A Problem Configuration Study of the Robustness of a Black-Box Search Algorithm Hyper-Heuristic

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

Black-Box Search Algorithms (BBSAs) tailored to a specific problem class may be expected to significantly outperform more general purpose problem solvers, including canonical evolutionary algorithms. Recent work has introduced a novel approach to evolving tailored BBSAs through a genetic programming hyper-heuristic. However, that first generation of hyper-heuristics suffered from over-specialization. This paper presents a study on the second generation hyper-heuristic which employs a multi-sample training approach to alleviate the over-specialization problem. In particular, the study is focused on the affect that the multi-sample approach has on the problem configuration landscape. A variety of experiments are reported on which demonstrate the significant increase in the robustness of the generated algorithms to changes in problem configuration due to the multi-sample approach. The results clearly show the resulting BBSAs' ability to outperform established BBSAs, including canonical evolutionary algorithms. The trade-off between a priori computational time and the generated algorithm robustness is investigated, demonstrating the performance gain possible given additional run-time.

Categories and Subject Descriptors

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

General Terms

Algorithms, Design

Keywords

Black-Box Search Algorithms, Evolutionary Algorithms, Genetic Programming, Hyper-Heuristics

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

http://dx.doi.org/10.1145/2598394.2609872.

1. INTRODUCTION

Practitioners tend to be interested in solving a particular problem class which may fall anywhere on the continuum from a single instance problem to an arbitrarily large problem class. However, progress in the field of meta-heuristics has typically been aimed at solving increasingly varied problem classes. There is a clear need for meta-heuristics tunable to the needs of practitioners in terms of the scope of the problem classes of interest, whether that be solving solely instances of MAXSAT with a fixed clause length and set number of variables, or arbitrary MAXSAT instances.

A novel approach to creating BBSAs through a hyper-heuristic using Genetic Programming (GP) demonstrated that there are problem classes for which BBSAs can be evolved which significantly outperform established BBSAs, including canonical the Evolutionary Algorithm (EA) [12]. That approach, however, had the drawback of tending to overspecialize the BBSAs to outperform established algorithms only on the trained problem configurations.

This paper presents a study on the second generation hyper-heuristics employing a multi-sample training approach which drastically decreases the probability of evolving BBSAs that suffer from over-specialization [13]. It is focused on the affect that the multi-sample approach has on the problem configuration landscape. An investigation is presented on the trade-off between the extra a priori computational time due to increasing sampling size and the increased robustness of the generated BBSAs in terms of lower variation in performance when varying the problem configuration. This is of critical importance to practitioners who need to be able to rely on the consistency of the generated BBSAs on all instances of their problem class of interest.

The goal of the research reported in this paper is to show that increasing the multi-sampling level increases the robustness of the generated BBSAs. Two primary measures of robustness are employed [7], as seen in Figure 1. The first is $fallibility$; if this value is large, it means that the BBSA can have a large difference in performance depending on the location on the problem configuration landscape. The second measure is *applicability*; it indicates the size of the problem configuration space in which the BBSA performs better than a threshold value. For a BBSA to be highly robust, it should have a small fallibility and a large applicability.

2. RELATED WORK

Most previous work on employing evolutionary computing to create improved BBSAs has focused on tuning parame-

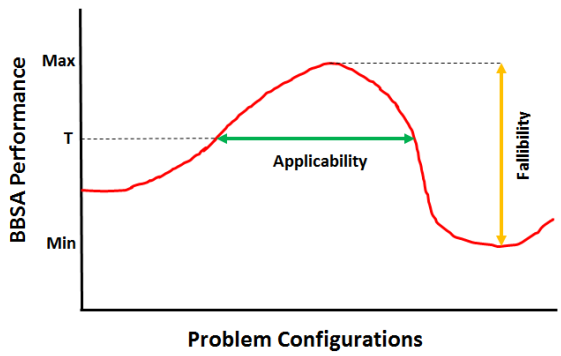

Figure 1: This figure demonstrates the concepts of Applicability and Fallibility. Applicability is the proportion of the problem configuration space that a BBSA can perform higher than a given threshold value. Fallibility is the difference between the highest and lowest performing problem configurations.

ters [16] or adaptively selecting which of a pre-defined set of operators to use and in which order [15]. 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 GP [4, 5, 14]. 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 evolving BBSAs employed Grammatical Evolution (GE) [10] which uses a grammar to describe structure, but is constrained to the canonical EA model. In later work [11], due to the computational load necessary for evaluating algorithms, a study was presented on how restricting the computational time for evaluating the evolved algorithms affects the structure.

First attempts at applying GP to the generation of BBSAs was to evolve individual EA operators [1, 6]. The primary effort has been to create improved EA variation operators [1, 6, 9, 19]. Some work has been done on evolving EA selection operators [17, 18].

Burke et al. described a high-level approach to evolving heuristics [2]. That approach was extended to evolve entire BBSAs of indiscriminate type [12]. This paper describes an improvement on that extension employing multi-sample evaluation to increase the robustness of the produced BBSAs.

3. METHODOLOGY

The specific focus of the research reported in this paper is to demonstrate the significant increase in the robustness of the generated algorithms to changes in problem configuration due to the multi-sample approach. GP was employed to evolve the algorithms where fitness was based upon the performance averaged over a set of training problem configurations.

3.1 Parse Tree

In order to condense the quantity of code needed to be evolved, the common iterative nature of BBSAs is exploited by representing a single iteration of a BBSA rather than the entirety of the algorithm. A parse tree is used to represent the iteration for the evolutionary process such that standard GP operators will work effectively.

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 sets(s) and then return a single set of solutions. The nodes continue operating in a post-order fashion and 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. The terminal nodes can either be sets of previous solutions or a set of randomly generated solutions. The sets include the 'Last' set as well as auxiliary sets which will be explained in Section 3.2.4. An example of a BBSA represented as a parse tree and related code representation are shown in figures 2 and 3.

3.2 Nodes

The non-terminal nodes that compose these trees are operations extracted from pre-existing algorithms. The nodes are broken down into selection, variation, set-manipulation, terminal, and utility nodes. The following subsections describe the operations employed of each type for the experiments reported in this paper.

3.2.1 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 to it, n being one of its parameters.

3.2.2 Variation Operation Nodes

For the experiments, three primary variation operations are used; the first one is standard bit-flip mutation. This operation has a single argument, *rate*, which is the probability that a given bit is flipped. The second operation is the standard uniform recombination with an arbitrary number of parents. This operation has a single argument, *count*, which designates the number of children generated. The final primary variation operation is diagonal crossover [8] which returns the same number of solutions as are passed in. This variation node has one parameter, n, which determines the number of points used by the crossover operation.

3.2.3 Set Operation Nodes

The experiments reported in this paper employ two distinct set operations. The first is the union operation. 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 3.2.4.

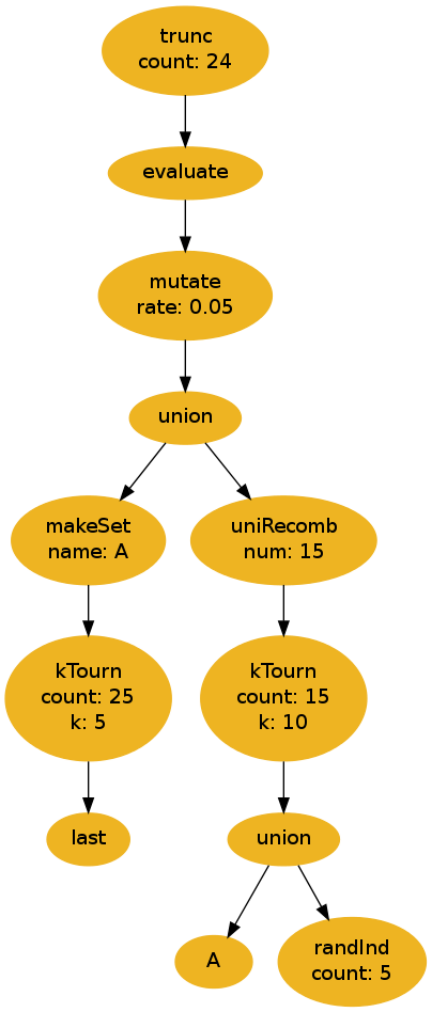

Figure 2: Example Parse Tree

```
Last = [initialize population]
evaluate(Last)
A = [ ]
while termination condition not met do
    X = kTournament(Last, k = 5,count =25)
    A = X
    Y = randInd(count = 5)
    Y = A + Y
    Y = kTournament(Y,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 3: Example Parse Tree Generated Code

3.2.4 *Terminal Nodes*

The terminal nodes in this representation are sets of solutions. They can either be the 'Last' set returned by the previous iteration, a set that was created by the save operation, or a set of randomly created solutions. The 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 previous iteration. At the beginning of each run, the saved sets are set to the empty set and the 'Last' set is set to a randomly generated population of solutions. The randomly generated set of solutions terminal node creates a set of n solutions, n being one of its parameters, and returns that to its parent node.

3.2.5 *Utility Nodes*

There is currently one utility operation employed for use in the experiments. This node is the evaluation node which evaluates all of the solutions that are passed into it. Operations that can be added to this group in the future can include looping nodes and conditional nodes.

3.3 Meta-Algorithm

GP is employed to meta-evolve the BBSAs. The two primary variation operators employed are the standard subtree crossover and mutation altered to make the maximum number of nodes being added a user defined value. Another mutation operation was added to this algorithm that with equal chance randomizes the size of the initial 'Last' set or selects a random node from the parse-tree and randomizes the parameters if it has any; if the node does not have any parameters, the mutation is executed again. To ensure that the genetic program produces good BBSAs, the ones which do not evaluate any solutions are discarded upon generation.

3.3.1 *Black-Box Search Algorithm*

Each individual in the GP population encodes 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 population initialization, followed by the parse-tree being repeatedly evaluated until one of the termination criteria is met. Once a run of the BBSA is completed, the 'Last' set and all saved sets are 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 converges and what the average solution quality and best current solution is.

The fitness of a BBSA is estimated by computing the fitness function that it employs on the solutions it evolves averaged over multiple runs. Parsimony pressure is added to temper the growth of the parse trees. 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.

Learning conditions were added to terminate poor solutions before they are fully evaluated in order to ameliorate the very computationally intensive nature of hyper-heuristics analogously to [11]. This is accomplished by applying four limiting factors. First of all, if a BBSA exceeds the maximum number of evaluations, then it will automatically be terminated 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

Number of Samples	Bit-Length	Trap Size
1	100	5
2	200	5
3	105	7
4	210	7
5	300	5

Table 1: Problem Configurations for Multi-Sampling Test. Each test includes prior tests' problem configurations; e.g., the run in which there are two problem configurations uses the first two problem configurations shown.

to evolve algorithms with more evaluations per iteration. 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 perform. 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 prevent inefficient algorithms which despite evaluating few solutions incur a high computational cost. The fourth method terminates algorithms which have converged based on not having improved in i iterations.

3.3.2 Multi-Sampling

A major issue identified in [12] is the problem of over-specialization when training on a single problem configuration of a given problem class. Following the approach suggested in [12], the BBSAs are executed on multiple problem configurations of the problem class of interest. On each problem configuration, the BBSAs run a user-specified number of times. This addition allows the user to control the robustness of the generated BBSA. If the user requires a BBSA that performs very consistently, then running the algorithm with more problem configurations is beneficial.

3.3.3 External Verification

To assure that the performance of the evolved BBSA is consistent with its performance reported during evolution, executable code is generated to represent the parse tree as a full BBSA. This is done to externally verify that the performance that the GP shows for a given BBSA is accurate when actually implemented. The generated code is used in all of the experiments to insure unbiased execution of the BBSAs. An example of a parse tree and pseudo-code generated can be found in Figure 2 and Figure 3. This verification was employed for the testing of the BBSAs in all experiments.

4. EXPERIMENTS

To demonstrate that employing multi-sample evaluation of the BBSA reduces the probability of over-specialization, the algorithm was run on a series of multi-sampling levels, where a level is defined by the number of training problem configurations it samples. Once the BBSA has been evolved with a given multi-sampling level, it is tested on a super-set of problem configurations to determine the preliminary robustness of the BBSA and to demonstrate that they can out perform a standard EA.

Bit-Length	Trap Size
100	5
200	5
105	7
210	7
300	5
99	9
198	9
150	5
250	5
147	7
252	7

Table 2: Problem Configurations that were used to test the robustness of the BBSA.

Parameter	Value
Population Size	50
Children per Generation	20
Parent Selection k	15
Recombination	Uniform
Mutation Rate	5%
Survival Selection	Truncation

Table 3: EA Configurations

The classic Deceptive Trap problem [3] is employed as benchmark in this paper. It divides a bit-string into traps of size j bits each which are scored using the following equation where t is equal to the sum of the bit values in the trap.

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

This problem was chosen to compare the results in this paper with those in [12], where BBSAs were evolved and suffered from over-specialization.

The BBSAs were evolved with a multi-sampling level from one to five. The problem configurations are shown in Table 1. Each run includes the problem configurations from the runs before; e.g., the runs with two samples use the problem configurations from the first two rows. For each evolved BBSA, code was generated to determine its robustness externally from the evolution. To test the preliminary robustness of the generated BBSAs, they were run on a super-set of problem configurations as shown in Table 2. This set includes the training set to validate that the fitness found during evolution is accurate.

The EA has an initial population of 50 and generates 20 children each generation. It uses k-Tournament with replacement for parent selection with k being 15, uniform recombination, bit-flip mutation with a 5% rate, and truncation survivor selection. The EA parameter settings are summarized in Table 3. These values were selected to be similar to those in [12] with minor hand tuning to perform well with the first problem configuration shown in Table 1.

For these experiments, four BBSAs were evolved at each multi-sampling level. During the evolution each problem configuration was run five times. Meaning that in the experiment with multi-sampling level one, each BBSA evaluation is five runs where with multi-sampling level five, each

Parameter	Value
Evaluations	5000
Initial Population	100
Children per Generation	40
k-Tournament	8
Sub-Tree Crossover Probability	30%
Sub-Tree Mutation Probability	30%
Alternate Mutation Probability	40%
Alternate Mutation Depth	5
Parsimony Pressure	0.001
Maximum Operations	5,000,000
Maximum Iterations	10,000
Maximum Evaluations in BBSA	100,000

Table 4: GP Configurations

BBSA evaluation is twenty-five runs. All of the testing data was produced by executing the code generated by the meta-algorithm. Each of the evolved BBSAs were executed 30 times for each of the problem configurations. Each algorithm was run for 100,000 evaluations. These results were compared to an EA executed 30 times for each of the problem configurations.

After collecting results from the first experiment which was focused on determining the preliminary robustness increase caused by multi-sampling, a secondary experiment was run to study the effect of multi-sampling on the performance landscape across a wide set of problem configurations. The areas of interest in this experiment are the problem configurations that were significantly different from the trained problem configurations. The BBSA with the largest fallibility, where fallibility indicates the difference between best and worst performance on the test problem configurations, was selected from each multi-sampling level to demonstrate a worst case scenario. These BBSAs, along with an EA, were run on all problem configurations with k from 4 to 20 inclusive and bit-lengths from roughly 70 to 500. The algorithms were run five times on each problem configuration.

All of the experiments were conducted under the same settings. The meta-algorithm was run for 5000 evaluations. The initial population was 100 individuals and each generation 40 new individuals were created. k-tournament selection with replacement and $k = 8$ was employed for parent selection. The sub-tree crossover and mutation operations had 30% chance of being used while the alternate mutation had a probability of 40%. The parsimony pressure for the tree size was 0.001. The maximum number of operations the BBSAs could use was 5,000,000, the maximum number of iterations was 10,000, and the maximum number of evaluations in the BBSA was 100,000. All the parameter settings for the meta-algorithm are summarized in Table 4. Due to the high computational cost of running hyper-heuristics, only minimal tuning of the meta-algorithm is feasible.

For the generation of the BBSAs, heuristic constraints were employed to limit various parameters to reasonable values. The maximum number of individuals in the initial population was set to 50. The range of individuals selected by selection nodes was set to be from 1 to 25 inclusive. The range of the k value used for the k-tournament is from 1 to 25 inclusive. The range of the number of points for diagonal crossover is from 1 to 25 points inclusive. All the parameter settings for the BBSA are summarized in Table 5.

Parameter	Value
Runs per Problem Configuration	5
Maximum Initial Population	50
k Value Range	[1,25]
Number of Selected Individuals Range	[1,25]
Crossover Points Range	[1,25]
Randomly Generated Set Size Range	[1,25]
Children for Uniform Recombination Range	[1,25]

Table 5: Black-Box Search Algorithm Settings

Level	Run	Train Fit.	Test Fit.	Fallibility
1	1	1.0	0.976	0.094
1	2	1.0	0.999	8.33 E-3
1	3	0.944	0.883	0.082
1	4	0.976	0.894	0.224
2	1	0.997	0.996	0.023
2	2	0.992	0.959	0.130
2	3	0.966	0.970	0.054
2	4	0.979	0.947	0.120
3	1	0.965	0.966	0.050
3	2	0.984	0.980	0.065
3	3	0.899	0.886	0.059
3	4	0.926	0.898	0.073
4	1	0.976	0.999	5.00 E-3
4	2	0.973	0.969	.0903
4	3	0.982	0.975	0.059
4	4	0.993	0.999	5.00 E-3
5	1	0.973	0.977	0.050
5	2	0.893	0.879	0.035
5	3	0.850	0.850	0.045
5	4	0.955	0.986	0.029

Table 6: BBSA Experimental Results

5. RESULTS

The first experiment's results are summarized in Table 6. This table shows the fitness of the BBSAs at the end of evolution labelled as the 'Training Fit.'. The 'Test Fit.' is the averaged fitness across the testing set of problem configurations shown in Table 2. The 'Fallibility' field is the difference between the best and worst performing problem configuration for a given BBSA. As this number decreases, the BBSA can be said to be a more robust algorithm.

The comparison between the EA and the evolved BBSAs is shown in Table 7. The $-$ column represents the number of problem configurations that the EA performed better on than the BBSA. The $\sim$ column represents the number of problem configurations that there was no statistical difference between the EA and the BBSA. The $+$ column represents the number of problem configurations that the EA performs worse on than the BBSA. The t-test with $\alpha = 0.05$ was used to determine the statistically better algorithm.

To study the effect of multi-sampling on the performance landscape across a wide set of problem configurations, 3-dimensional plots were generated that represent the quality of solutions that can be found using different problem configurations. Figures 4-8 show the least robust BBSA evolved

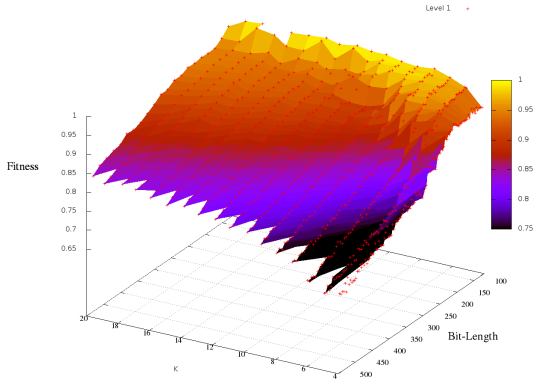

Figure 4: The worst BBSA found for multi-sampling level one, run on the problem configuration space.

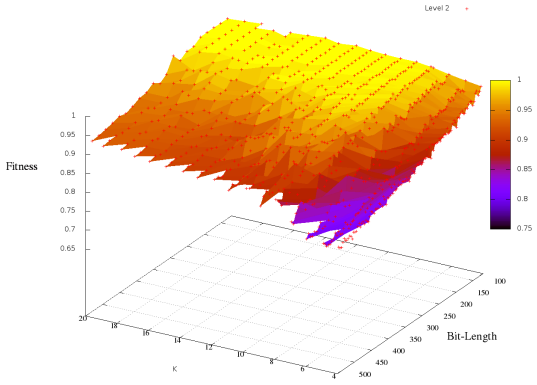

Figure 5: The worst BBSA found for multi-sampling level two, run on the problem configuration space.

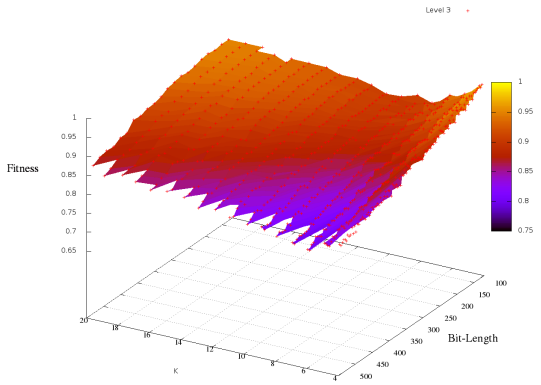

Figure 6: The worst BBSA found for multi-sampling level three, run on the problem configuration space.

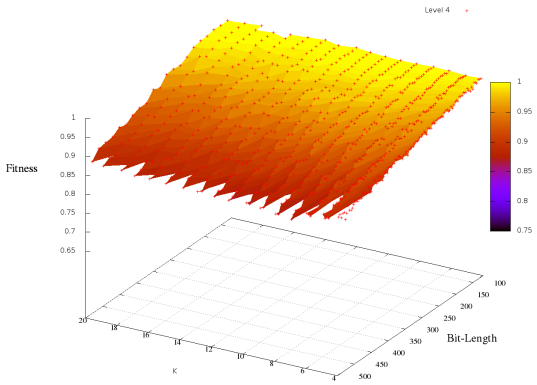

Figure 7: The worst BBSA found for multi-sampling level four, run on the problem configuration space.

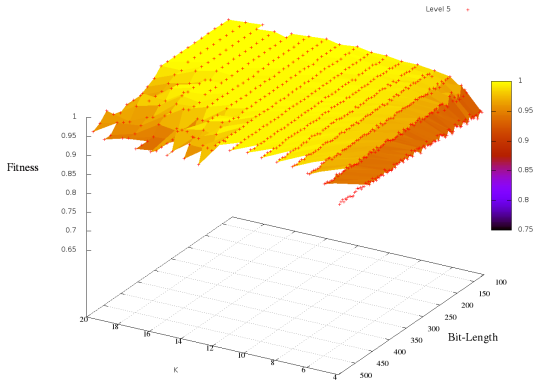

Figure 8: The worst BBSA found for multi-sampling level five, run on the problem configuration space.

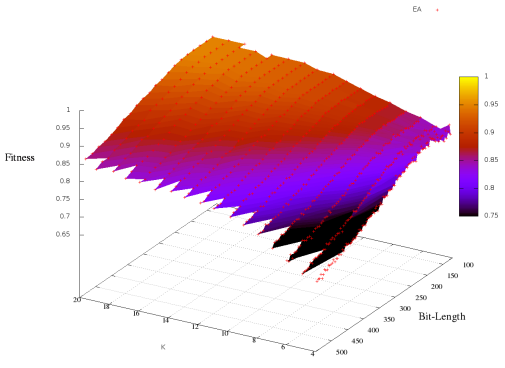

Figure 9: A standard EA run on the problem configuration space.

Level	Run	+	∼	−
1	1	11	0	0
1	2	11	0	0
1	3	11	0	0
1	4	6	2	3
2	1	11	0	0
2	2	11	0	0
2	3	11	0	0
2	4	11	0	0
3	1	11	0	0
3	2	11	0	0
3	3	11	0	0
3	4	11	0	0
4	1	11	0	0
4	2	11	0	0
4	3	11	0	0
4	4	11	0	0
5	1	11	0	0
5	2	10	1	0
5	3	7	4	0
5	4	11	0	0

Table 7: This table is a summary of the comparison of the evolved BBSA and the standard EA.

at each multi sampling level. Figure 9 shows the baseline of a standard EA. These plots were generated averaging over five runs on each problem configuration.

6. DISCUSSION

The goal of the research reported in this paper is to show that increasing the multi-sampling level increases the robustness of the generated BBSAs. The two measurements of robustness that we chose to use were applicability and fallibility. Applicability is the size of the problem configuration space in which the BBSA performed higher than a given threshold value. Fallibility is the difference between the best and worst performing problem configuration. As the applicability increases and the fallibility decreases, the robustness of the BBSA increases. The results presented show that both of these happen as the multi-sampling level is increased.

T-tests were run on the selected testing problem configurations, the results of which are shown in Table 7. It can be seen that one of the BBSAs that was evolved with multi-sampling level one performed worse than the EA. From a practitioner's standpoint, this result would seem very surprising compared to its trained fitness. The runs of multi-sampling level five performed consistent with the trained fitness when compared to the EA.

Figure 4 shows the performance of the least robust BBSA found using multi-sampling level one when run on a wide variety of problem configurations. As can be seen, the BBSA performs well in the immediate area around the problem configuration that it was trained on ($k = 5$, $bit - length = 100$). Unsurprisingly, as the problem configuration gets farther away from the trained problem configuration, the fitness decreases. This algorithm performs similarly to other algorithms that are tuned to specific problem configurations. When compared to how the EA performs on the same problem configuration space in Figure 9, the BBSA outperforms the EA in problem configurations near the trained problem configuration, but performs at near the same level as the distance increases. As can be seen in figures 4-8, the variance in performance of the algorithm decreases as the multi-sampling level increases.

Training a BBSA on a larger number of training problem configurations improves the performance of the BBSA. However, in most cases, the improved performance is restricted to problem configurations that are relatively close to the trained problem configurations. These results look similar to the results shown in Figure 4 where near the trained problem configuration the BBSA performs well, but as the problem configuration differs more from the trained problem configuration, the BBSA performs poorly.

When multi-sampling is performed during the generation of the algorithm rather than solely the parameter tuning, the increased performance of the algorithm can be generalized to larger portions of the problem configuration space. As can can be seen in figures 4-8, the fallibility decreases as the multi-sample level increases. Note that the training sets that these algorithms were evolved on had problem configurations with k ranging from 5 to 7 and a $bit - length$ ranging from 100 to 300 and the problem configuration space shown in figures 4-8 includes a k from 4 to 20 and a $bit - length$ from approximately 75 to 500. This demonstrates the enhanced robustness of the BBSAs evolved with a higher multi-sampling level. This robustness is superior to that of strictly parameter optimization of a BBSA due to its ability to generalize to problem configurations much different to those trained on.

There was a case in which a BBSA evolved with a multi-sampling level of one, when tested, was shown to be robust. As in previous work [12], there is always the potential of producing a robust algorithm trained on a single sample; however, this is highly unreliable and the method introduced in this paper significantly increases the probability of evolving a robust BBSA.

One drawback of this method is the increased computational time that it requires. One cause of this increase are the additional runs that are necessary during the evaluation of a given BBSA. This extra computational time increases linearly with the multi-sampling level. It was noticed during testing and in the final results, that the experiments run at a higher multi-sampling level can have a lower average fitness. Due to this result, a trend in the applicability is not statistically discernible. As is shown in Table 6, the BBSAs evolved at multi-sampling level five had the lowest trained fitness. This is believed to be caused by the increased difficulty of finding an algorithm that performs well on all of the training problem configurations. These two aspects cause the computational time increase of $\Omega(L)$, with L being the multi-sampling level.

7. CONCLUSIONS

The research reported in this paper demonstrates that employing the proposed multi-sampling method tends to increase the robustness of evolved BBSAs. This method is shown to not only generate BBSAs that generalize to the problem configuration space close to the trained problem configurations, but to create BBSAs that have generalized to a much wider area of the problem configuration landscape. Though it is possible to evolve robust algorithms without using the multi-sampling method, it is shown that with a higher multi-sampling level, the general robustness

of the evolved BBSA is increased along with the certainty that the evolved BBSA will indeed be robust.

The predominant disadvantage to this method is the increased computational time that is necessary to evolve high-performance BBSAs when using high multi-sampling levels. However, it is shown that it is possible to evolve high-performance BBSAs when using high multi-sampling levels that are also robust as shown in Table 6.

8. FUTURE WORK

The next step to improve upon the proposed approach is to perform run-time analysis of the BBSAs. This will help reveal what features cause the BBSAs to run slower compared to others. This allows the computational time necessary for running with a higher multi-sampling level to decrease which will make this approach more feasible for practitioners.

This multi-sampling approach also needs to be tested on a larger variety of problem classes to better understand how the multi-sampling level affects the robustness of evolved BBSAs. A larger variety of node operations may have to be added to allow this approach to create well-performing BBSAs. As well as testing this approach on other problem classes, an in depth study should be conducted to determine the correlation between the proximity of the training classes and the robustness of the resulting BBSAs.

Finally, multi-objective optimization should be introduced into the meta-algorithm such that it is capable of creating BBSAs that are not only robust, but quick to converge as well. This is necessary to enable the proposed method to evolve human-competitive BBSAs.

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] E. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and J. Woodward. Exploring Hyper-heuristic Methodologies with Genetic Programming. In C. Mumford and L. Jain, editors, *Computational Intelligence*, volume 1 of *Intelligent Systems Reference Library*, pages 177–201. Springer, 2009.

[3] 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.

[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 GECCO 2007 - Genetic And Evolutionary Computation Conference*, GECCO '07, pages 2442–2449, New York, NY, USA, 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. Eiben and S. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19 – 31, 2011.

[8] 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.

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

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

[11] N. Lourenço, F. B. Pereira, and E. Costa. The Importance of the Learning Conditions in Hyper-heuristics. In *Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference*, GECCO '13, pages 1525–1532, New York, NY, USA, 2013. ACM.

[12] M. A. Martin and D. R. Tauritz. Evolving Black-box Search Algorithms Employing Genetic Programming. In *Proceeding of the Fifteenth Annual Conference Companion on Genetic and Evolutionary Computation Conference Companion*, GECCO '13 Companion, pages 1497–1504, New York, NY, USA, 2013. ACM.

[13] M. A. Martin and D. R. Tauritz. Multi-Sample Evolution of Robust Black-Box Search Algorithms. In *Proceeding of the Sixteenth Annual Conference Companion on Genetic and Evolutionary Computation Conference Companion*, GECCO '14 Companion, New York, NY, USA, 2014. ACM.

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

[15] 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.

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

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

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

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