Outlining a Practitioner's Guide to Fitness Function Design

Josh Wilkerson – jlwc46@mst.edu
Daniel Tauritz – tauritzd@mst.edu

Proceedings of the 4^{th} Annual ISC Research Symposium ISCRS 2010

***Abstract*— Fitness function design is often both a design and performance bottleneck for evolutionary algorithms. The fitness function for a given problem is directly related to the specifications for that problem. In this paper we outline a guide for transforming problem specifications into a fitness function. The target audience for this guide are non-expert practitioners. The goal of the research presented in this paper is to investigate and formalize the fitness function generation process that expert developers go through and in doing so make fitness function design less of a bottleneck. Solution requirements in the problem specifications are identified and classified; then an appropriate fitness function component is generated based on its classifications. The fitness function components are then combined to yield a fitness function for the problem in question. Illustrative examples utilizing the guide are presented. Also, the performance of a guide generated fitness function is compared to that of an expert designed fitness function demonstrating the competitiveness of the guide generated fitness function.**

I. Introduction

THE design of an effective fitness function is often difficult (even for experienced designers [1]). Evidence of this difficulty can be seen in publications like [2], [3], [4], which are by researchers who use Evolutionary Algorithms (EAs) but have experienced difficulty in the design of an effective fitness function. The goal of this research is to create a guide to assist non-expert practitioners in the design of high performance fitness functions.

The formalization of such a fitness function design procedure is novel, to the best knowledge of the authors. Many researchers informally develop application specific methods for fitness function design, but do not seem to generalize these methods. Research into the evolution of fitness functions (e.g., [5], [6]) has some relevance to fitness function design, though much of that research is also driven by application specific goals.

The goal of a fitness function is to guide the evolutionary process through the problem environment to an optimal solution. The effectiveness of the fitness function used by an EA is directly related to the effectiveness of the EA as a whole. The fitness function is the primary point in the EA where the problem specifications are enforced. For this reason, the problem specifications are an ideal location to start the fitness function design process. The presented guide starts by identifying the requirements that define a solution to the problem outlined in the specifications. Each requirement is classified using the provided taxonomy and then a fitness function component is generated (based on the classifications applied) that is responsible for enforcing the requirement in the fitness function. The fitness function components are finally combined into a fitness function for the problem, which can take any form desired by the user (e.g., composite function, multi objective fitness function, etc.).

A number of approaches have been investigated in related research to attempt to classify fitness functions or determine their effectiveness for a particular problem. [7] presents a survey of fitness function classification techniques for EAs using binary string representation. Many of the classified techniques are focussed on calculating GA-hardness, i.e., epistasis variance, fitness distance correlation, and bit wise epistasis. The survey points out where the presented methods have critical flaws, making them only applicable to specific problems. [8] presents a similar survey for fitness functions used in evolutionary robotics, providing another method for classification. The classes presented, however, are specific to the types of fitness functions used in evolutionary robotics; the majority of fitness functions used in other types of EAs would all fall into a single of the presented classifications.

II. Fitness Function Generation

The proposed method for fitness function generation can be broken up into a series of generalized steps. The first step is to identify the individual requirements indicated by the problem specifications, i.e., the problem requirements such that if a candidate solution fully satisfies each requirement then it is a valid solution to the problem. The second step is to classify each requirement according to a taxonomy. Each classification implies information regarding the nature and design of an associated fitness function component. The last step is to use the classifications for each requirement to generate an appropriate fitness function component using the information from the requirement's classification. These fitness function components are then composited together either into a single fitness function or implemented as a new dimension in a multi-objective fitness function.

Through this discussion, the classic Traveling Salesman Problem (TSP) [9] will be employed as running example to help illustrate the various classifications in the taxonomy. The specifications for the TSP are: given an adjacency matrix A, find the shortest Hamiltonian circuit of the graph represented by A.

A. Addressing Problem Specifications

The purpose of the fitness function is to guide the evolutionary process through the problem space, ultimately arriving at a valid solution. In order to generate an effective fitness function, we must first define what a valid solution to the problem is. If properly indicated, the problem specifications should have this information in some form, which means that the first step is to identify each solution requirement from the problem specifications.

Each identified requirement should address a single aspect of the solution, as doing so will ultimately ease the construction of a fitness function. The fitness function must address every aspect of a problem's defined solution in order to properly guide the evolutionary process. So, each solution requirement obtained from the problem specifications will ultimately yield a fitness function component. Once all requirements have been classified,

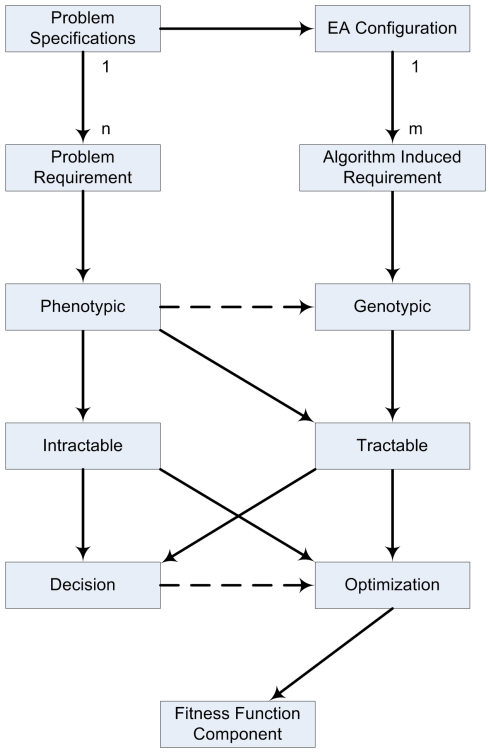

Fig. 1. Requirement Classification Taxonomy

the resulting fitness function components are combined into the fitness function. This ensures that every requirement set forth by the problem specifications is considered in the fitness function.

The TSP has two apparent requirements for a candidate solution (i.e., a path through the graph represented by A) to be valid:

1) The path must be a Hamiltonian cicuit (i.e., it must visit all nodes without revisiting and return to the starting node).
2) The Hamiltonian circuit must be the shortest possible.

B. Classification Taxonomy

The next step is to begin bridging the gap between written problem requirements and a fitness function. The method proposed addresses this task by defining a taxonomy, which classifies the problem requirements and in doing so provides information on the nature of the fitness function. The taxonomy is shown in Fig. 1. The starting point is the problem specifications; determining the solution requirements is the action going from *Problem Specifications* to *Problem Requirement*.

Next, an appropriate solution representation and EA configuration must be determined to solve the problem, which are both based on the problem specifications. There may be solution requirements that arise based on the algorithm selected, due solely to the selected EA configuration and as such cannot be expected to be included in the problem specifications, since for a given problem there may be a number of possible algorithms to use. As example of an *algorithm induced requirement*, suppose that for a given problem we choose to employ Genetic Programming (GP). An issue that is typical to GP is tree bloat in the evolving candidate solutions. So in response to this, suppose we want our fitness function to penalize bloating by promoting smaller tree structure. The fitness function component responsible for this requirement has nothing to do with the problem being solved, just the algorithm that was selected to solve the problem. This illustrates the need for *algorithm induced requirements*.

For the TSP example, assume a standard EA is used and the candidate solutions is an array of graph node identifiers, indicating the order in which nodes are visited. For example, if a candidate solution *C=[a,b,c,a]* then the path that C takes is node a, node b, node c, then back to node a. To make this example simpler, assume that the EA operators will only generate valid paths through the graph, so the fitness function does not need to check path validity. There are no obvious *algorithm induced requirements* for this EA configuration and representation, so this example will only have problem based requirements.

1) Phenotypic and Genotypic: The first level of classification addresses how the requirement will be assessed. A phenotypic requirement is based on some aspect of a candidate solution's performance in the problem environment, independent of the candidate's genetic representation. Conversely, a genotypic requirement is based on some aspect of a candidate solution's genetic structure. Since the problem specifications are stated independent of a specific algorithm it is impossible for a problem requirement to be genotypic, since genotypic requirements are based on an individual's representation. Similarly, algorithm requirements are based solely on the algorithm selected and are independent of the specific problem being addressed. As such, problem requirements are always phenotypic and algorithm requirements are always genotypic. A working hypothesis that is still being investigated is that it is advantageous to convert phenotypic requirements to genotypic, if possible. The reasoning behind this hypothesis is that if the desired candidate solution behavior can be mapped to a specific genetic configuration, then guiding candidate solutions to this configuration will be much easier (as it will be known what genes are responsible for the desired behavior) and convergence will occur much quicker. In Fig. 1 this conversion is shown as the dashed edge from *Phenotypic* to *Genotypic*.

In terms of the fitness function component, the genotypic and phenotypic classifications indicate whether or not input will be required by the component. Genotypic components will not require any input aside from the candidate solution itself, since they are based solely on the genetic structure of the candidate. Phenotypic components will require input from the problem environment, since the component is assessing an aspect of a candidate solution in the context of its problem environment. As such, phenotypic components will require the input of the entire, or a sampling of the, problem environment (which is decided in the second classification, discussed in the next section).

For the TSP example, both requirements from Section II-A are problem requirements, so they fall under *phenotypic* classification. However, the second requirement (i.e., the Hamiltonian circuit must be the shortest possible) can be converted to *genotypic* based on the representation being used for candidate solutions. Instead of counting the number of steps taken when traversing the graph, the number of elements in the candidate solution array can be used to calculate the path length. This conversion will not improve the rate of convergence of the fitness function, but it will speed up the calculation of the component fitness.

2) Tractable and Intractable: The second classification is concerned with the practicality of assessing a given requirement. Essentially this classification determines if the resulting fitness function component will calculate the true fitness value or an approximation to the true fitness value. Suppose for a given

requirement the problem domain is all real numbers; for this requirement it is impossible to calculate the true fitness value, so an approximation must be made by using a sampling from the set of real numbers. This requirement would be classified as *Intractable*. On the other hand, suppose the problem domain for a requirement is a finite graph. This requirement would likely be classified as *Tractable*, since it is feasible to calculate the true fitness by operating on the problem graph. These two examples, though illustrative, do not represent all possible scenarios for this classification. Since the primary concern is practicality, a requirement may be classified as tractable in one case and yet in another the exact same requirement could be classified as intractable. This requirement is very much dependent on the resources available to a specific user.

In Fig. 1 you can see that genotypic requirements can only be classified as tractable. The reasoning behind this restriction is that genotypic requirements are based on the genetic structure of a candidate solution, which implies that the calculation of the true fitness of such a requirement should be feasible as long as the candidate solution representation is practical.

This classification further fleshes out the details of the input to a fitness function component. Phenotypic fitness function components can calculate either the true or approximate fitness for the requirement. If the true fitness is calculated (i.e., the requirement is tractable) then the component will operate on the entire problem domain for the requirement, and thus will need to have access to it. If the component calculates an approximate fitness (i.e., the requirement is intractable) then it will operate on a sample set taken from the problem domain, which means a sampling method must also be decided upon for the fitness function component. In many cases a single sampling function is used for multiple fitness function components (to ensure that each component operates on the same sample set), so this decision may affect more than one component.

In the running TSP example, the first requirement was classified as $phenotypic$ so it can be either $tractable$ or *intractable*. For this example assume that resources are such that it is feasible to navigate a full Hamiltonian circuit of the graph, if such a circuit is discovered. So since the problem space is manageable and navigation of it is feasible, it is possible to calculate the true fitness for the first requirement, thus the requirement is classified as $tractable$. The second TSP requirement was converted to $genotypic$ classification so it will also be classified as $tractable$, based on the prior reasoning in this section.

3) Decision and Optimization: The third classification defines the basic nature of the requirement in question. If a requirement is either satisfied or it is not, then it is a decision requirement. If there are intermediate levels of satisfaction of the requirement, then it is an optimization requirement.

EAs require a gradient in the fitness function in order for the evolutionary process to be effective. If a fitness function is defined as a decision problem then the evolutionary process degenerates into basically a random search. For this reason, any requirement that is classified as a decision requirement must be transformed into an optimization requirement. For example, suppose that a requirement is that the output of a candidate solution is to be in sorted order. Clearly, the output is either sorted or it is not, so the requirement receives a decision requirement. So in order to transform the requirement to optimization, a method is needed for determining how close to being sorted the output is and then use that for the fitness function component. Additionally, the more gradient that each fitness function has the better; i.e., having a single *'partially satisfied'* intermediate level of satisfaction is not going to greatly benefit the evolutionary process. So, some requirements may be classified as optimization, but will still need to be expanded upon in order to generate a more effective fitness function. This classification indicates how the fitness function component is actually going to enforce the requirement that it is based on. As a result, this classification will often require more consideration than that of the other classifications. However, once this classification is made, the user should have a very good idea about how the fitness function component will work, making implementation much simpler.

In the TSP example, the first requirement is stated as a decision problem, i.e., either the path is a Hamiltonian circuit or it is not. So that means that the next step is to decide on a way to convert this requirement into an optimization problem. One possible conversion is to reward for each unique node visited and to penalize for both revisiting a node and for not returning to the starting node (if necessary, a penalty could also be applied for illegal moves through the graph). So assuming that this conversion is acceptable, the requirement is classified as a decision problem with a conversion to an optimization problem. The second TSP requirement is already an optimization problem, i.e., the shorter the path the better. However, the requirement is a minimization problem; so it must be converted to maximization. Taking the inverse of the value is one method to perform this conversion, assume for the running example that this is what is decided upon.

C. Fitness Function Components

After the classifications for each requirement have been made, the next step is to implement the fitness function components that will enforce the requirements in the fitness function. The central idea behind the classification scheme described in the preceding sections is to guide the developer's thought process through the various aspects of fitness function design, in order to force each aspect to be considered ahead of time. This should make the process of actually developing a fitness function much simpler.

From the first level of classifications we know what each fitness function component will require as input. The second level of classification details specifically what will be provided to the fitness function component, whether or not a sample method will be necessary, and whether or not the fitness function component will calculate true fitness or approximate fitness. The third level of classification provides information on how the fitness function component will perform its assessment of a candidate solution based on the inputs provided. With all of this information, piecing together the actual fitness function components should prove to be a relatively simple task.

The last step is to combine all of the fitness function components into the fitness function for the problem. This step is largely dependent on the developer's desired format for the fitness function. One option for this step is to combine the fitness function components into a large single function in which the component fitness values are combined together into a single fitness value. This option may work well for some cases; however, combining the various component fitness values can sometimes be difficult. Weighting the component fitnesses can often be challenging and, even if a good weighting scheme is developed, it is still possible for components to conspire in order to increase

their component fitness values to the detriment of the fitness function overall. A second option is to use Multi-Objective EA (MOEA) [10] methods to calculate a fitness value based on the pareto front generated by using each fitness function component as a new dimension. This allows for the optimization of the component fitness values without the potential trouble of component weighting and conspiracies.

In the TSP example, the first requirement was classified as a phenotypic, tractable, decision problem with optimization conversion. So from this, the component will take the problem space (i.e., the A matrix) as an argument (in addition to a candidate solution) and will calculate the true fitness using the method discussed. Suppose this method is implemented as a function called *CheckHamCirc* which takes an individual *S* and the adjacency matrix *A* as arguments. The second requirement was classified as a (converted) genotypic, tractable, optimization problem. So the fitness function component for the second requirement will take only a candidate solution as an argument and will calculate the true fitness for the requirement by calculating the inverse of the path length. Suppose this component is implemented as a function called *InverseLength* that takes a candidate solution as an argument. The last step in this example is to generate the fitness function F for the problem using the fitness function components. One option is to combine the components into a single expression:

$$F(S) = CheckHamCirc(S, A) + InverseLength(S)$$

If the component fitness values are normalized to fall in the same range (e.g., [0,100]) then there will likely be no problem with using this fitness function. However, another option is to implement this fitness function as a two dimensional MOEA.

III. Examples

A. Finding the Inverse of a Function

For this example suppose our problem is to find the inverse of a given function f non-symbolically. If the function g is the inverse of f then if $f(x)=y$ that means $g(y)=x$. A formal discussion of this problem can be found in [11]. From these specifications the sole requirement for this problem can be identified:

1) For a function g to be a solution, $f(g(x)) = x$

Now an appropriate EA configuration must be decided upon. The evolutionary operators will need to be able to easily modify the elements in candidate solution functions, so a tree representation for the candidates would be useful; this means that GP would be a good decision for this problem. GP implementations often experience tree bloat during the evolutionary process, so it would be a good idea to promote smaller tree size in the fitness function, which means there will be an algorithm induced requirement:

2) Tree size should be minimized

1) Classify Requirement One: This first requirement was identified from the problem specifications, which means that it is phenotypic. The domain for this problem is all real numbers (assuming no domain restrictions based on the particular function used for f). Clearly, it will be impossible to calculate a true fitness for this requirement, so it is intractable. This means that the component will require a sampling method of some sort to select values with which to test the candidate solutions. There are three implementation questions concerned with how this testing is conducted (these questions are typically common to problems requiring sampling methods):

- How many values is each candidate tested against?
- How often are the values reselected?
- Are all candidates tested against the same values?

Suppose for this problem it is arbitrarily decided that there will be 1000 values selected using a random sampling function prior to execution and each candidate will be tested against all selected values. Since there are multiple intermediate fitness values for each candidate, a composition method will need to also be decided upon. Typical composition methods are summation and averaging; for this example suppose averaging is decided upon (i.e., the component fitness will be the average of all 1000 intermediate fitness values). This requirement is stated as a decision problem, so it must be transformed to an optimization problem. One possible transformation is to use the fact that if a function g is an inverse of f then $f(g(x)) = x$ and anything else yielded by this can be used to generate an error value (i.e., $error = |f(g(x)) - x|$).

2) Classify Requirement Two: The second requirement is algorithm induced, which means that it is genotypic. Also, since it is genotypic the requirement is also tractable, so the true fitness will be calculated by the fitness function component. The second requirement is stated as an optimization problem, so no transformation will be necessary. However, the requirement is a minimization problem and fitness functions are typically maximized. One method of changing minimization to maximization is to use the inverse of the output value (i.e., if a tree has *n* nodes then the component fitness will be *1/n*); suppose this is what is decided upon for this problem.

3) Fitness Function Generation: The first requirement was classified as a phenotypic, intractable, decision problem with a transformation to optimization. From the classification process we determined that the candidate solutions will be tested against a static set of values selected before execution, say these values are stored in an array *D*. Also, the component fitness will be calculated by taking the average of all 1000 intermediate fitness values. With this information the fitness function component can now be pieced together (where g is the candidate whose fitness is being calculated): $0.001 \cdot \sum_{x \in D} |f(g(x) - x|$

The second requirement was classified as a genotypic, tractable, optimization problem (with a conversion from minimization to maximization). Through the classification process, the implementation of this fitness function component is straightforward (assume that the *TreeSize* function counts the number of nodes in the argument candidate solution): $\frac{1}{TreeSize(g)}$

The final step is to decide how to combine the fitness components. Just like in the first example, this fitness function would likely work fine as a single equation (i.e., just sum the component fitness values) or a MOEA could be created using the components.

B. Correction of a Sorting Program

This example will show the application of the guide to a real world problem and compare the results of a system using the generated fitness function with that of the state of the art solution to the problem. The real world problem is to implement a system that performs automated correction of bugs in software artifacts. The two systems being compared use the same general idea in their implementations: use a coevolutionary system that evolves candidate solutions (i.e., programs based on the provided software artifact) in one population and test cases for the programs in another population. The program population is evolved using GP and the test case population is evolved using a standard EA. A

fitness function must be designed that guides the evolutionary process toward a correct version of the software artifact.

For this example, both systems will attempt to correct buggy versions of the bubble sort algorithm (that is the problem presented for the state of the art system). As a result, the problem specifications become that of any sorting algorithm: the output of the algorithm should be the input in sorted order.

1) CASC System Fitness Function: In [12] the Coevolutionary Automated Software Correction (CASC) system was introduced. The CASC system operates as is described in the previous section. This will be the system that uses the generated fitness function. The CASC system is not designed to be a multi-objective system, so the components must be combined into a single fitness function. From the specifications described in the previous section, two solution requirements can be identified:

1) The output must be in sorted order
2) The output must be a permutation of the input

Since we are using GP, tree bloat could be an issue; however, preliminary testing of the CASC operators have shown that the system does not suffer from bloating. So bloat does not need to be considered in the fitness function.

The first requirement is a problem requirement, which means that it will be phenotypic. Because it is phenotypic, the next step is to determine if it is tractable or intractable. There is an infinite number of possible inputs to a sorting algorithm, so the problem is clearly intractable. This problem shows an example of a case where a requirement is intractable but the sampling method is built into the problem thus making an additional sampling method unnecessary. This is due to the fact that the candidate solution inputs are also being evolved in a second population. So when calculating the fitness for a candidate solution, the algorithm's selection operator will select the inputs. The component will have multiple intermediate fitness values, so a composition method will need to be selected; say that averaging the intermediate fitness values is decided upon. The requirement is stated as a decision problem (i.e., the output is either sorted or it is not), so must be transformed into an optimization problem. One way of doing this is by first considering what it means for an element to be in sorted order among other elements: the element is greater than or equal to all elements before it and less than or equal to all elements after it. From this it is straightforward to derive a scoring method for the fitness function component (called *CountSorted*): generate a score for each element x that is equal to the number of elements before x that should be before x plus the number of elements after x that should be after x; the scores for all elements is then summed up and returned as the score for the output.

The second requirement is also a problem requirement, so it is phenotypic. There is an infinite number of possible inputs to a sorting program so the requirement is intractable. The component for this requirement should operate on the same output values that the first component does, so there will not need to be any additional input selection. The requirement is stated as a decision problem, so it must also be transformed. A straightforward transformation is to just count the number of elements missing and use that value. The number of missing elements should be minimized, so a conversion to maximization must be performed. One way to convert this value is to use it to apply a penalty to the first component (called *CalcPenalty*), e.g., if all elements are present then there is no penalty, if half of the elements are present then the fitness of the first component is halved, etc.

Using the classifications for the requirements, piecing together the resulting fitness function is straightforward. The requirements operate on the same set of inputs (determined by the algorithm selection method) and the second component acts as a penalty to the first component. The resulting fitness function will be:

```
Score = 0
Inputs = SelectInputArrays()
for i = 1 to SizeOf(Inputs) do
   output = execute(P, Inputs[i])
   newScore = CountSorted(output)
   Score = Score + newScore · CalcPenalty(inputs[i], output))
end for
P.Fitness = Score / SizeOf(Inputs)
```

2) Results in Comparison: In [13], [14] Arcuri presents the state of the art system for automated bug correction. The presented system has the same overall design as CASC; however, it differs in key implementation aspects. Arcuri's system automatically generates a fitness function based on formal specifications provided to the system. For specific details on the fitness function generated for correcting a sorting algorithm see [13]. There are two additional requirements considered in Arcuri's system that are not considered in the CASC system. The full set of requirements considered in Arcuri's fitness function is as follows:

1) The output must be in sorted order
2) The output must be a permutation of the input
3) Program tree size should be minimized
4) Run time exceptions should be minimized

The added third requirement is a straight forward algorithm induced requirement, an example is shown in Section III-A. The fourth requirement is an additional problem requirement that is made possible by the fact that Arcuri's system interpretively executes the candidate solution programs, whereas the CASC system compiles the programs and executes the resulting binary program. Using interpretive execution, Arcuri's system is able to count and respond (i.e., step over) exceptions that arise during execution. Run time exceptions in the CASC system result in program termination and the candidate solution is assigned the minimum fitness value. The guide classifications of both fitness functions are shown in Table I.

Req.	Classification		
CASC Fitness Function			
1	Phenotypic	Intractable	Decision
2	Phenotypic	Intractable	Decision
Arcuri Fitness Function			
1	Phenotypic	Intractable	Decision
2	Phenotypic	Intractable	Decision
3	Genotypic	Tractable	Optimization
4	Phenotypic	Intractable	Optimization

TABLE I
CLASSIFICATION OF BOTH FITNESS FUNCTIONS CONSIDERED

When comparing the systems, both were tested against the same buggy software using similar parameter configurations. The software being corrected consists of eight buggy implementations of bubble sort detailed in [13]. Four system configurations were considered. Results are presented for Arcuri's system using the automatically generated fitness function and the same fitness func-

tion with an added short program penalization (under the rationale that the correct version of the buggy software artifact will be similar in size). Experiments were conducted with the CASC system using the guide generated fitness function and Arcuri's fitness function (except for the run time exception portion).

Bug ID	Arcuri	Arcuri w/ Penalty	CASC w/ Guide Fit.	CASC w/ Arcuri Fit.
1	64%	84%	98%	100%
2	74%	94%	92%	90%
3	83%	97%	96%	43%
4	68%	85%	53%	66%
5	68%	79%	0%	0%
6	0%	0%	6%	6%

TABLE II
EXPERIMENT SUCCESS RATES OVER 100 RUNS

The results presented in Table II provide evidence showing that the guide generated fitness function performs at least as well as the fitness function used in the state of the art system (and even performs better in some cases) for the first three bugs considered. For the fourth bug the guide fitness function is still competitive, though it does not perform as well as Arcuri's fitness function. The significance of these results is that they provide evidence that the guide can be used by a practitioner to generate a competitive fitness function (in terms of quality). The performance of the two systems on the fifth bug is anomalous, as there is no obvious reason for the differences in performance. However, this difference is likely due to the different operators employed by the two systems.

IV. FUTURE WORK

There is still an element of fitness function design experience that is helpful in the design process even while using the guide. Areas where that expertise is useful should be investigated and formalized, removing the need for that expertise. Additionally, methods for determining the quality of the fitness function generated for a problem need to be provided. This would allow the guide to show how to generate high quality fitness functions using methods like fitness landscape characterization [15], fitness function approximation [16], adaptive fitness function design [17], etc. The implementation of this expansion would involve investigating what problem conditions make the advanced algorithms effective and to implement a checklist for these conditions that would indicate when and how the algorithms should be used. Fitness functions are not used solely by EAs. The applicability of the outlined guide could be widened to include all black box search algorithms. Generalized coding templates could potentially be used to generate a fitness function given a few problem specific details and the classifications applied. This would definitely increase the usefulness of the guide for practitioners, though it will likely be difficult to decide on a set of optimal implementations for common fitness function design needs. This would make it possible to encode the whole process into a user guided tool that would generate a fitness function in a fully automated manner, again, dramatically increasing the usefulness of the guide.

V. CONCLUSIONS

The goal of this research is to develop a guide for fitness function design that can be used by non-expert practitioners that would like to use EAs but do not have the expertise to develop an effective fitness function. This paper outlines a practitioner's guide for fitness function design, illustrating how the guide would be used and provides examples of fitness function generation using the guide. Results are presented that show, when properly applied, the guide is capable of generating a fitness function that can be competitive with an expertly designed fitness function. Also presented are a number of future avenues of research that not only improve the effectiveness of the guide but also widen the guide's applicability area that guide could be used for.

REFERENCES

[1] Y. Jin and J. Branke, "Evolutionary Optimization in Uncertain Environments - A Surevy," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 3, pp. 303–317, 2005.

[2] A. Rodrigues, P. de Mattos Neto, and T. Ferreira, "A prime step in the time series forecasting with hybrid methods: The fitness function choice," in *International Joint Conference on Neural Networks*, 2009, pp. 2703–2710.

[3] C. Yalcin, "Evolving Aggregation Behavior for Robot Swarms: Evolving aggregation behavior for robot swarms: A cost analysis for distinct fitness functions," in *International Symposium on Computer and Informational Sciences*, 2008, pp. 1–4.

[4] L. Doitsidis and N. Tsourveloudis, "An Empirical Study for Fitness Function Selection in Fuzzy Logic Controllers for Mobile Robot Navigation," in *Annual Conference on IEEE Industrial Electronics*, Nov. 2006, pp. 3868–3873.

[5] S. Remde, P. Cowling, K. Dahal, and N. Colledge, "Evolution of Fitness Functions to Improve Heuristic Performance," in *Learning and Intelligent Optimization: Second International Conference*, 2008, pp. 206–219.

[6] K. Dahal, S. Remde, P. Cowling, and N. Colledge, "Improving Metaheuristic Performance by Evolving a Variable Fitness Function," in *8th European Conference on Evolutionary Computation in Combinatorial Optimization*, 2008, pp. 170–181.

[7] T. Jansen, "On the Classification of Fitness Functions," University of Dortmund, Tech. Rep., 1999.

[8] A. Nelson, G. Barlow, and L. Doitsidis, "Fitness functions in evolutionary robotics: a survey and analysis," *Robotics and Autonomous Systems*, vol. 57, no. 4, pp. 345–370, 2009.

[9] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 2001.

[10] K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley and Sons, 2001.

[11] E. Weisstein, "Inverse function," From Mathworld - A Wolfram Web Resource. http://mathworld.wolfram.com/InverseFunction.html.

[12] J. Wilkerson, "Coevolutionary Automated Software Correction: A Proof of Concept," Master's thesis, Missouri University of Science and Technology, 2007.

[13] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *CEC 2008 - the 2008 IEEE Congres on Evolutionary Computation*, Jun. 2008, pp. 162–168.

[14] A. Arcuri, "Automatic software generation and improvement through search based techniques," Ph.D. dissertation, University of Birmingham, 2009.

[15] M. Mitchell, S. Forrest, and J. Holland, "The Royal Road for Genetic Algorithms: Fitness Landscapes and GA Performance," in *Proceedings of the First European Conference on Artificial Life*, 1991, pp. 245–254.

[16] Y. Jin, "A comprehensive survey of fitness approximation in evolutionary computation," *Soft Computing*, vol. 9, no. 1, pp. 3–12, 2005.

[17] M. Majig and M. Fukishima, "Adaptive Fitness Function for Evolutionary Algorithm and Its Applications," in *International Conference on Informatics Education and Research for Knowledge-Circulating Society*, 2008, pp. 119–124.