Fall 2015

Hyper-heuristics for the automated design of black-box search algorithms

Matthew Allen Martin

HYPER-HEURISTICS FOR THE AUTOMATED DESIGN OF BLACK-BOX SEARCH ALGORITHMS

by

MATTHEW ALLEN MARTIN

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2015

Approved by

Dr. Daniel Tauritz, Advisor
Dr. Zhaozheng Yin
Dr. Samuel Mulder

PUBLICATION THESIS OPTION

This thesis has been prepared in the form of three papers formatted to university standards.

Paper 1. Pages 4–29 have been published as *Evolving Black-Box Search Algorithms Employing Genetic Programming*, Genetic and Evolutionary Computation Conference 2013 with Daniel Tauritz.

Paper 2. Pages 30–57 have been published as *A Problem Configuration Study of the Robustness of a Black-Box Search Algorithm Hyper-Heuristic*, Genetic and Evolutionary Computation Conference 2014 with Daniel Tauritz.

Paper 3. Pages 58–83 have been published as *Hyper-Heuristics: A Study On Increasing Primitive-Space*, Genetic and Evolutionary Computation Conference 2015 with Daniel Tauritz.

ABSTRACT

Within the field of Black-Box Search Algorithms (BBSAs), there is a focus on improving algorithm performance over increasingly diversified problem classes. However, these general purpose problem solvers have no guarantee to perform well on an arbitrary problem class that a practitioner needs to solve. The problem classes that the research in this thesis most applies to are difficult problems that are going to be solved multiple times. BBSAs tailored to one of these problem class can be expected to significantly outperform the more general purpose problem solvers, including canonical Evolutionary Algorithms (EAs). The first paper in this thesis explores a novel method in which these BBSAs can be created through the use of hyper-heuristics.

Hyper-heuristics have the tendency to over-specialize on the problem configuration that it is given rather than generalizing to the problem class. The evolved BBSA should be robust to changes in problem configuration. The second paper in this thesis presents a multi-sample approach geared towards increasing the robustness of the resulting BBSAs.

As with other CI techniques, such as Genetic Programming, hyper-heuristics are affected by the size of the search space. If the hyper-heuristic has too much genetic material, it could cause the search space to be too large to effectively traverse. However if the hyper-heuristic has too little genetic material, it may not be capable of creating a high quality BBSA. The third paper in this thesis explores the scalability of hyper-heuristics as the amount of genetic material is increased. Additionally, this paper explores the impact that the nature of the added primitives have on the performance of the hyper-heuristic. These papers show that hyper-heuristics can be used to evolve BBSAs that perform well on a problem of indiscriminate type.

ACKNOWLEDGMENTS

There are many people who have greatly helped me along my path to completing this thesis to which I owe my gratitude. First and foremost, I would like to thank my adviser Dr. Daniel Tauritz. While taking his Introduction to Evolutionary Computation course, a research opportunity opened up and a fellow student recommended me for the position. After discussion with Dr. Tauritz, he gave me my first position as an undergraduate researcher. Over the years he has offered guidance in my academic and professional pursuits. He has pushed me to my full potential in both my coursework and my research. His encouragement over the years will continue to motivate me to exceed expectations in every aspect of my life.

I would also like to thank Dr. Samuel Mulder for giving extremly valuable feedback on all of my publications. His feedback, while at times critical, was essential to the completion of my thesis. I would like to thank Sandia National Laboratories for providing my funding through their Critical Skills Master's Program that made my graduate studies possible. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

I also wish to thank my entire family for supporting me over the years, especially my wife, Jennifer, and brother, Nathan. Lastly and most importantly, I would like to thank my parents Koral and Daniel, for raising, supporting, and loving me throughout my life. Their countless sacrifices have helped make me who I am today. to them, I dedicate this thesis.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

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 [1]. This means that attempting to create a BBSA that out-performs all other BBSAs on all problems is infeasible. This also implies that any general purpose BBSA, such as Evolutionary Algorithms (EAs) or Simulated Annealing (SA), has no guarantee to perform well on an arbitrary problem. This does not bode well for a practitioner with a problem not shown to be easily solvable by these general purpose BBSAs. However, by limiting the optimization problems to a sub-set of all optimization problems, a BBSA can be produced that will perform better than other BBSAs on that sub-set. Instead of using a general purpose BBSA, a BBSA that is designed specifically for that subset can be use. This ensures high-performance instead of leaving it up to chance with the general purpose BBSAs.

While custom BBSAs are a more reliable solution it raises the problem of how to design such a BBSA. Designing BBSAs that perform well on an arbitrary problem is often very difficult and require in depth knowledge of the problem which is not always available. This leaves the designer at a large disadvantage. A solution to this problem is to use a method of automated design of algorithms such as hyper-heuristics. Hyper-heuristics are a type of meta-heuristic in which the search space also consists of meta-heuristics.

Hyper-heuristics typically use Genetic Programming (GP) as their means to create meta-heuristics. In the hyper-heuristic, instead of evolving a fully functioning BBSA, it is common to evolve a single iteration of the BBSA since the typical structure of a BBSA is based on the repetition of a function which performs the search. This reduces the amount of code that is necessary to generate automatically. An ideal representation of hyper-heuristics is the standard tree-based GP, though other methods can be used as well. To determine the quality of the GP represented BBSA the BBSA is executed on the problem of interest. The quality of the BBSA is tied to the quality of the solutions it can find.

An important characteristic of the evolved BBSAs is robustness to a varying problem configuration within the problem class of interest. In this relationship a problem class is a set of problem configurations. An example of this might be where the problem class is the 3-SAT problem and a problem configuration might be a given instance of the 3-SAT problem. In this example the hyper-heuristic can generate BBSAs that outperform general SAT solvers only if it is run on a subset of SAT instances which represent a specific distribution such as those generated for circuit testing. The robustness of BBSA can be defined by two measures [2]. The first is fallibility and is defined by the difference between the performance on the best and worst problem configurations. 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 and is defined by 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.

There are two approaches to design a hyper-heuristic. The first is a bottom-up approach in which the primitives of the hyper-heuristic are low-level. Assuming the set of low-level primitives are Turing complete, any BBSA can be represented using them. Using GP techniques such as Automatically Defined Functions (ADFs) [3], common operators, such as selection and mutation operators, can be built up. Unfortunately, the search space of a bottom-up hyper-heuristic is extremely large and it can be very difficult to find any working, much less high-performing, BBSAs. The second approach is a top-down approach in which primitives of the hyper-heuristic are high-level. The high-level primitives generally include operators of pre-existing

BBSAs. The search space of a top down approach is much smaller than the bottom up method and is more likely to find working BBSAs. This advantage makes it the more practical method. The disadvantage of this method is that the process of lowering the level of primitives is a manual process. Unlike the bottom-up approach that can automatically create higher level primitives through the use of ADFs, there is currently no process of creating lower level primitives.

To solve some of these problems a hybrid approach can be done. This hybrid approach would have both high and lower level primitives. This can be done by starting with solely high-level primitives and manually decomposing these primitives into more simple primitives. Instead of solely using these new lower level primitives, they are combined with the high-level primitives. This enables working BBSAs to be evolved easily but increases the search space to allow for novelty. A draw back to this method is the increase in search space of hyper-heuristic. While the search space of this hybrid approach is not as large as the bottom-up approach, it can still hinder performance.

PAPER

I. EVOLVING BLACK-BOX SEARCH ALGORITHMS EMPLOYING GENETIC PROGRAMMING

Matthew A. Martin, and Daniel R. Tauritz

Natural Computation Laboratory

Department of Computer Science, Missouri Univerisity of Science and Technology, Rolla, MO 65409

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.

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 [1]. 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 out-perform 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 [4] or adaptively selecting which of a predefined set of operators to use and in which order [5]. 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) [6, 7, 8]. 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) [9] which uses a grammar to describe structure, but highly constrained to the standard EA model. No significant increase in result quality was reported.

Genetic Programming (GP) introduced the concept of evolving executable programs [3]. 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 [10, 11, 12, 13]. Some limited work has been done on evolving EA selection operators [14, 15]. 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 can be seen in Figure 3.1.

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

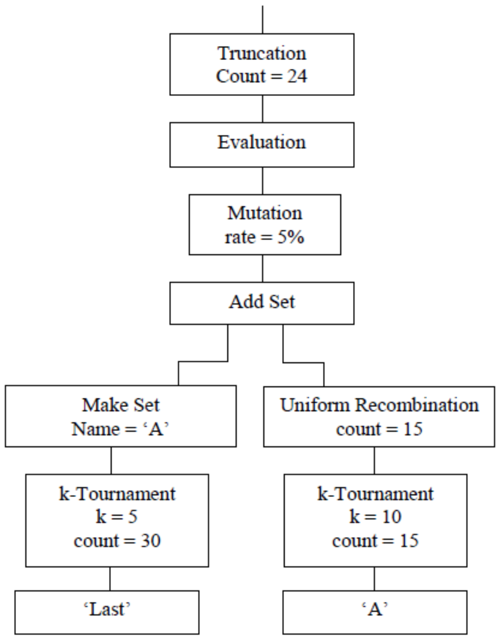

Figure 3.1: Example Parse Tree

```
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 3.2: Example Parse Tree Generated Code

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 [16] which returns the same number of solutions as were passed in. This variation node has one parameter, 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 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 bulky algorithms with few or no evaluations to be terminated. 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 can be found in Figure 3.1 and Figure 3.2. This verification was employed for the testing of the BBSAs in all experiments.

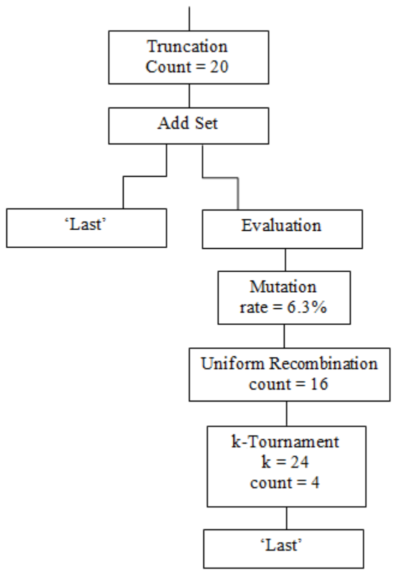

Figure 3.3: Example EA Parse Tree

```
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 3.4: EA Parse Tree Generated Code

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 the 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 3.3 which generated the code shown in Figure 3.4. The parameter values for this algorithm were found using GP's Alternate Mutation for 2000 evaluations which is the 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 [17] is employed as benchmark in this paper. 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 as the problem configuration. 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 4.1. This is done to determine if the evolved algorithm is a general solver for the problem class.

Table 4.1: Problem Configurations for Deceptive Trap

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

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.

There are many settings that can be parameter tunings that can be set for both the GP as well as the BBSA generation settings. 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 Table 4.2.

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 the 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 4.3.

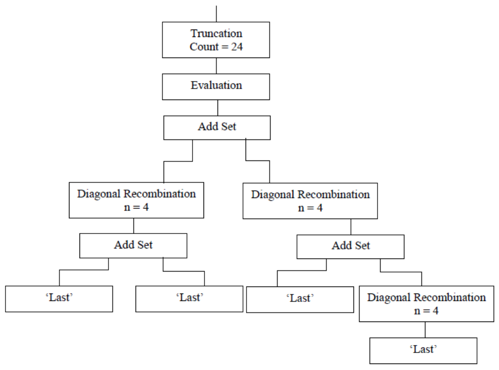

Figure 4.5: BBSA1 evolved for Deceptive Trap in parse tree form. Initial population of 49 solutions

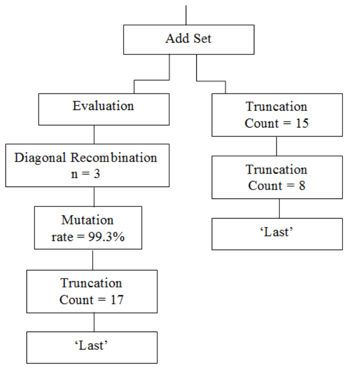

Figure 4.6: BBSA2 evolved for Deceptive Trap in parse tree form. Initial population of 29 solutions

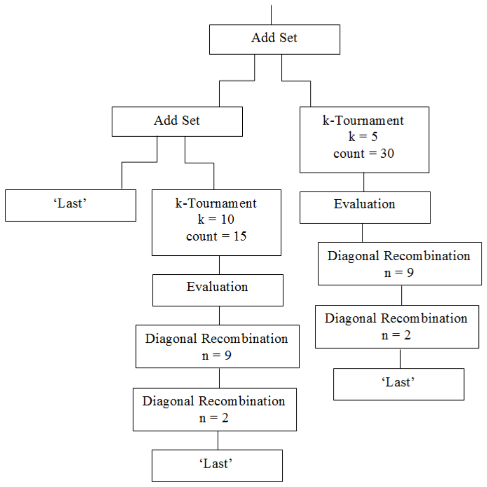

Figure 4.7: BBSA3 evolved for Deceptive Trap in parse tree form. Initial population of 39 solutions

Table 4.2: GP Configurations

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 4.3: Black-Box Search Algorithm Settings

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

5 RESULTS

Three algorithms were selected from the fifteen evolved algorithms to discuss in more detail. They were selected because they had features that help characterize the features and flaws 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 4.5-4.7. The algorithms are labeled BBSA1, BBSA2, and BBSA3, respectively.

Comparisons of the evolved BBSAs and the EA can be seen in figures 5.8-5.11. 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 4.1.

A summary of the final states of the various BBSAs can be found in Table 5.4 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 6.11.

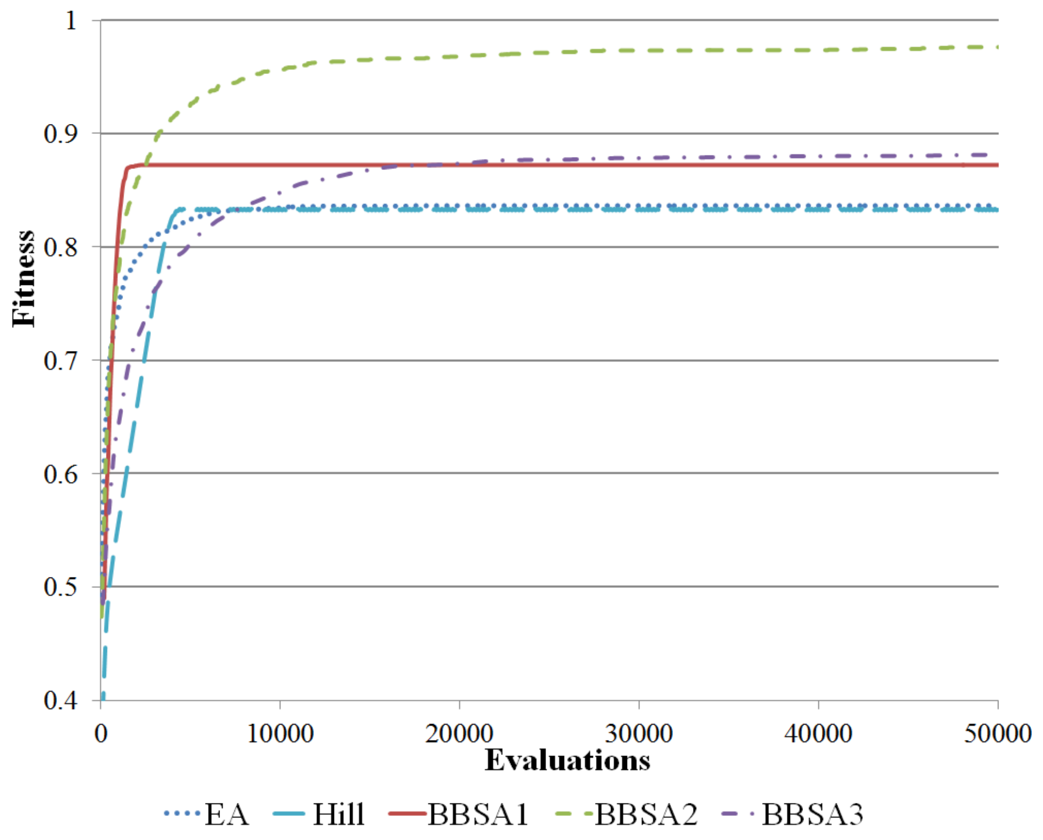

Figure 5.8: Comparison of an EA, Hill-Climber, and the BBSAs evolved with bit-length=100 and trap size=5 and evaluated on bit-length=100 and trap size=5

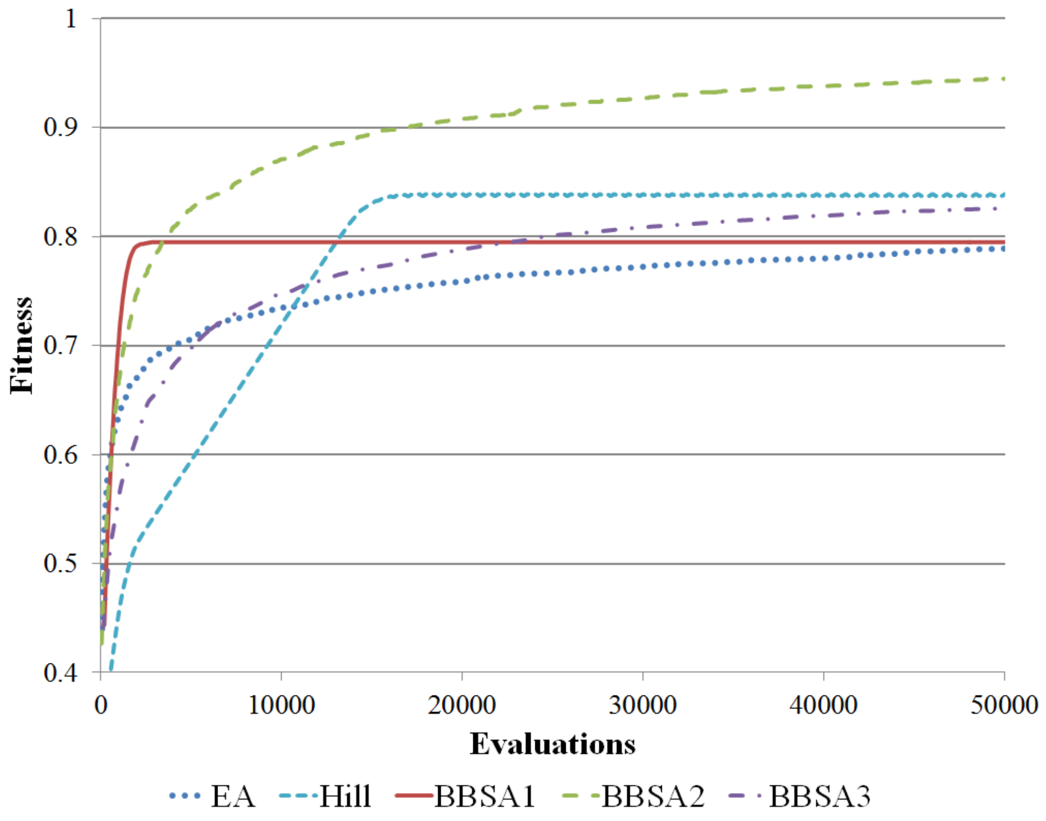

Figure 5.9: Comparison of an EA, Hill-Climber, and the BBSAs evolved with bit-length=100 and trap size=5 and evaluated on bit-length=200 and trap size = 5

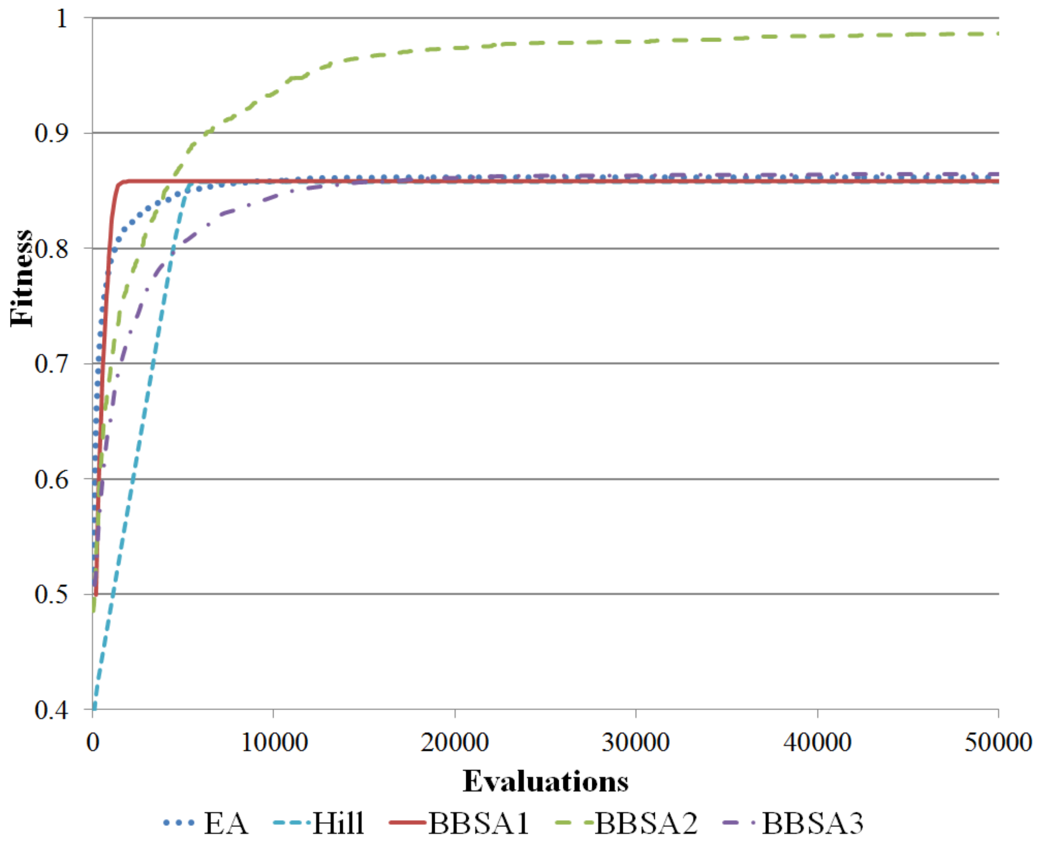

Figure 5.10: Comparison of an EA, Hill-Climber, and the BBSAs evolved with bit-length=100 and trap size=5 and evaluated on bit-length=105 and trap size = 7

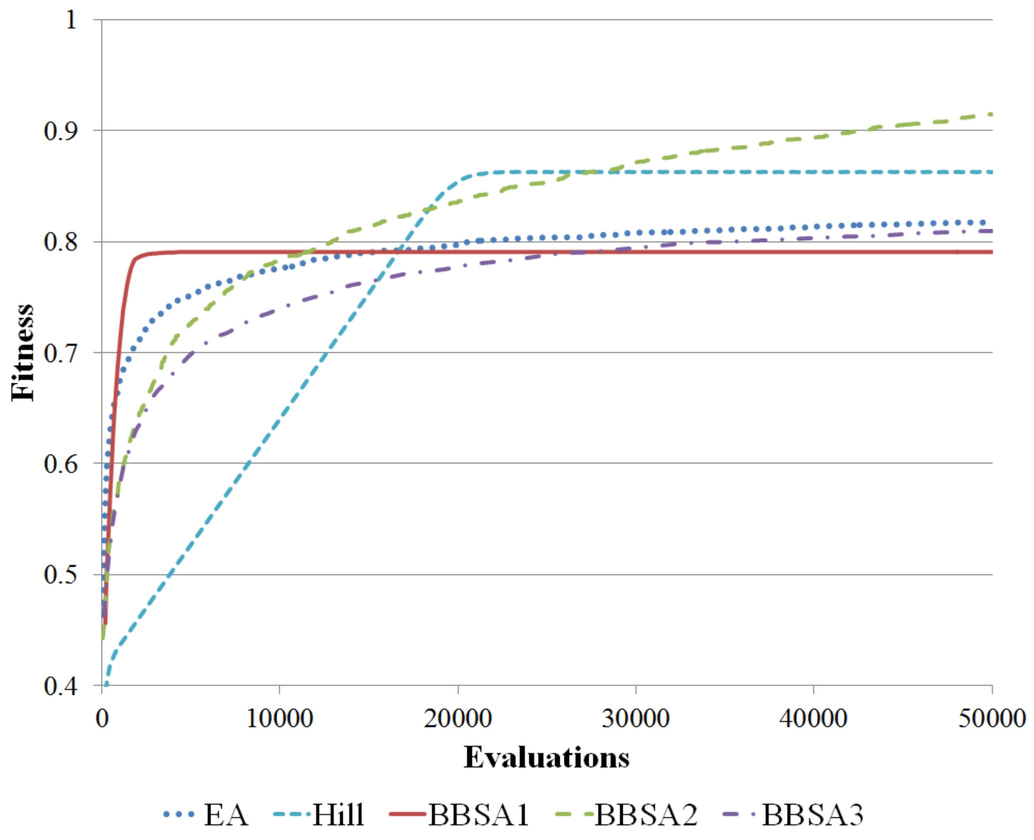

Figure 5.11: Comparison of an EA, Hill-Climber, and the BBSAs evolved with bit-length=100 and trap size=5 and evaluated on bit-length=210 and trap size = 7

Table 5.4: Final results of all tests averaged over 30 runs with standard deviation

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)

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. 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 [18] by determining how similar the structure of the BBSAs are.

In Table 6.11 it can be seen that nine of the fifteen BBSAs performed better than both the EA and Hill-Climber. The remaining six algorithms were found to perform 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.

Table 6.5: T-Test results for evolved BBSA1 and EA with α=0.05

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

Table 6.6: T-Test results for evolved BBSA2 and EA 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 6.7: T-Test results for evolved BBSA3 and EA 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	$\sim$	0.217
210	7	$\sim$	.0412

Table 6.8: T-Test results for evolved BBSA1 and Hill-Climber 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	$\sim$	0.844
210	7	-	1.23 E-15

Table 6.9: T-Test results for evolved BBSA2 and Hill-Climber 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 6.10: T-Test results for evolved BBSA3 and Hill-Climber with α=0.05

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

Table 6.11: T-test results for all fifteen evolved algorithms run on the evolved problem configuration with α=0.05

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

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. While the current nodes are high level operations, this paper shows that these high level operations can be used in a more effective manner, for certain problems, than established BBSAs.

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 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 the problem class.

8 FUTURE WORK

The next step to improve upon the proposed 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.

Other future work includes using Multi-Objective GP to evolve BBSAs. This will 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.

II. A PROBLEM CONFIGURATION STUDY OF THE ROBUSTNESS OF A BLACK-BOX SEARCH ALGORITHM HYPER-HEURISTIC

Matthew A. Martin, and Daniel R. Tauritz

Natural Computation Laboratory

Department of Computer Science, Missouri Univerisity of Science and Technology, Rolla, MO 65409

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.

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 employing Genetic Programming (GP) demonstrated that there are problem classes for which BBSAs can be evolved which significantly outperform established BBSAs, including canonical Evolutionary Algorithms (EAs) [19]. 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 [20]. 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 [2], as shown in Figure 1.1. The first is $fallibility$; if this value is large it means that the BBSA can have a large difference

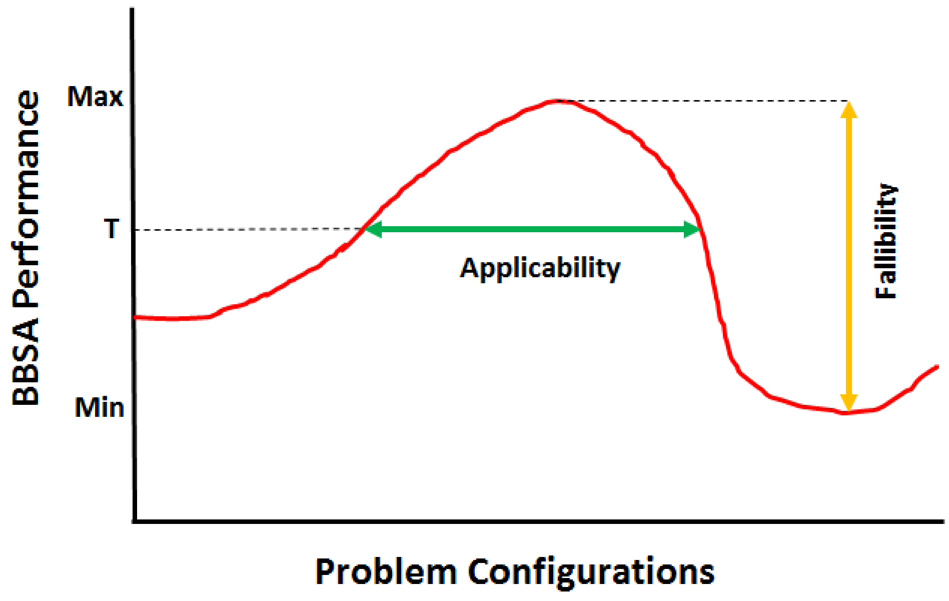

Figure 1.1: This figure demonstrates the concepts of Applicability and Fallibility. Applicability is the proportion of the problem configuration space that a BBSA can perform higher than a given threshold value. Fallibility is the difference between the highest and lowest performing problem configurations.

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 parameters [4] or adaptively selecting which of a pre-defined set of operators to use and in which order [5]. 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 [6, 7, 8]. 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) [9] employs a grammar to describe structure, but is constrained to the canonical EA model. In later work [21], 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 [10, 11]. The primary effort has been to create improved EA variation operators [10, 11, 12, 13]. Some work has been done on evolving EA selection operators [14, 15].

Burke et al. described a high-level approach to evolving heuristics [22]. That approach was extended to evolve entire BBSAs of indiscriminate type [19]. 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 3.2 and 3.3.

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.

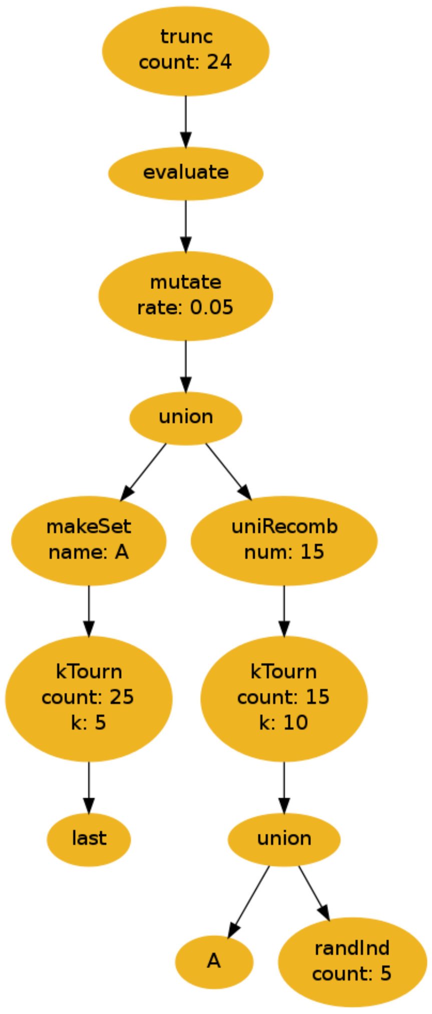

Figure 3.2: Example Parse Tree

```
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 3.3: Example Parse Tree Generated Code

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 [16] 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.

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 sub-tree 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 [21]. 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 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 [19] is the problem of over-specialization when training on a single problem configuration of a given problem class. Following the approach suggested in [19], 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 3.2 and Figure 3.3. This verification was employed for the testing of the BBSAs in all experiments.

4 EXPERIMENTS

To demonstrate that the addition of multi-sampling 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.

The classic Deceptive Trap problem [17] 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 [19], 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 4.1. 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 4.2. 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

Table 4.1: 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.

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

selection. The EA parameter settings are summarized in Table 4.3. These values were selected to be similar to those in [19] with minor hand tuning to perform well with the first problem configuration shown in Table 4.1.

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 BBSA evaluation is twenty-five runs. All

Table 4.2: Problem Configurations that were used to test the robustness of the BBSA.

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 4.3: EA Configurations

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

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

Table 4.4: GP Configurations

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

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 4.4. 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 4.5.

Table 4.5: Black-Box Search Algorithm Settings

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]

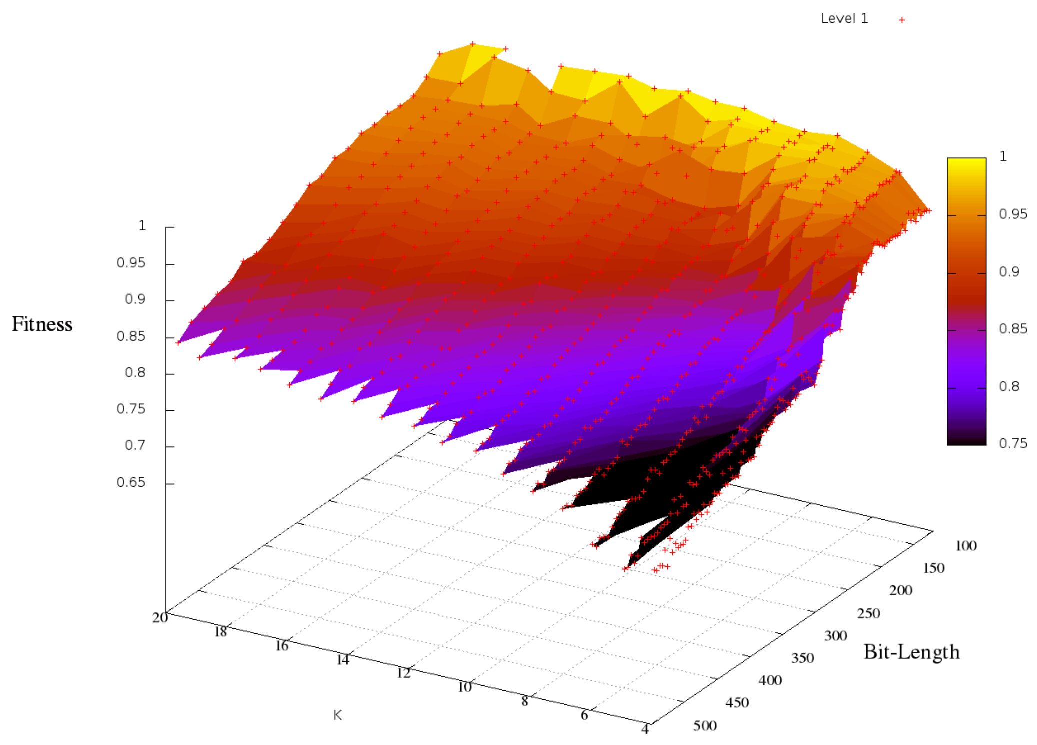

Figure 4.4: The worst BBSA found for multi-sampling level one run on the problem configuration space.

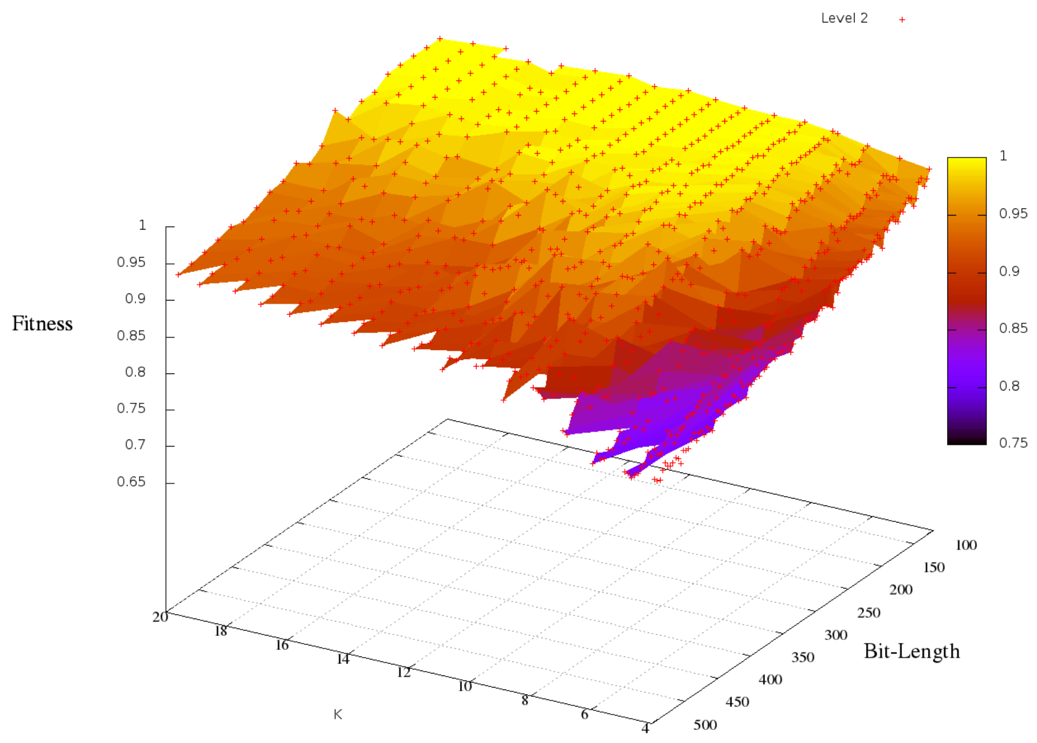

Figure 4.5: The worst BBSA found for multi-sampling level two run on the problem configuration space.

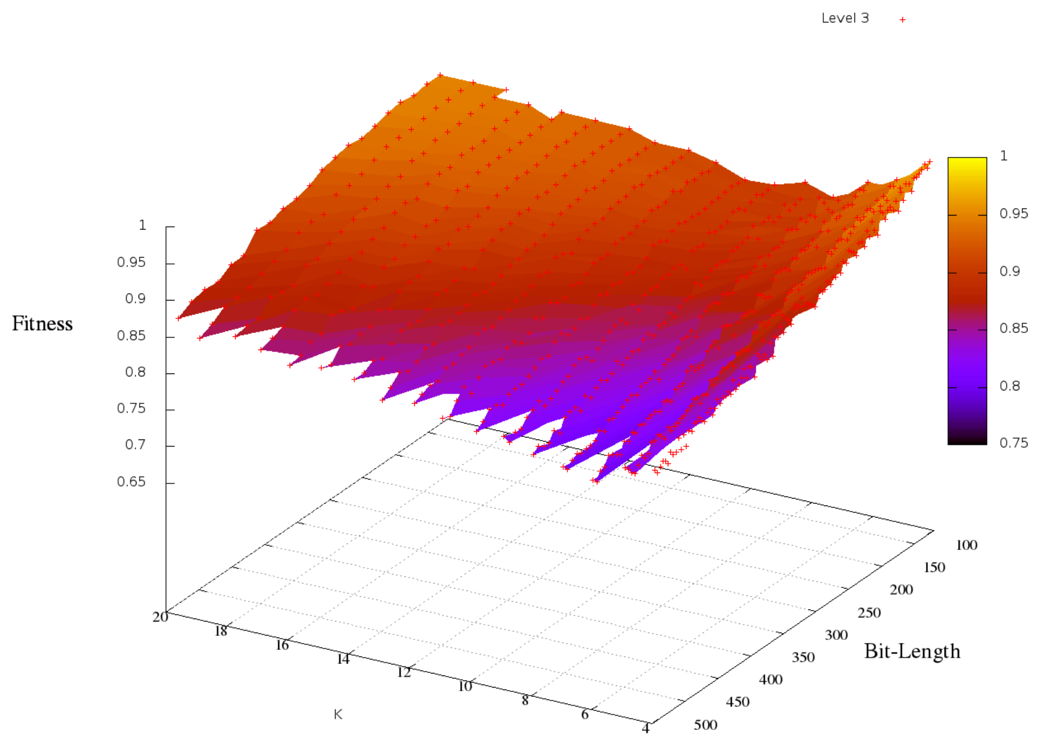

Figure 4.6: The worst BBSA found for multi-sampling level three run on the problem configuration space.

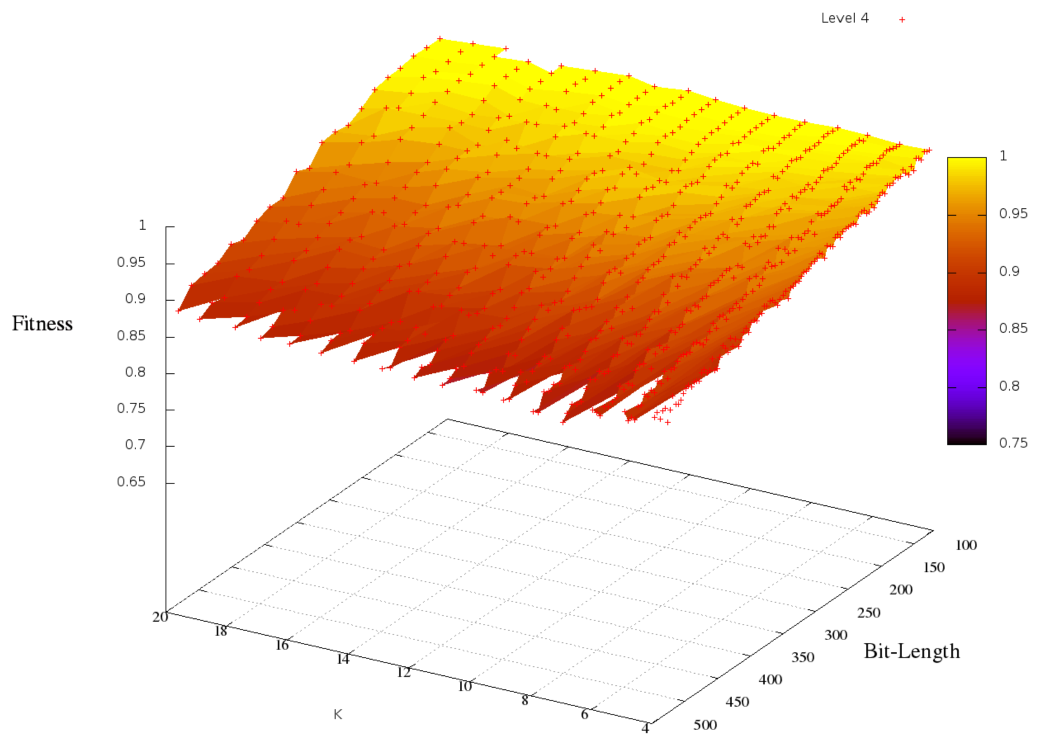

Figure 4.7: The worst BBSA found for multi-sampling level four run on the problem configuration space.

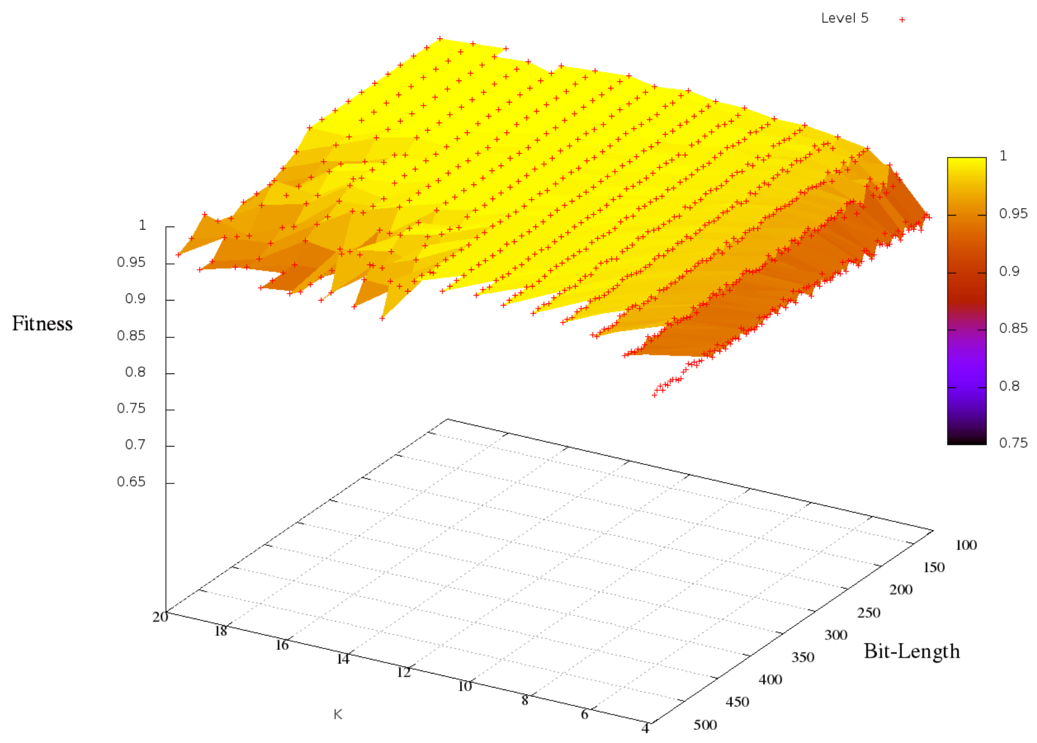

Figure 4.8: The worst BBSA found for multi-sampling level five run on the problem configuration space.

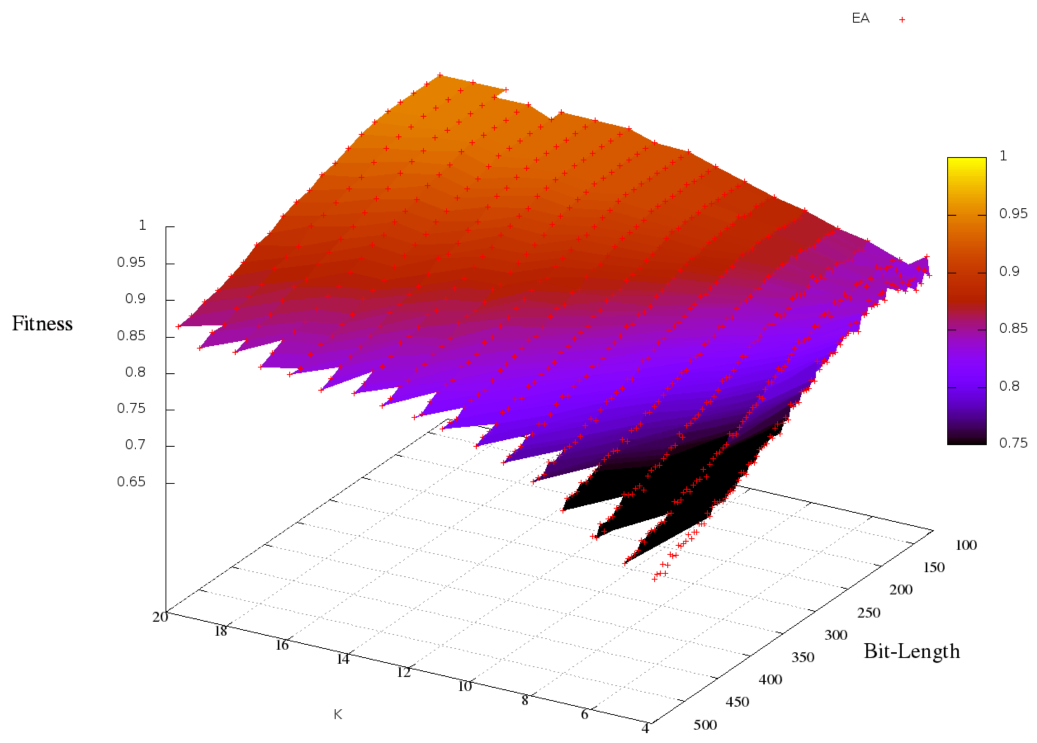

Figure 4.9: A standard EA run on the problem configuration space.

5 RESULTS

The first experiment's results are summarized in Table 5.6. 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 4.2. 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 5.7. 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 perform 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 4.4-4.8 show the least robust BBSA evolved at each multi sampling level. Figure 4.9 shows the baseline of a standard EA. These plots were generated averaging over five runs on each problem configuration.

Table 5.6: BBSA Experimental Results

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

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

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 would increase. 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 5.7. 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 4.4 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 4.9, 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 4.4-4.8, the variance in performance of the algorithm decreases as the multi-sampling level increases.

It is widely known that training an BBSA on a larger number of training problem configurations will improve 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 will look similar to the results shown in Figure 4.4 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.

However, when multi-sampling is done 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 4.4-4.8, the fallibility decreases as the multi-sample level increase. Note that the training sets that these algorithms were generated on had k from 5 to 7 and a $bit-length$ from 100 to 300 and the problem configuration space shown in figures 4.4-4.8 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 more 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 [19], 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, it a trend in the applicability is not statistically discernible. As is shown in Table 5.6, 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 to 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 5.6.

8 FUTURE WORK

The next step to improve upon the proposed approach is to do 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.

III. HYPER-HEURISTICS: A STUDY ON INCREASING PRIMITIVE-SPACE

Matthew A. Martin, and Daniel R. Tauritz

Natural Computation Laboratory

Department of Computer Science, Missouri University of Science and Technology, Rolla, MO 65409

ABSTRACT

Practitioners often need to solve real world problems for which no custom search algorithms exist. In these cases they tend to use general-purpose solvers that have no guarantee to perform well on their specific problem. The relatively new field of hyper-heuristics provides an alternative to the potential pit-falls of general-purpose solvers, by allowing practitioners to generate a custom algorithm optimized for their problem of interest. Hyper-heuristics are meta-heuristics operating on algorithm space employing targeted primitives to compose algorithms. This paper explores the advantages and disadvantages of expanding a hyper-heuristic's primitive-space with additional primitives. This should allow for an increase in quality of evolved algorithms. However, increasing the search space of a meta-heuristic almost always results in longer time to convergence and lower quality results for the same amount of computational time, but also all too often lower quality results at convergence, potentially making a problem impractical to solve for a practitioner. This paper explores the scalability of hyper-heuristics as the primitive-space is increased, demonstrating

significantly increased quality solutions at convergence with a corresponding increase in convergence time. Additionally, this paper explores the impact that the nature of the added primitives have on the performance of the hyper-heuristic.

1 INTRODUCTION

Practitioners are frequently faced with increasingly complex problems for which no polynomial time, guaranteed optimal solvers exist and for which off-the-shelf general-purpose solvers, whether they be deterministic or stochastic, do not provide satisfactory performance. When these problems need to be repeatedly solved, it may be cost-effective to create a custom algorithm which, unlike general-purpose solvers, does not trade off performance on specific problems for generality. Hyper-heuristics are meta-heuristic algorithms which search algorithm-space employing primitives typically derived from existing algorithms, automating the creation of custom algorithms. The highest possible level primitives are complete algorithms, while the lowest possible level are a Turing-complete set of primitives. The former translates into automated algorithm selection, while the latter results in an intractable search of complete algorithm space (which grows exponentially with the number of operations). In order to minimize the search space, the highest primitive level which is sufficient to represent the optimal custom algorithm is ideal. However, determining that level is an open problem in hyper-heuristics. Additionally, adding primitives to an existing level increases the search space, thus increasing coverage at the expense of computational time.

This paper explores the advantages and disadvantages of increasing the search space of a hyper-heuristic by expanding its primitive space. The study reported here analyzes the performance of a hyper-heuristic, which has been previously demonstrated to produce high-quality Black-Box Search Algorithms (BBSAs) for the Deceptive Trap Problem [19, 20], on a more complex benchmark which has the necessary characteristics in order to reveal nuances in the trade-off between search space size (smaller is preferable) and coverage (larger is preferable).

This paper also examines how the nature of the added primitives impacts the performance of the evolved BBSAs. Two distinct sets of primitives are added to the previously employed set of primitives. One set comprises low-level "statement primitives" in the form of a set of "auxiliary" nodes that control program flow, such as loops and branching statements. The second set comprises "derived primitives" extracted from existing algorithms such as Simulated Annealing and Steepest Ascent Hill-Climber. How the nature of the primitives affects the trade-off between increased search space and higher quality BBSAs is explored.

The goal of this research is to demonstrate that while adding primitives to a hyper-heuristic's primitive space increases the search space, which requires additional time to convergence, it also increases the total number of high-quality algorithms produced, as well as increasing the quality of the best evolvable algorithms.

2 RELATED WORK

Recent efforts have applied hyper-heuristics to problems such as the Timetabling Problem [23], bio-informatics [24], and multi-objective optimization [25]. Much of the previous work on employing evolutionary computing to create improved BBSAs has focused on tuning parameters [4] or adaptively selecting which of a pre-defined set of primitives to use and in which order [5]. The latter employed Multi Expression Programming to evolve how, and in what order, the Evolutionary Algorithm (EA) used selection, mutation, and recombination. This approach used four high level primitives: Initialize, Select, Crossover, and Mutate. These primitives were combined in various ways to evolve a better performing EA. Later this approach was also attempted employing Linear Genetic Programming [6, 7, 8]. While this allowed the EA to identify the best combination of available selection, recombination, and mutation primitives to use for a given problem, it was limited to a predefined structure.

A more recent approach to evolving BBSAs employed Grammatical Evolution (GE) [9] which uses a grammar to describe structure, but was constrained to the primitives of the canonical EA model. In later work [21], 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.

Burke et al. described a high-level approach to evolving heuristics [22]. That approach was extended to evolve entire BBSAs of indiscriminate type [19, 26]. The research in this paper builds upon this work by analyzing the advantages and disadvantages of increasing the primitive-space the hyper-heuristic has access to. This paper will also look at how the nature of the added primitives affects the performance of the hyper-heuristic. This analysis is similar to an effort to determine the effect of varying primitive sets has on the performance of selection hyper-heuristics [27], though expanded to a generic hyper-heuristic.

3 METHODOLOGY

The focus of the research reported in this paper is to demonstrate the ability of hyper-heuristics to scale as the number of primitives available is increased. Increasing the number of primitives available to a hyper-heuristic potentially allows it to create higher quality algorithms and tackle more difficult problems. This section will discuss the base hyper-heuristic employed in the reported experiments along with the expanded set of primitives given to the hyper-heuristic to show its scalability.

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 Genetic Programming (GP) primitives 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 a primitive on the sets(s) and then return a single set of solutions. The parse tree is evaluated 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 3.2.6. Examples of a BBSA represented both as a parse tree and as source code are shown in Figure 3.1 and Figure 3.2 respectively.

3.2. NODES

The trees' non-terminal nodes are primitives extracted from existing algorithms such as Evolutionary Algorithms, Simulated Annealing (SA), and Steepest Ascent Hill-Climbing (SAHC). The nodes are broken down into selection, variation, set-manipulation, terminal, and utility nodes. The following subsections describe the primitives of each type employed in the experiments reported in this paper.

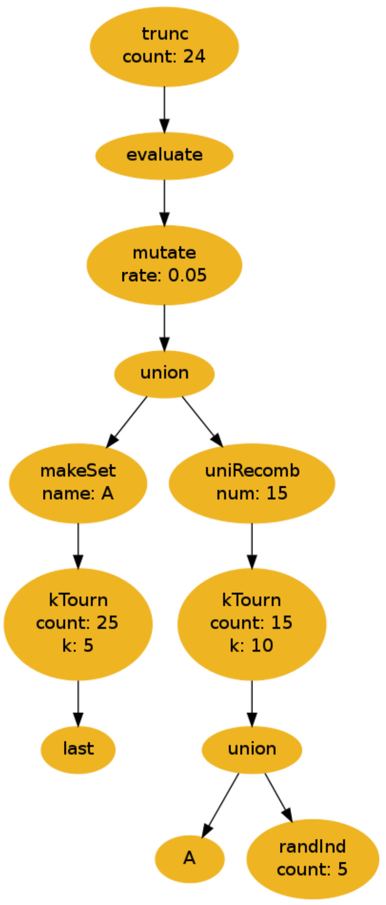

Figure 3.1: Example Parse Tree

```
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 3.2: Example Parse Tree Generated Code

3.2.1. Typing. Many BBSA primitives were designed to perform on a specified number of solutions. Typically in EAs, only two solutions are used for recombination. To allow for nodes to have requirements on the number of solutions that are passed, typing was added to the GP. In addition to the regular sets that have been employed previously, a singleton set type has been added. While the regular set type may be a singleton in some cases, the singleton set type must be a singleton set. Thus if a node needed two solutions, it would have two child nodes that each have the requirement to return the singleton set type. Some nodes can return either the regular set type or the singleton set type depending on which is needed. These situations are described in Section 3.3. In addition to the added flexibility that typing allows, it can also be used to limit the solution set size. Certain primitives can cause the size of the solution sets to increase exponentially if they were applied to a non-singleton set. For instance, if multiple 'Generate Neighborhood' primitives were chained together without a selection primitive between them, the resulting set would grow exponentially. By forcing the 'Generate Neighborhood' node to take a singleton set, the size of the resulting set is limited.

3.2.2. Selection Nodes. Three principal selection primitives were employed in the experiments. The first of these is k-tournament selection with replacement. This node has two parameters, namely k, the tournament size, and *count* which designates the number of solutions passed to the next node. The second selection primitive employed is truncation selection. This primitive takes the *count* best solutions from the set passed to it. The third selection primitive employed is the random subset primitive which takes *count* random solutions from the set passed to it. All of the selection nodes take the regular set type and can either return the singleton set type or the regular set type.

3.2.3. Variation Nodes. The original hyper-heuristic used only three types of variation primitives. The first of which is standard bit-flip mutation. This primitive has a single argument, *rate*, which is the probability that a given bit is flipped. The second original variation primitive is diagonal crossover [16], 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 primitive. The third original variation primitive is standard uniform recombination, which has one child node and returns a regular set type. It has a single argument, *count*, which is the number of solutions that it creates by randomly selecting a parent's gene for each position in the bit string.

The new version of the hyper-heuristic reported in this paper, employs all three the original variation primitives, and adds a fourth one, namely a second uniform recombination primitive which has two child nodes and requires that each of them return a singleton set type. This primitive creates two new solutions using the standard two-parent uniform recombination. Both uniform recombination primitives return a regular set type. The second uniform recombination primitive was added to determine if a typed variation primitive would be more useful than a generic variation primitive.

Additional primitives were added to the set of primitives to analyze how increasing the number of primitives from existing BBSAs affects the performance of the hyper-heuristic. From the SA algorithm two primitives were extracted. The first is the 'tempChange' primitive, which modifies the temperature parameter for the SA algorithm. The temperature parameter is stored at the global level such that all nodes have access to the same temperature. This primitive has a single parameter, *change*, which dictates how the temperature is changed when the node is called. This parameter is a floating point number which is added to, or subtracted from, the current temperature. The initial temperature is set to a constant value for each run of the BBSA. The second primitive from the SA algorithm is named 'tempFlip' which performs the SA variation primitive based on the current global temperature. Both of these nodes can take either a singleton or regular set and return the same set that they are passed. There were also two primitives taken from the SAHC algorithm.

The first is the 'greedyFlip' primitive. This primitive takes a singleton set and performs one step of SAHC by generating the neighborhood of the solution passed in and selecting the best solution from the neighborhood or the original individual and returns it as a singleton set. The second primitive is the 'Generate Neighborhood' function. This function takes a singleton set and generates the neighborhood of that individual and then returns the neighborhood and the original solution as a regular set. The neighborhood is defined by all solutions that vary by exactly one bit.

3.2.4. Utility Nodes. The original hyper-heuristic used only one utility primitive. This was the evaluation node which evaluates all of the solutions that are passed into it. This node can take either a singleton set type or a regular set type and returns the same type that was passed to it.

The following primitives are added to the set of primitives to analyze how increasing the number of utility primitives affects the performance of the hyper-heuristic. The first is the 'for' loop which runs its sub-tree n times, n being one of its parameters, and returns the combination of the results from those iterations. This node requires that its sub-tree return a singleton set type and it returns a regular set type. The second utility primitive is a conditional node called "if converged". If the current run of the BBSA has not found a better solution in *conv* iterations, *conv* being one of its parameters, it will run its right sub-tree, else it will run its left sub-tree. This node also has the option to reset the convergence counter to zero giving it the option to be run a single time at convergence. This node can take either the regular set type or the singleton set type and returns a regular set type. The final utility primitive is another conditional node that runs its right sub-tree *chance* percent of the time, *chance* being one of its parameters, and its left sub-tree $1 - chance$ percent of the time. This node can take either the regular set type or the singleton set type and returns a regular set type.

3.2.5. Set-Manipulation Nodes. The experiments reported in this paper employ two distinct set primitives. The first is the union primitive. This node combines the two sets of solutions passed into it and returns it. This node can take either the regular set type or the singleton set type. It always returns a regular set type. The other primitive is the save primitive called “Make Set”. This primitives saves either copies or pointers to the solutions passed into it. This set can be used elsewhere in the algorithm as explained in Section 3.2.6. This node can take either the regular set type or the singleton set type and returns the same type that it was passed.

3.2.6. 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 primitive, 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. Both of these terminal nodes return a regular set type. The terminal that generates a random set of solutions creates a set of n solutions, n being one of its parameters, and returns that to its parent node. This terminal node can return either a singleton set type or a regular set type.

3.3. META-ALGORITHM

GP is employed to meta-evolve the BBSAs. The two primary variation primitives employed are the sub-tree crossover and mutation, altered to make the maximum number of nodes being added a user defined value. Both of these primitives had to be modified to account for the typing that was introduced into the GP. The sub-tree crossover was modified to ensure that the two sub-trees that were crossed over both returned the same type of set. In the rare situation that one tree used only the singleton set type and the other tree used only the regular set type, the alternate mutation described below is used on one of the trees chosen randomly. The sub-tree mutation was altered to ensure that when a node was added that it was guaranteed to have the return type that its parent node needed. Another mutation primitive 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. The alternate mutation primitive is guaranteed not to change the type of a node that returns a singleton set type.

The evaluation time of the evolved BBSAs is tied to the certainty in the fitness of the BBSA as well as the generality of the BBSA. To increase the certainty in the fitness of the BBSA the number of runs must be increased. To reduce the probability of a BBSA over-fitting during evolution, the BBSA must be trained using multiple problem configurations. Thus, to create a better BBSA, more time must be invested in the evaluation of the BBSAs.

This large evaluation time can cause the hyper-heuristic to run extremely slow. To remedy this problem, a Parallel Evolutionary Algorithm (PEA) strategy was adopted to allow for the evaluations to be distributed across multiple machines. To ensure the most efficient use of the computing resources, an Asynchronous PEA was used [28]. The Asynchronous PEA uses a master-slave model in which the master node generates new BBSAs to be evaluated and the slave nodes evaluate those BBSAs. Using this Asynchronous PEA the speed-up granted from the additional CPUs is near linear [28].

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 on each of a set of problem configurations. 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 all of the runs.

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. 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. If this iteration limit were not put in place, it would take BBSAs with very low evaluations per iteration much longer to be evaluated. The third method terminates algorithms which have converged based on not having improved in i iterations. Finally, if the algorithm requires more than t seconds it is terminated and given no fitness. This is done to help ensure that algorithms evolved complete their execution in a reasonable amount of time.

3.4. EXTERNAL VERIFICATION

To ensure 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 stand-alone BBSA. This is done to verify external to the hyper-heuristic system employed, that the performance that the GP reports for a given BBSA is accurate. The generated code is used in all of the experiments to ensure unbiased execution of the BBSAs. An example of a parse tree and pseudo-code generated can be found in Figure 3.1 and Figure 3.2 respectively. This verification was employed for the testing of the BBSAs in all experiments.

Table 3.1: Primitive Breakdown

Base Primitives	**+Algorithms**	**+Utility**	**Full**
Bit-Flip Mutation	**Base Primitives**	**Base Primitives**	**Base Primitives**
Uniform Recombination	Change Temperature	For Loop	**+Algorithms**
Uniform Recombination(Typed)	SA Variation	If Converge	**+Utility**
Diagonal Recombination	Greedy Flip	Left or Right	
Union	Generate Neighborhood		
Make Set			
k-Tournament Selection			
Truncation Selection			
Random Subset			
Evaluation Node			
Random Individual Terminal			
‘Last’ set Terminal			
Persistent set Terminal			

4 EXPERIMENTS

To analyze how the addition of more primitives affects the performance of the hyper-heuristic, four sets of experiments were performed. The first ran the base hyper-heuristic without the addition of any primitives. The second ran the hyper-heuristic with the addition of the nodes extracted from the SA and SAHC algorithms. The third ran the hyper-heuristic with the addition of the utility primitives. The fourth ran the hyper-heuristic with the addition of all of the new primitives. A summary of the primitives that are included in each of the experiments can be seen in Table 3.1

The data used to determine the presence of these characteristics was gathered from running the single and multi-objective algorithms 30 times each. All four sets of experiments were run using three different sets of three instances of the NK-Landscapes benchmark problem [29] each. The parameters of these three sets can be seen in Table 4.2. These parameters were chosen to be consistent with a recent publication using NK-Landscapes [30]. The data used to analyze the scalability of this hyper-heuristic was gathered by running each problem configuration 10 times. Once all 10 runs were completed, external verification was run on the best BBSA produced by each run. During the external verification, each BBSA was run 30 times for 100,000 evaluations or until convergence.

Table 4.2: Problem Configurations

Problem Set	**N**	**K**
Set 1	30	5
Set 2	40	5
Set 3	50	5

All of the experiments were conducted under the same settings. The meta-algorithm was run for 5000 evaluations. The initial population consisted of 100 individuals and each generation 50 new individuals were created. k-tournament selection with replacement and $k = 8$ was employed for parent selection. The sub-tree

Table 4.3: GP Configurations

Parameter	**Value**
Evaluations	5000
Runs per Problem Instance	5
Initial Population	100
Children per Generation	50
k-Tournament	8
Sub-Tree Crossover Probability	47.5%
Sub-Tree Mutation Probability	47.5%
Alternate Mutation Probability	5%
Alternate Mutation Depth	5
Maximum Time(sec)	90
Maximum Iterations	10,000
Maximum Evaluations in BBSA	100,000

crossover and mutation primitives had 30% chance of being used while the alternate mutation had a probability of 40%. The maximum time for the evaluation of a BBSA was 90 seconds, the maximum number of iterations was 10,000, and the maximum number of evaluations in the BBSA was 100,000. The meta-algorithm parameter settings are summarized in Table 4.3. Due to the high computational cost of running hyper-heuristics, only minimal tuning of the meta-algorithm was feasible.

The BBSAs had certain parameters that related to the ranges of the parameters that some nodes have. Each of the integer parameters ranged from 1 to 25, except for the convergence conditional node which ranged from 5 to 25. The bit-flip mutation nodes parameter *rate* ranged from 0 to 1.0. The floating point parameter on the 'tempChange' node ranged from -3.0 to 3.0. The initial population ranged from 1 to 50 solutions. A detailed list of all of the parameter ranges is shown in Table 4.4.

Table 4.4: Black-Box Search Algorithm Settings

Node	**Parameter**	**Range**
N/A	Initial Population	[1,50]
k-Tournament	*k*	[1,25]
*	*count*	[1,25]
Random Subset	*count*	[1,25]
Truncation	*count*	[1,25]
Bit-Flip	*rate*	[0,1]
Uniform Recombination	*count*	[1,25]
Diagonal Recombination	points	[1,25]
Change Temperature	*change*	,[-3,3]
If Converge	*conv*	[5,25]
Left or Right	*rate*	[0,1]
For loop	*iterations*	[1,25]
Random Individuals	*count*	[1,25]

5 RESULTS

The first results gathered were to determine if there was a significant improvement in fitness of the BBSAs when additional operations were added to the hyper-heuristic. To determine this, the Wilcoxon signed-rank test was performed to determine if a statistical difference existed. In all of these tests α was set to be 0.05. The results of these tests can be seen in Table 5.5. This table shows how a given set of primitives compared to another. Each entry is a tuple of symbols that convey the relationship between the performance of the experiments on the three problem configurations ($N = 30$, $N = 40$, $N = 50$). A $+$ symbol indicates that the experiment on the row performed statistically better than the experiment in the column on a given problem configuration. A $-$ symbol indicates that the experiment on the row performed statistically worse than the experiment in the column on a given problem configuration. A $\sim$ symbol indicates that there was no statistical difference between how the two experiments performed. A X indicates that this entry is duplicate information found elsewhere on the table.

Table 5.5: Rank-Sum Results of Experiment Comparison

	Base	**+Utility**	**+Algorithm**
+Utility	(∼,∼,+)	X	X
+Algorithm	(+,+,+)	(+,+,∼)	X
+Full	(+,+,+)	(+,+,∼)	(+,∼,∼)

The box-plots in figures 5.3 through 5.5 provide a visual comparison of the experiments. The impact of the difficulty of the problem configuration on the different experiments is visualized in Figure 5.6. The performance of the hyper-heuristic decreases as N is increased, which is to be expected as increasing N increases the difficulty of the problem configuration.

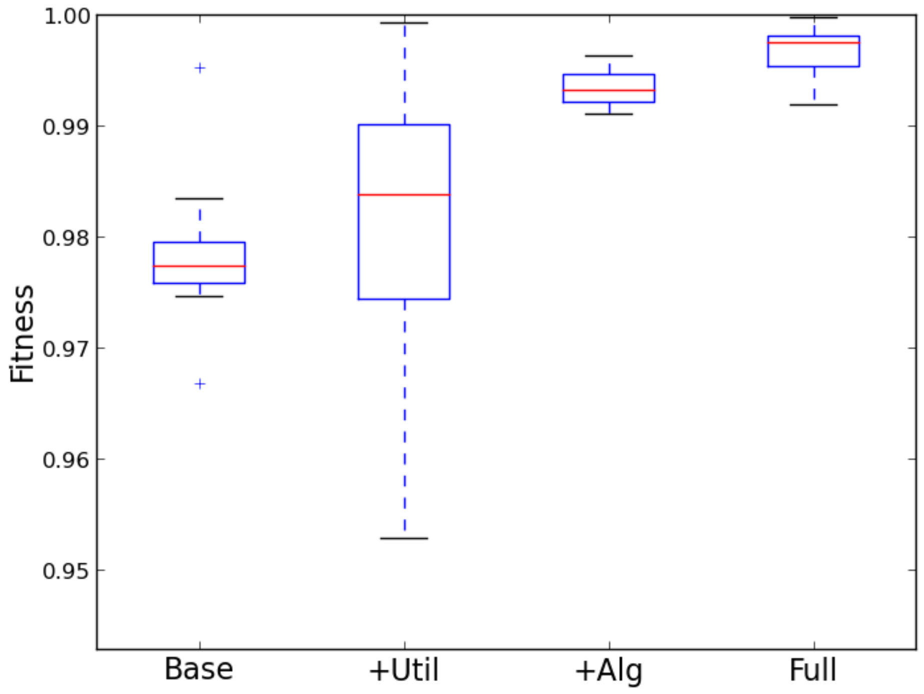

Figure 5.3: This figure shows a box-plot of the four experiments with $n = 30$, where the labels along the x axis correspond to the experiments described in Table 3.1

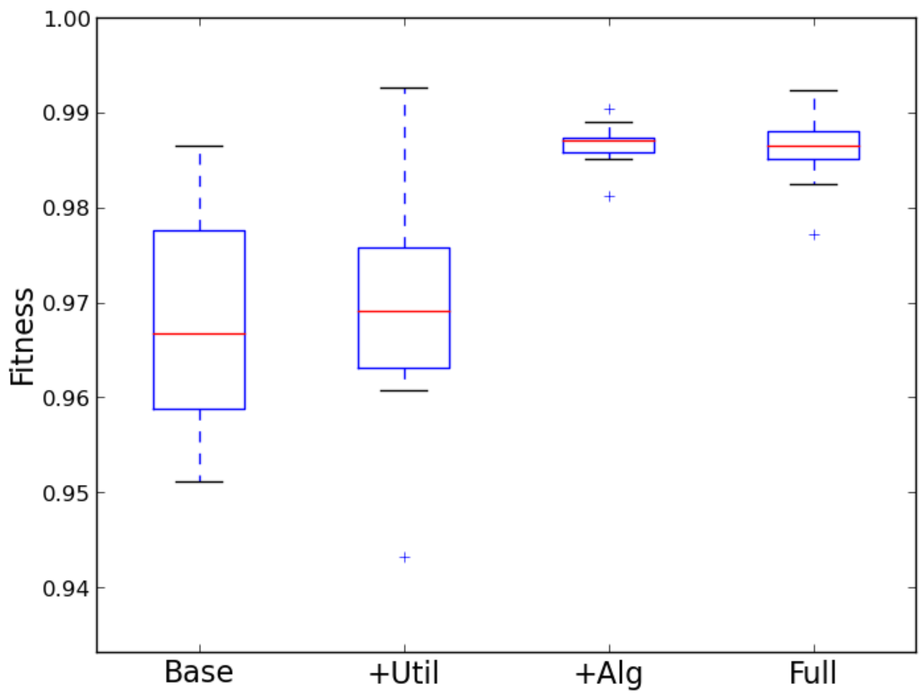

Figure 5.4: This figure shows a box-plot of the four experiments with $n = 40$, where the labels along the x axis correspond to the experiments described in Table 3.1

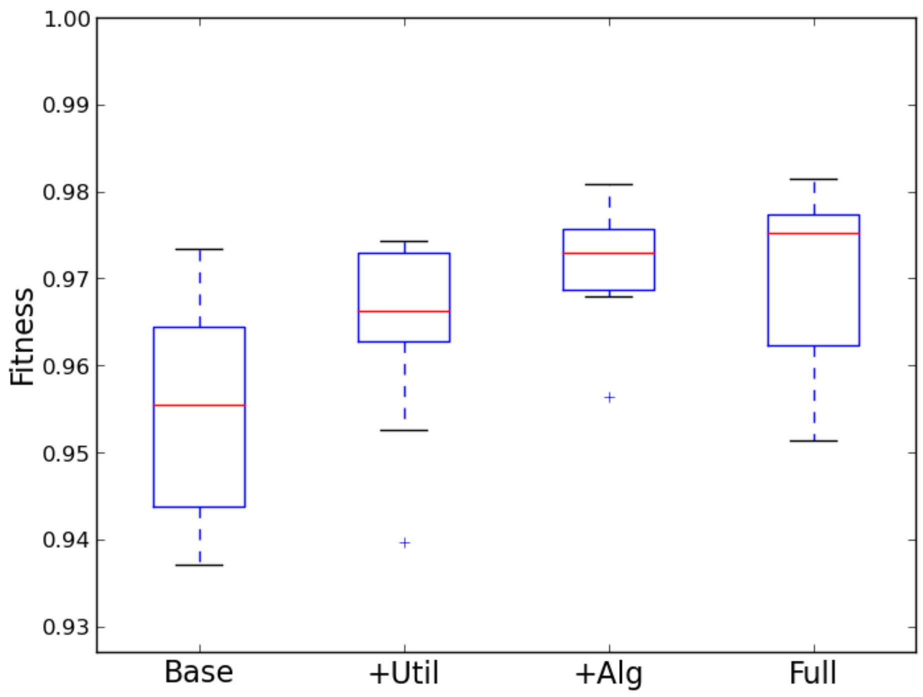

Figure 5.5: This figure shows a box-plot of the four experiments with $n = 50$, where the labels along the x axis correspond to the experiments described in Table 3.1

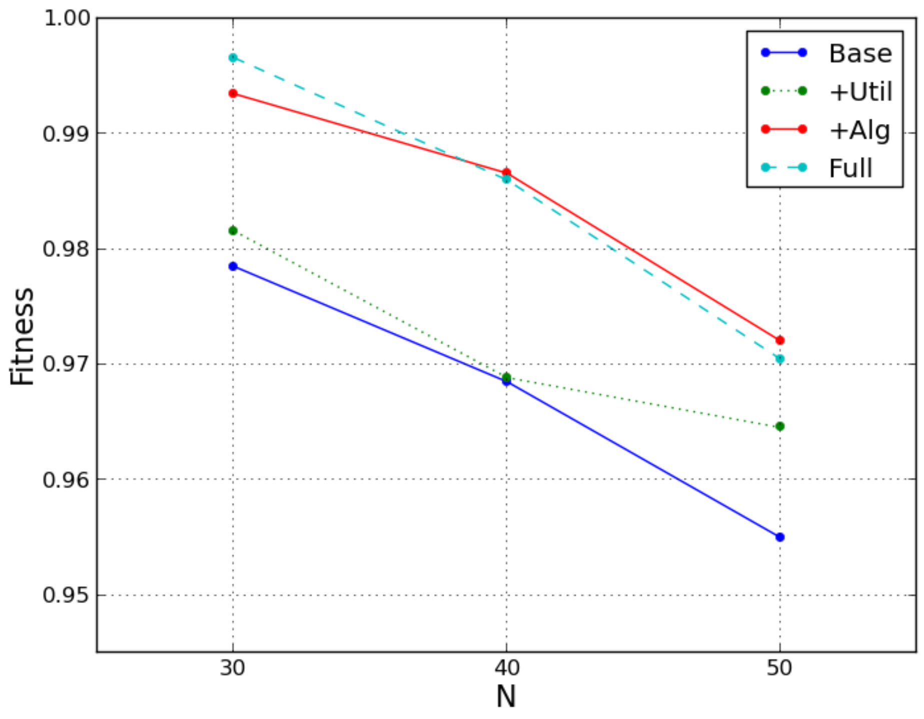

Figure 5.6: Graph of the trend of the four experiments as the problem configurations increases in difficulty

6 DISCUSSION

An important trade-off, when analyzing how the increase in genetic material of a hyper-heuristic, is that between the average performance of the BBSAs and the size of the search space. The the size of the search space can be approximated by variance of the distribution of BBSA fitnesses. The larger the variance is, the larger the search space is. Obviously the larger the mean fitness is the better the hyper-heuristic can perform; however, if the variance of the distribution of BBSAs is large, this indicates that the search space may be much too large to easily traverse.

This assumption can be reinforced by analyzing the differing results between adding utility primitives versus algorithmic primitives. The algorithmic primitives that were included were all unary primitives, and two of the three utility primitives were binary primitives. This means that the increase in search space caused by adding the utility primitives was much more significant than the increase caused by adding the algorithmic primitives. This is supported when analyzing the results of the experiments in figures 5.3 through 5.5. The best BBSA found in the '+Utility' experiments were on par with the best BBSAs found in the '+Algorithm' experiments. However, the difference between best and worst BBSAs is much larger in the '+Utility' experiments likely due to the greater increase in search space. This is reinforced when including the 'Full' experiments in this analysis. The 'Full' experiments had a larger difference between best and worst BBSAs

While the increase in search space caused by the increase in genetic material does increase the difficulty in finding good BBSAs, the quality of the best BBSA found does increase when using more genetic material compared to the 'Base' experiment. In all problem configurations, the best BBSA found in experiments ran with more genetic material performed better than the best BBSA found in the 'Base' experiment. This helps the argument that increasing the genetic material does indeed allow for the hyper-heuristic to find better BBSAs.

The difficulty of the problem configuration did not uniformly affect the performance of the hyper-heuristic. As can be seen in Figure 5.6, as the difficulty of the problem configuration was increased, the performance of each experiment decreased which was expected. However, the performance of the '+Util' experiment did drastically increase in relationship to the other three experiments. This result, however, could not be explained and may be caused solely by the inherent randomness in hyper-heuristics.

7 CONCLUSIONS

This paper is a first investigation of the effects that the amount and nature of genetic material has on the performance of hyper-heuristics. Expanding the amount of genetic material increases the chance that the genetic material of the global optimal solution can be represented. However, this also enlarges the search space which makes it more difficult to find the most optimal representable solution. In the cases examined, this trade-off was beneficial as the hyper-heuristic was able to find more optimal solutions when provided with additional genetic material. If at some point this trade-off no longer is beneficial, then reducing/partitioning the primitives may become useful [31]. It was also found that the arity of the genetic material can have a large impact on the increase in search space. It was seen that when primitives with an arity of two were added, they caused a much larger increase in search space compared to primitives with an arity of one.

The research reported in this paper does show that expanding the amount of genetic material can cause scalability issues for hyper-heuristics, as additional run-time is needed to converge. However, these experiments were run for only 5,000 evaluations, which is very short compared to the typical maximum number of evaluations employed by evolutionary algorithms. This restriction is driven by the high computational cost of evaluating a BBSA. The use of parallel evolutionary algorithms can drastically reduce the total run time, allowing for experimentation with higher numbers of evaluations.

8 FUTURE WORK

This paper has demonstrated the limitations of scaling the genetic material in hyper-heuristics. The next step to better analyze these limitations is to do an in depth study on how much longer hyper-heuristics need to be run to yield converging results. However, if the results converge on non-optimal solutions, then the focus should shift to increasing diversity. Other paths of research include a methodology for creating lower level primitives from existing primitives. In this paper, primitives were extracted from EAs, Simulated Annealing, and Steepest Ascent Hill-Climbers. The same process of extracting primitives can be applied to other algorithms as well as the primitives that we have already extracted. This process could be continued until it yielded a Turing-Complete set of primitives which could then create all BBSAs. However, the research in this paper shows that as the primitive set gets larger, it becomes more difficult to find high quality BBSAs. The goal then would be to identify the set of primitives with the optimal balance between coverage of high quality BBSAs and minimizing the primitive search space.

SECTION

2. CONCLUSIONS

This thesis introduces a Genetic Programming based Hyper-heuristic that can evolve BBSAs that outperform canonical BBSAs for a given problem class. While this hyper-heuristic uses the same primitives as these canonical BBSAs, it has the ability to use these primitives much more effectively than the canonical BBSAs do and bears little resemblance to them. By removing the human bias of attempting to fit a BBSA into a category such as Genetic Algorithm, Evolutionary Programming, or Evolutionary Strategies, the hyper-heuristic is able to create BBSAs that can out-perform algorithms that fit nicely into these categories.

The multi-sampling method applied to this hyper-heuristic can drastically increase the robustness of the evolved BBSAs. This method is shown to not only generate BBSAs that generalize to the problem configuration space close to the training problem configurations, but to a much wider area of the problem configuration landscape. This multi-sampling method, however, comes with an increase in computational time necessary to evolve high-performance BBSAs due to needing to evaluate the BBSAs on multiple problem configurations during training. There is a trade-off that exists between computational time in training and the probability of robustness in the evolved BBSAs.

It is found that the amount and nature of the genetic material that a hyper-heuristic has available to it can drastically affect the performance of the resulting BBSAs.

Expanding the amount of genetic material (i.e., the set of primitives) increases the chance that the optimal solution can be represented, but also causes the search space to increase correspondingly. Given unlimited time and full coverage variation operations, this will in crease the expected performance of the evolved BBSAs. However, in practice time is severely limited, and therefore there is typically a trade-off point beyond which it is not beneficial to expand the genetic material. The parity of the genetic material can have a large impact on the increase in search space. It was seen that when primitives with a parity of two were added, the search space increased much more than when compared to primitives with a parity of one.

The results in this thesis show that the use of hyper-heuristics is a feasible mean to create novel, high-performing BBSAs. When using hyper-heuristics the practitioner needs no domain specific knowledge of the problem. The hyper-heuristic also has no biases that would cause it to develop a sub-par BBSA as a human developer might. This lack of bias truly allows it to develop novel BBSAs that can out-perform canonical algorithms that are widely used.

2.1 LIMITATIONS

While this thesis has shown that hyper-heuristics can perform well on the problems that were presented, hyper-heuristics have many limitations that presently prevent them from creating human competitive BBSAs. One of the major limitations of hyper-heuristics is their run-time. Hyper-heuristics can take an extremely long time to run. This is caused by the necessity to run each BBSA many times to determine the quality of the BBSA. Another limitation of hyper-heuristics is the availability of genetic material for them to use. All of the genetic material that hyper-heuristics use must be manually extracted from existing algorithms. While extracting high-level primitives can be done more easily, extracting low-level primitives can be extremely difficult for practitioners. Thirdly, hyper-heuristics can give no guarantee that the performance of the resulting BBSAs be high quality. While hyper-heuristics will always try to optimize the BBSAs it is evolving, it can not guarantee a priori if it can represent a high-quality BBSA. Finally, the hyper-heuristic presented in this thesis does not take into account the number of evaluations necessary to find a high-quality solution. This prevents the hyper-heuristic from creating truly human competitive BBSAs.

2.2 FUTURE WORK

There is much research that can be done to help fix or mitigate the limitations of this hyper-heuristic in its current state. To help mitigate the problem of long run-times, hyper-heuristics can be parallelized. The third paper presented in this thesis drastically improved the run-time of the hyper-heuristic through asynchronous parallelization. This asynchronous parallelization may have an affect on the population due to the different population mechanics when compared to synchronous parlleliza-tion techniques and analysis should be done of this affect. Another improvement that could be done is developing a methodology for automatically creating primitives from existing algorithms and primitives. This process would automatically decompose high-level primitives to obtain lower-level primitives and then recompose these primitives into novel higher-level primitives. This would reduce the strain on practitioners and allow hyper-heuristics to more easily be used. To allow hyper-heuristic to develop BBSAs that are human competitive, the hyper-heuristic would need to be multi-objective where it would attempt to maximize the solution quality as well as minimizing the number of evaluations necessary to find these high-quality solutions.

BIBLIOGRAPHY

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

[2] A.E. Eiben and S.K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

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

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

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

[6] Laura Dioşan and Mihai Oltean. Evolutionary Design of Evolutionary Algorithms. *Genetic Programming and Evolvable Machines*, 10(3):263–306, September 2009.

[7] Laura Silvia Diosan and Mihai 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.

[8] Mihai Oltean. Evolving Evolutionary Algorithms Using Linear Genetic Programming. *Evol. Comput.*, 13(3):387–410, Sept. 2005.

[9] Nuno Lourenço, Francisco Pereira, and Ernesto 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.

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

[11] Bruce 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.

[12] Brian W. Goldman and Daniel 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.

[13] John R. Woodward and Jerry 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.

[14] 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.

[15] John Robert Woodward and Jerry 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.

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

[17] 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.

[18] 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.

[19] Matthew A. Martin and Daniel 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.

[20] Matthew A. Martin and Daniel 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.

[21] Nuno Lourenço, Francisco Baptista Pereira, and Ernesto 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.

[22] Edmund K. Burke, Mathew R. Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R. Woodward. Exploring Hyper-heuristic Methodologies with Genetic Programming. In ChristineL. Mumford and LakhmiC. Jain, editors, *Computational Intelligence*, volume 1 of *Intelligent Systems Reference Library*, pages 177–201. Springer, 2009.

[23] Jorge A Soria-Alcaraz, Gabriela Ochoa, Jerry Swan, Martin Carpio, Hector Puga, and Edmund K Burke. Effective learning hyper-heuristics for the course timetabling problem. *European Journal of Operational Research*, 238(1):77–86, 2014.

[24] Aleksandra Swiercz, Edmund K Burke, Mateusz Cichenski, Grzegorz Pawlak, Sanja Petrovic, Tomasz Zurkowski, and Jacek Blazewicz. Unified encoding for hyper-heuristics with application to bioinformatics. *Central European Journal of Operations Research*, 22(3):567–589, 2014.

[25] Mashael Maashi, Graham Kendall, and Ender Özcan. Choice function based hyper-heuristics for multi-objective optimization. *Applied Soft Computing*, 28:312–326, 2015.

[26] Matthew A. Martin and Daniel R. Tauritz. A Problem Configuration Study of the Robustness of a Black-box Search Algorithm Hyper-heuristic. In *Proceedings of the 2014 Conference Companion on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1389–1396, New York, NY, USA, 2014. ACM.

[27] M Mısır, Katja Verbeeck, Patrick De Causmaecker, and G Vanden Berghe. The effect of the set of low-level heuristics on the performance of selection hyper-heuristics. In *Parallel Problem Solving from Nature-PPSN XII*, pages 408–417. Springer, 2012.

[28] Matthew A. Martin, Alex R. Bertels, and Daniel R. Tauritz. Asynchronous Parallel Evolutionary Algorithms: Leveraging Heterogeneous Fitness Evaluation Times for Scalability and Elitist Parsimony Pressure. In *Proceeding of the Seventeenth Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '14 Companion, New York, NY, USA, 2015. ACM.

[29] Stuart A Kauffman and Edward D Weinberger. The NK model of rugged fitness landscapes and its application to maturation of the immune response. *Journal of theoretical biology*, 141(2):211–245, 1989.

[30] Brian W. Goldman and Daniel R. Tauritz. Supportive Coevolution. In *Proceedings of GECCO 2012 Companion - Genetic And Evolutionary Computation Conference*, GECCO Companion '12, pages 59–66, New York, NY, USA, 2012. ACM.

[31] Stephen Remde, Peter Cowling, Keshav Dahal, Nic Colledge, and Evgeny Selensky. An empirical study of hyperheuristics for managing very large sets of low level heuristics. *Journal of the operational research society*, 63(3):392–405, 2012.

VITA

Matthew Allen Martin grew up in Carthage, Missouri. He graduated from Carthage Senior High School. From Fall of 2009 to Spring of 2013, he attended Missouri University of Science and Technology to earn a Bachelor of Science degree in Computer Science. In the summer of 2010 Matthew worked for Sandia National Laboratories as a technical intern at their Livermore, California location. In the summers of 2011 and 2012, Matthew worked for Sandia National Laboratories at their Albuquerque, New Mexico location in the Center for Analysis Systems and Applications internship program. He was then accepted into Sandia's Critical Skills Master's Program that lasted from the Summer of 2013 to Spring of 2015 during which he worked at Sandia in the summers. With funding from Sandia's Critical Skills Master's Program, Matthew earned his Master of Science degree in Computer Science from Missouri University of Science and Technology in August of 2015 and performed the research upon which the three papers in this thesis were based.