individual, its encoded BBSA is run for a user-defined number of times. Each run of the BBSA begins with the population initialization and the evaluation of the initial population. The size of the population is evolved along with the structure of the algorithm. Then the parse-tree is evaluated until one of the termination criteria are met. Once a run of the BBSA is completed, the 'Last' set is evaluated to ensure that the final fitness value is representative of the final population. Logging is performed during these runs to track when the BBSA converged and what the converged solution quality is.

The fitness of a BBSA is primarily determined by the fitness function that it employs to evaluate the solutions it evolves. In addition to this, parsimony pressure is added to ensure that the parse trees do not get too large. The parsimony pressure is calculated by multiplying the number of nodes in a tree by a user defined value. The parsimony pressure is subtracted from the best solution in the final population averaged over all runs to get the fitness of the BBSA. When comparing two BBSAs, in case of equal fitness, convergence time is employed as a tie-breaker.

The evaluation of the BBSAs is the computational bottleneck for this approach. Thus, to minimize time wasted on poor solutions, a partial evaluation is supported to allow terminating poor solutions before they are fully evaluated. This is accomplished by applying four limiting factors. First of all, there is a maximum number of evaluations that a BBSA may perform during each run. If a BBSA exceeds that number, then it will automatically terminate mid-run. Secondly, there is a maximum number of iterations that the BBSA may perform before it will halt. This addition of an iteration limit adds pressure to the GP to evolve algorithms with more evaluations per iteration. If this iteration limit were not put in place, it would take BBSAs with very low evaluations per iteration much longer to be evaluated. Thirdly, the algorithm counts the relative number of operations performed. Each node represents an operation, and these operations can take a significant amount of time to execute. A weight is associated with each node that represents an estimation of how many operations that node takes per input solution. Once a node is executed, that weight is added to a running total of the operations for that run. Once the limit is reached, the run will end. This is to ensure that algorithms whose computational time complexity is dominated by operations rather than evaluations are terminated based on those operations. The fourth method is by convergence. If an algorithm has not improved in i iterations, then the run will end. If the operation limit or the evaluation limit are reached mid-way through an iteration, then the rest of that iteration is not run.

3.2.2 External Verification

To ensure that the performance of the evolved BBSA is accurate, code is generated to represent the parse tree. This is done to externally verify that the performance that the GP shows for a given BBSA is accurate when actually implemented. An example of a parse tree and the code generated is shown in Figure 1 and Figure 2. This verification was employed for the testing of the BBSAs in all experiments.

4. EXPERIMENTS

To demonstrate the proposed approach's ability to create novel, high-performance BBSAs, it was run on a selected problem class and compared with established BBSAs. For the selected problem class, the BBSAs are evolved with a given problem configuration. Once a BBSA has been evolved, it is run on different problem configurations to determine if the solution is a good solver for the generalized problem class. The evolved BBSAs are compared against a standard EA and a Steepest Ascent Hill-Climber. To ensure that the human bias of implementing an algorithm would not sway the results of the experiments, the EA was produced by the same external verification method described earlier. The EA was encoded with the parse tree shown in Figure 3 which generated the code shown in Figure 4. The parameter values for this algorithm were found using GP's Alternate Mutation for 2000 evaluations which is the

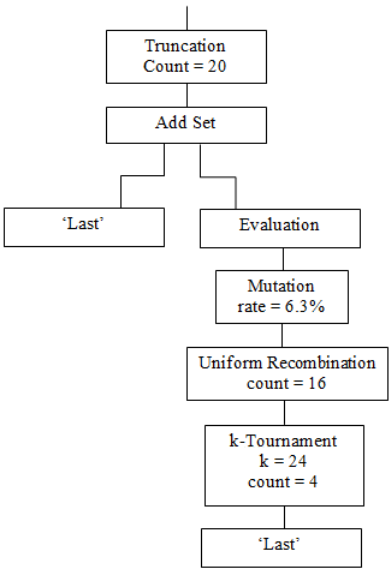

Figure 3: Example EA Parse Tree

initialize population
evaluate initial population
while termination condition not met **do**
  Y = kTournamentLast(k = 16, count = 4)
  Y = uniformRecombination(Y, count = 16)
  Z = mutate(Z, rate = 6.3%)
  evaluate(Z)
  Z = Z+Last
  Last = truncate(Z, 20)
end while

Figure 4: EA Parse Tree Generated Code

Bit-Length	Trap Size
100	5
200	5
105	7
210	7

Table 1: Problem Configurations for Deceptive Trap

maximum amount of parameter tuning that a BBSA could have during the experiments. The Hill-Climber could not be perfectly reproduced with the currently implemented nodes. Thus, this code had to be generated manually for the tests.

The classic Deceptive Trap problem [2] is employed as benchmark. It divides a bit-string into traps of size j bits each which are scored by using the following equation where t is equal to the sum of the bit values in the trap.

$$trap(t) = \begin{cases} j-1-t & (t < j) \\ j & (t = j) \end{cases}$$

The BBSAs were evolved with a bit-length of 100 and a trap size of 5. For the evolved BBSAs, code was generated using the external verification method described earlier. This generated code was run on the problem configurations shown in Table 1. This is done to determine if the evolved algorithm is a general solver for the problem class.

For these experiments, fifteen BBSAs were evolved. During the evolution process, each BBSA was run five times. The external verification method was used to generate code for data-gathering. Each of the evolved BBSAs was run 30 times for each of the problem configurations. Each of the algorithms was run for 50,000 evaluations. Then the results were compared with an EA and a Hill-Climber, each run 30 times with the same problem configurations.

Both the meta-GP and the BBSA generator have many parameters that can be tuned to optimize performance. The research reported in this paper utilized manual tuning; more rigorous tuning in the future may be expected to improve results. All of the experiments were conducted under the same settings. The GP was run for 2000 evaluations. The initial population was 50 individuals and each generation 20 new individuals are created. k-tournament selection with $k = 15$ was employed for parent selection. All of the recombination and mutation operations have an equal chance of being used. The parsimony pressure for the tree size was set to .001 per node. The maximum number of iterations the BBSAs can use is 500000 and the maximum number of iterations is 10000. All the parameter settings for the GP are summarized in Table 2.

For the generation of the BBSAs, heuristic constraints were employed to limit various parameters to reasonable values. The maximum number of individuals selected in selection nodes was set to 25. The maximum initial population was set to 50 individuals. The maximum k value used for k-tournament is 25. The maximum number of points for diagonal crossover is 10 points. All the parameter settings for the BBSA are summarized in Table 3.

A complete package of our data as well as the source code employed in testing is available from our website[1].

[1] `https://github.com/mam446/EvoBBSA`

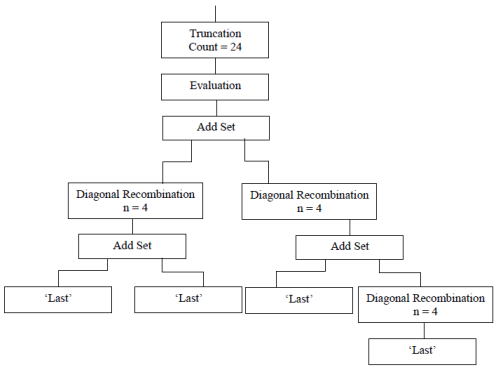

Figure 5: BBSA1 evolved for Deceptive Trap in parse tree form. Initial population of 49 solutions

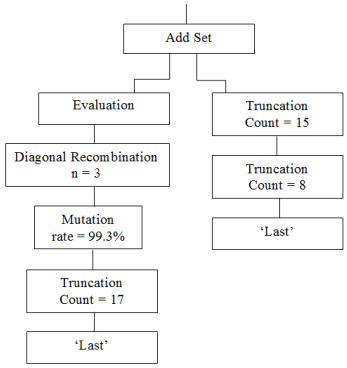

Figure 6: BBSA2 evolved for Deceptive Trap in parse tree form. Initial population of 29 solutions

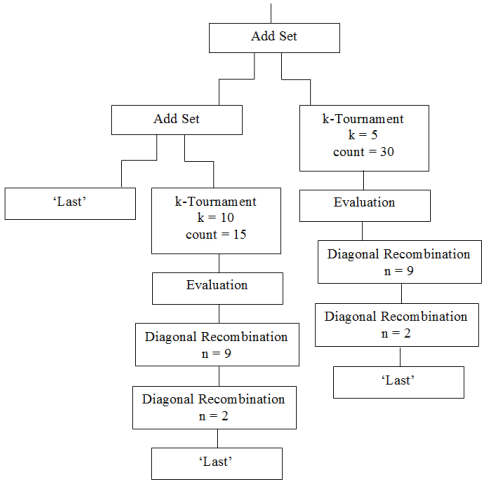

Figure 7: BBSA3 evolved for Deceptive Trap in parse tree form. Initial population of 39 solutions

Parameter	Value
Evaluations	2000
Initial Population	50
Children per Generation	20
k-Tournament	15
Sub-Tree Crossover Probability	33%
Sub-Tree Mutation Probability	33%
Alternate Mutation Probability	33%
Alternate Mutation Depth	5
Parsimony Pressure	0.001
Maximum Operations	500,000
Maximum Iterations	10,000

Table 2: GP Configurations

Parameter	Value
Evolution Runs	5
Evaluations	50,000
Maximum k Value	25
Maximum Number of Selected Individuals	25
Maximum Initial Population	50
Maximum Crossover Points	10

Table 3: Black-Box Search Algorithm Settings

5. RESULTS

Three algorithms were selected from the fifteen evolved algorithms to discuss in more detail. They were selected because they have features that help characterize the strengths and weaknesses of the proposed approach. The three algorithms have a very different structure from each other and versus existing canonical BBSAs. The structure of the algorithms is presented in figures 5-7. The algorithms are labeled BBSA1, BBSA2, and BBSA3, respectively.

Comparisons of the evolved BBSAs and the EA can be seen in figures 8-11. These graphs are the averages of the 30 runs that were performed for the statistical tests. For all of these tests the algorithms were evolved on the problem configuration of a bit-length of 100 and a trap size of 5, and were run on the problem configurations shown in Table 1.

A summary of the final states of the various BBSAs can be found in Table 4 which shows the results for each BBSA and problem configuration combinations averaged over all runs along with the standard deviation.

To determine statistically if the evolved BBSAs performed better than the EA and the Hill-Climber, a two-tailed t-test was used. The results of these tests are presented in tables 5-10. In the results column of these tables, a + indicates that the BBSA performed better than the EA/Hill-Climber. A - indicates that the BBSA performed worse than the EA/Hill-Climber. A $\sim$ indicates that there is no statistical difference between the algorithms. A summary of the t-test run on all of the BBSAs can be found in Table 11.

6. DISCUSSION

On the problem configuration for which the BBSAs were evolved, the quality of the solutions found was better than the EA and the Hill-Climber. However, on the other problem configurations the results were generally not as good.

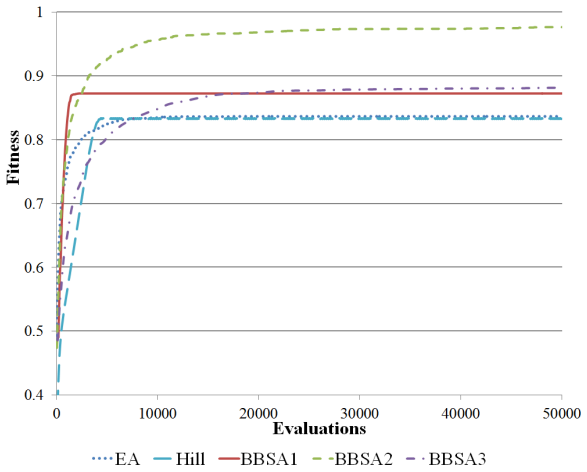

Figure 8: Comparison of an EA, Hill-Climber, and the BBSAs evolved with bit-length=100 and trap size=5 and evaluated on bit-length=100 and trap size=5

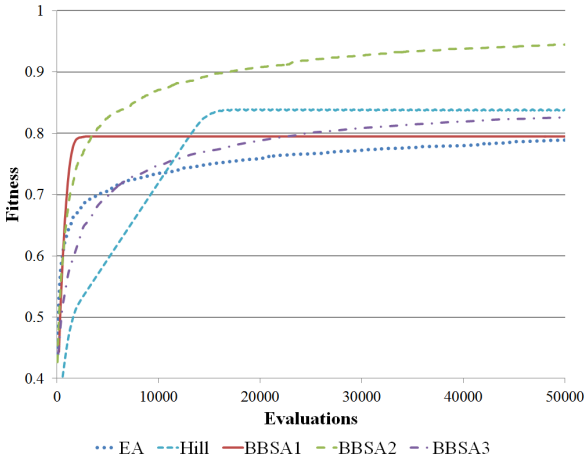

Figure 9: Comparison of an EA, Hill-Climber, and the BBSAs evolved with bit-length=100 and trap size=5 and evaluated on bit-length=200 and trap size = 5

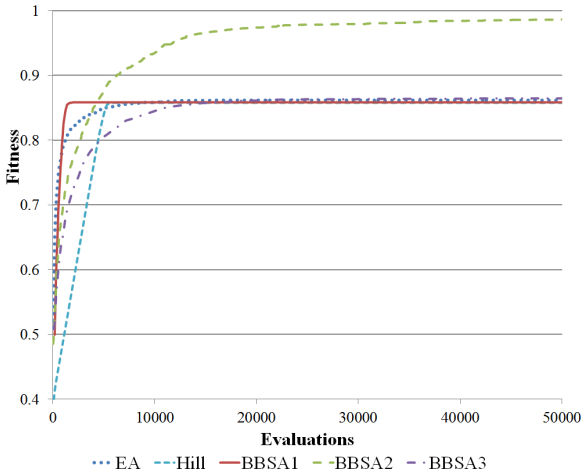

Figure 10: Comparison of an EA, Hill-Climber, and the BBSAs evolved with bit-length=100 and trap size=5 and evaluated on bit-length=105 and trap size = 7

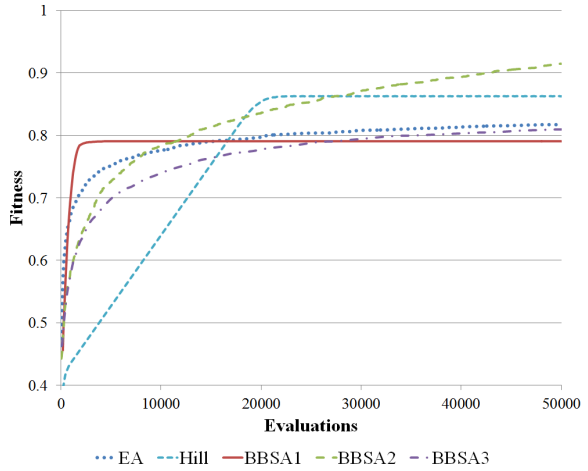

Figure 11: Comparison of an EA, Hill-Climber, and the BBSAs evolved with bit-length=100 and trap size=5 and evaluated on bit-length=210 and trap size = 7

Bit-Length	Trap Size	EA	Hill-Climber	BBSA1	BBSA2	BBSA3
100	5	0.836 (0.0245)	0.834 (0.0145)	0.872 (0.0236)	0.976 (0.0102)	0.881 (0.0275)
200	5	0.789 (0.0249)	0.839 (0.0108)	0.795 (0.0273)	0.945 (0.00990)	0.826 (0.0178)
105	7	0.862 (0.0277)	0.858 (0.00884)	0.858 (0.0149)	0.986 (0.00841)	0.864 (0.0195)
210	7	0.818 (0.0208)	0.863 (0.00517)	0.791 (0.0219)	0.915 (0.0195)	0.810 (0.0218)

Table 4: Final results of all tests averaged over 30 runs with standard deviation

The algorithms BBSA1 and BBSA3 performed no better than the EA and the Hill-Climber. It appears as though these BBSAs over-specialized to the problem configuration they were evolved on.

BBSA2, however, performed better than all other algorithms on all problem configurations. Its only noticeable drawback is its relatively slow convergence. This BBSA shows that it is possible to evolve generic solvers that can perform very well on a problem class regardless of problem configuration.

In all experiments, the EA converges more quickly than the evolved BBSA; the Hill-Climber converges more quickly than two of the three evolved BBSAs. This is primarily due to the speed at which the evolved algorithms converge being secondary to solution quality. This problem might be avoided by using Multi-Objective GP which would allow the user to select the trade-off between speed and quality that best suits their needs.

The BBSAs that were evolved for this problem preferred to use diagonal recombination rather than uniform recombination. This is primarily due to how the problem was represented. Each trap was in a continuous part of the bit-string and thus, it would be more beneficial for those parts to be kept together to ensure the integrity of the already solved traps.

This experiment also confirmed an observation from the preliminary experiments that there is redundancy in the structure of the algorithm. An example of this can be seen in BBSA3. On the right side of the tree the 'Last' set is added to itself which yields the 'Last' set. This add operation could be replaced by the 'Last' set and would behave in the same way. Other examples of this were when a set would be evaluated multiple times without being altered. In this case one of the evaluations could be removed without changing how the BBSA performed. Some of these redundancies are very difficult to remove with the standard GP recombination and mutation operations. One way to fix this would be a pruning method that would intelligently remove redundant nodes in the tree.

From analysis of the populations of the failing runs, the failure was most likely due to a problem with diversity of the population. Upon examination of the runs in which they did succeed in finding good solutions, this problem of diversity still existed. Once the GP is made multi-objective, this problem with diversity might be fixed by employing the crowding distance metric of NSGA-II [3] by determining how similar the structure of the BBSAs are.

Table 11 shows that nine of the fifteen BBSAs performed better than both the EA and Hill-Climber. The remaining six algorithms performed worse than both the EA and Hill-Climber. This demonstrates that this approach not only has the ability to create well performing algorithms, but can create them more consistently than previous methods.

7. CONCLUSIONS

In this paper it was shown that using GP it is possible to evolve BBSAs that can beat canonical BBSAs for a given problem class. Though many of the primitives were extracted from these canonical BBSAs, the resulting BBSAs bear little resemblance to them.

One problem with the current method is that the algorithms can become over-specialized if the problem class can have multiple problem configurations. In the case of BBSA1

Bit-Length	Trap Size	Result	p-Value
100	5	+	6.69 E-9
200	5	∼	0.141
105	7	∼	0.145
210	7	-	2.91 E-8

Table 5: T-Test results for evolved BBSA1 and EA with α=0.05

Bit-Length	Trap Size	Result	p-Value
100	5	+	1.2 E-9
200	5	-	4.23 E-11
105	7	∼	0.844
210	7	-	1.23 E-15

Table 6: T-Test results for evolved BBSA1 and Hill-Climber with α=0.05

Bit-Length	Trap Size	Result	p-Value
100	5	+	1.35 E-42
200	5	+	1.38 E-56
105	7	+	6.16 E-52
210	7	+	2.27 E-25

Table 7: T-Test results for evolved BBSA2 and EA with α=0.05

Bit-Length	Trap Size	Result	p-Value
100	5	+	1.58 E-42
200	5	+	2.91 E-43
105	7	+	1.61 E-52
210	7	+	2.97 E-15

Table 8: T-Test results for evolved BBSA2 and Hill-Climber with α=0.05

Bit-Length	Trap Size	Result	p-Value
100	5	+	7.93 E-13
200	5	+	3.53 E-13
105	7	∼	0.217
210	7	∼	.0412

Table 9: T-Test results for evolved BBSA3 and EA with α=0.05

Bit-Length	Trap Size	Result	p-Value
100	5	+	1.77 E-10
200	5	-	0.00179
105	7	∼	0.0875
210	7	-	4.43 E-14

Table 10: T-Test results for evolved BBSA3 and Hill-Climber with α=0.05

BBSA	EA	Hill-Climber
1	+	+
2	+	+
3	+	+
4	-	-
5	+	+
6	+	+
7	+	+
8	-	-
9	-	-
10	-	-
11	+	+
12	-	-
13	+	+
14	+	+
15	-	-

Table 11: T-test results for all fifteen evolved algorithms run on the evolved problem configuration with α=0.05

and BBSA3, they performed well for the problem configuration they were evolved on, but they did not perform as well on other problem configurations. BBSA2, on the other hand, did not over-specialize and performs very well on every problem configuration. This shows that this method can evolve general problem solvers for a problem class.

8. FUTURE WORK

The next step to improve upon the presented approach, is to solve the issue of over-specialization. This might be achieved by evolving the BBSAs using multiple problem configurations. Each evolved BBSA would be evaluated using a set of problem configurations that better represents the problem configurations that the user cares about.

Multi-Objective GP should be employed to evolve BBSAs to allow users to select the BBSA with the best trade-off between speed and solution-quality for their purposes.

The proposed approach needs to be tested on a wider variety of problem classes to validate it more thoroughly. While this paper demonstrates that the proposed method can evolve efficient BBSAs for the deceptive trap problem, it is yet to be proven that this method will work well for other problems and representations.

Finally, while the specific focus of this paper was to evolve tailored BBSAs which significantly outperform more general BBSAs on specific problem classes, the proposed approach can easily be extended to evolve more general purpose BBSAs to compete directly with established general purpose BBSAs such as EAs.

9. REFERENCES

[1] P. J. Angeline. Two Self-Adaptive Crossover Operators for Genetic Programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming*, pages 89–109. MIT Press, Cambridge, MA, USA, 1996.

[2] K. Deb and D. Goldberg. Analyzing Deception in Trap Functions. In *Proceedings of FOGA II: the Second Workshop on Foundations of Genetic Algorithms*, pages 93–108, 1992.

[3] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[4] L. Dioşan and M. Oltean. Evolutionary Design of Evolutionary Algorithms. *Genetic Programming and Evolvable Machines*, 10(3):263–306, Sept. 2009.

[5] L. S. Diosan and M. Oltean. Evolving Evolutionary Algorithms Using Evolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '07, pages 2442–2449, New York, NY, USA, July 2007. ACM.

[6] B. Edmonds. Meta-Genetic Programming: Co-evolving the Operators of Variation. CPM Report 98-32, Centre for Policy Modelling, Manchester Metropolitan University, UK, Aytoun St., Manchester, M1 3GH. UK, Jan. 1998.

[7] A. E. Eiben and C. H. van Kemenade. Diagonal Crossover in Genetic Algorithms for Numerical Optimization. *Journal of Control and Cybernetics*, 26(3):447–465, 1997.

[8] B. W. Goldman and D. R. Tauritz. Self-Configuring Crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '11, pages 575–582, New York, NY, USA, July 2011. ACM.

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

[10] N. Lourenço, F. Pereira, and E. Costa. Evolving Evolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO Companion '12, pages 51–58, New York, NY, USA, 2012. ACM.

[11] M. Oltean. Evolving Evolutionary Algorithms Using Linear Genetic Programming. *Evolutionary Computation*, 13(3):387–410, Sept. 2005.

[12] M. Oltean and C. Grosan. Evolving Evolutionary Algorithms Using Multi Expression Programming. In *Proceedings of The 7th European Conference on Artificial Life*, pages 651–658. Springer-Verlag, 2003.

[13] S. Smit and A. Eiben. Comparing Parameter Tuning Methods for Evolutionary Algorithms. In *IEEE Congress on Evolutionary Computation (CEC '09)*, pages 399–406, May 2009.

[14] E. Smorodkina and D. Tauritz. Toward Automating EA Configuration: the Parent Selection Stage. In *IEEE Congress on Evolutionary Computation (CEC '07)*, pages 63–70, Sept. 2007.

[15] D. Wolpert and W. Macready. No Free Lunch Theorems For Optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, Apr. 1997.

[16] J. R. Woodward and J. Swan. Automatically Designing Selection Heuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '11, pages 583–590, New York, NY, USA, July 2011. ACM.

[17] J. R. Woodward and J. Swan. The Automatic Generation of Mutation Operators for Genetic Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO Companion '12, pages 67–74, New York, NY, USA, July 2012. ACM.