Hyper-Heuristics: A Study On Increasing Primitive-Space

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

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

ABSTRACT

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.

Categories and Subject Descriptors

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

Keywords

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

GECCO '15, July 11 - 15, 2015, Madrid, Spain
© 2015 ACM. ISBN 978-1-4503-3488-4/15/07...$15.00
DOI: http://dx.doi.org/10.1145/2739482.2768457

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 [11, 13], 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 [19], bio-informatics [20], and multi-objective optimization [9]. Much of the previous work on employing evolutionary computing to create improved BBSAs has focused on tuning parameters [18] or adaptively selecting which of a pre-defined set of primitives to use and in which order [16]. 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 [2, 3, 15]. 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) [7] which uses a grammar to describe structure, but was constrained to the primitives of the canonical EA model. In later work [8], 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 [1]. That approach was extended to evolve entire BBSAs of indiscriminate type [11, 12]. 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 [14], 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 1 and Figure 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.

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

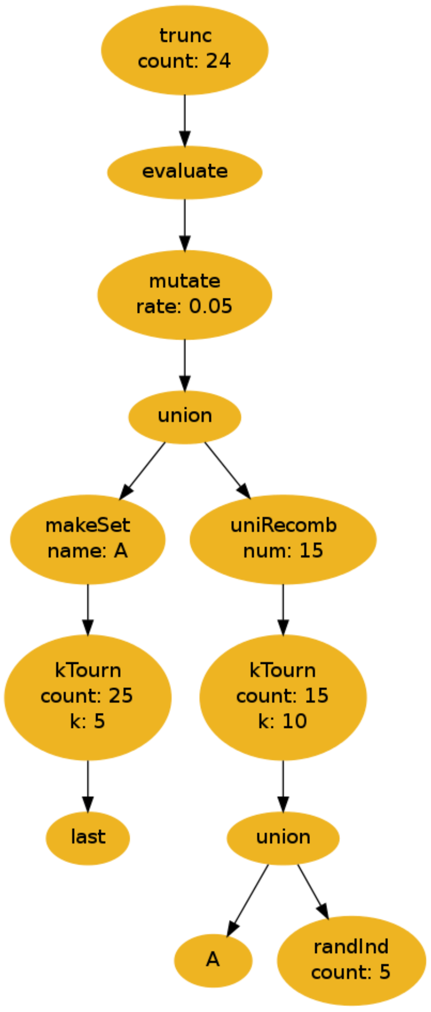

Figure 1: Example Parse Tree

```
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 2: Example Parse Tree Generated Code

original variation primitive is diagonal crossover [4], 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 [10]. 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 [10].

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

Table 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			

Table 2: Problem Configurations

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

pseudo-code generated can be found in Figure 1 and Figure 2 respectively. This verification was employed for the testing of the BBSAs in all experiments.

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 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 [6] each. The parameters of these three sets can be seen in Table 2. These parameters were chosen to be consistent with a recent publication using NK-Landscapes [5]. 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.

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 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 3. Due to the high computational cost of running hyper-heuristics, only minimal tuning of the meta-algorithm was feasible.

Table 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

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.

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

Table 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*	[25,50]
Left or Right	*rate*	[0,1]
For loop	*iterations*	[1,25]
Random Individuals	*count*	[1,25]

Table 5: Rank-Sum Results of Experiment Comparison

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

be seen in Table 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.

The box-plots in figures 3 through 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 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.

6. DISCUSSION

An important trade-off, when analyzing 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 size of the search space can be approximated by the variance of the distribution of BBSA fitnesses. The larger the variance is, typically 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 al-

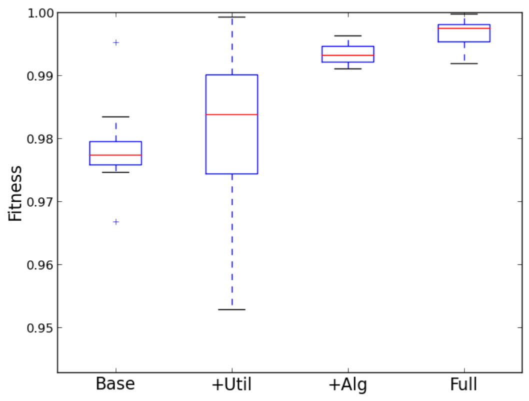

Figure 3: This figure shows a box-plot of the four experiments with $n = 30$, where the labels along the x axis correspond to the experiments described in 1

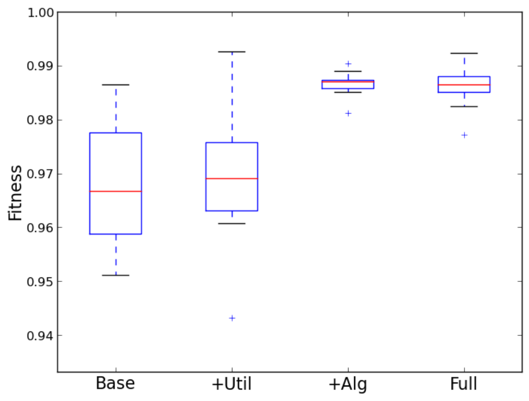

Figure 4: This figure shows a box-plot of the four experiments with $n = 40$, where the labels along the x axis correspond to the experiments described in 1

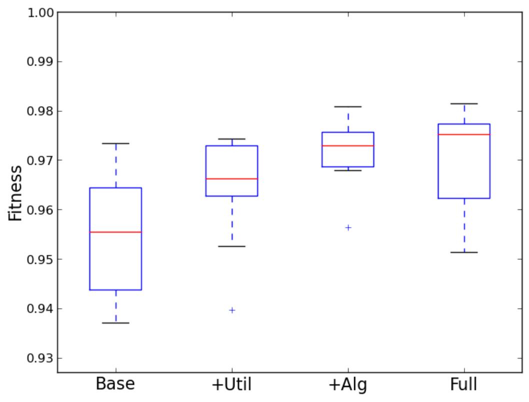

Figure 5: This figure shows a box-plot of the four experiments with $n = 50$, where the labels along the x axis correspond to the experiments described in 1

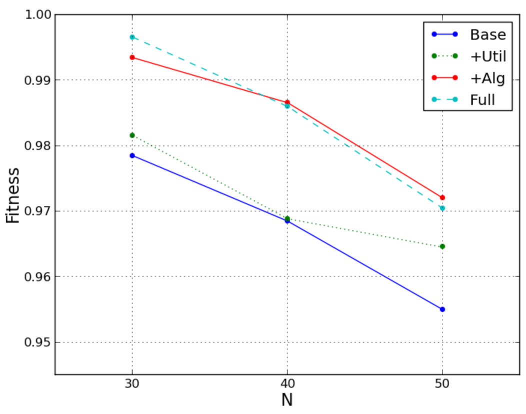

Figure 6: Graph of the trend of the four experiments as the problem configurations increases in difficulty

gorithmic 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 3 through 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 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 [17]. 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.

Acknowledgment

This work was generously supported by Sandia National Laboratories.

9. REFERENCES

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

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

[3] L. S. Diosan and M. Oltean. Evolving Evolutionary Algorithms Using Evolutionary Algorithms. In

Proceedings of GECCO 2007 - Genetic And Evolutionary Computation Conference, GECCO '07, pages 2442–2449, New York, NY, USA, 2007. ACM.

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

[5] B. W. Goldman and D. 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.

[6] S. A. Kauffman and E. 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.

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

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

[9] M. Maashi, G. Kendall, and E. Özcan. Choice function based hyper-heuristics for multi-objective optimization. *Applied Soft Computing*, 28:312–326, Mar. 2015.

[10] M. A. Martin, A. R. Bertels, and D. 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. ACM, 2015.

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

[12] M. A. Martin and D. 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. ACM, 2014.

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

[14] M. Mısır, K. Verbeeck, P. De Causmaecker, and G. V. 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.

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

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

[17] S. Remde, P. Cowling, K. Dahal, N. Colledge, and E. 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.

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

[19] J. A. Soria-Alcaraz, G. Ochoa, J. Swan, M. Carpio, H. Puga, and E. K. Burke. Effective learning hyper-heuristics for the course timetabling problem. *European Journal of Operational Research*, 238(1):77–86, 2014.

[20] A. Swiercz, E. K. Burke, M. Cichenski, G. Pawlak, S. Petrovic, T. Zurkowski, and J. Blazewicz. Unified encoding for hyper-heuristics with application to bioinformatics. *Central European Journal of Operations Research*, 22(3):567–589, 2014.