Motivations for Exhaustive Search Algorithms Based on Evolutionary Algorithms

ABSTRACT

We theoretically explore some of the properties of evolutionary algorithms. We discover that under certain conditions, it is more advantageous to utilize a restarting procedure for the evolutionary algorithm than to continue to allow the algorithm to run, due to an exponentially increasing time required for transitions between optima. We discover conditions where a *pseudoexhaustive algorithm* based on a given evolutionary algorithm may be able to outperform the evolutionary algorithm on which it is based. The algorithm is shown to perform as well or better than the evolutionary algorithm it is built from on problems taken from the literature and a digital hardware design problem.

track:evolutionary combinatorial optimization

1. INTRODUCTION

Evolutionary algorithms [5, 6, 1] have been in existence for more than forty years, and thus far have been applied to a variety of problems. An evolutionary algorithm is an optimization algorithm that utilizes evolutionary operators such as crossover, mutation, and reproduction. The evolutionary algorithm starts with an initial population. It proceeds by carrying out cycles of random alterations to individuals, which could involve more than one individual, evaluations of the individuals used to determine how "fit" each individual is, and selection and reproduction, yielding the next set of individuals. Those individuals higher fitnesses typically have a better chance of being selected for the next set. These algorithms can quickly traverse the search space, finding ever better individuals as the traversal continues. Evolutionary algorithms have been found to perform surprisingly well, leading to their application to many problems in the literature.

Prior to evolutionary algorithms, much work was done on exhaustive search algorithms. Exhaustive search algorithms search through all reachable pathways in the search space until they reach the global optimum. There are several different types of exhaustive search: depth-first search, breadth-first search, and best-first search. Depth-first search chooses a path in the search space and travels down that path. In order to fully explore the search space, backtracking is permitted. Yet, the time it takes for the search to find the global optimum is large for relatively broad search spaces. Another type of search is the breadth-first search, which explores the search space one level at a time. For search spaces with a relatively large number of levels, however, the performance time may be very large. The final type of exhaustive search that we will discuss is the best-first search. This search chooses the most fit path first, and travels down that path, choosing the more fit paths first. For deceptive search spaces, this search is not the best candidate for optimally finding the global optimum.

Recently, researchers have been using memetic algorithms which combine exhaustive search algorithms with evolutionary algorithms. The result has been algorithms that are quite good at discovering optima that neither algorithm can find individually in comparable computation times. This indicates that properties of evolutionary algorithms exist that can be useful to exhaustive algorithms, and vice-versa. We believe that the completeness of the search of an exhaustive algorithm is its strength while the adaptivity of the evolutionary algorithm is its strength. Combining these two properties has proven to be a useful technique in optimization.

While the marriage of these two methodologies seems to create an algorithm with capabilities that exceed those of either one, this by no means indicates that it is impossible to have a similar effect in approaching the same thing from another point of view. In this paper, we examine an algorithm that is more exhaustive than evolutionary, but contains the strong points from both parts of the standard memetic algorithm. We demonstrate that the algorithm has performance that is at worst comparable to an evolutionary algorithm and at best much superior to that of an evolutionary algorithm constructed with the same evolutionary operators. We give this particular algorithm the name Directed Pseudoexhaustive Search(DPS).

We motivate our work in this paper by first theoretically examining evolutionary algorithms in general. We explore the performance of an evolutionary algorithm, characterizing it as a combination of two processes - traversing a pathway to an optimum and transitioning from one pathway to another. We demonstrate that under certain conditions, using long runs can be advantageous, while under other conditions, restarting the algorithm is more advantageous. The transition regions between local optima are defined and used to explore the conditions under which either strategy is preferred.

Two corollaries of the current theoretical formalism give rather suprising results. The first is that under certain conditions, it is a mathematical certainty that the genetic algorithm will *always find the optimum of the search space*. The conditions under which this is true are given. The second theoretical result is that a dynamic change of encoding, under certain conditions, is advantageous, and can be exponentially so. While these results are not further explored in this work, they do serve to clarify some of the empirical results that have appeared over the years.

The remainder of the paper is organized as follows. Section 2 describes some theoretical results that are important for evolutionary algorithms. Section 3 introduces DPS. Section 4, discusses the results of simulations in which simplified evolutionary algorithms and DPS algorithms are used on problems borrowed from the literature. Section 5, provides data from our simulations and examines its implications. Section 6 offers some concluding remarks.

2. THEORETICAL CONSIDERATIONS

This section begins with a description of the basic assumptions of the class of evolutionary algorithms examined here. Once the basic assumptions have been clarified, we describe an *extended diversification operate* which is a generalized mutation/crossover operator. This operator is tied in with the reproduction operator, and can be used to understand why the genetic algorithm is not generally limited to specific subspaces.

We continue with an examination of the meaning of different representations in optimization algorithms. This examination leads to the idea that optimization algorithms behaving like evolutionary algorithms tend to move between regions of the search space. The movement is essentially connected to the connectiveness of the space, which itself is a result of the representation. The representation, therefore, is important in the sense that different regions of the search space may be "further" or "closer", according to the number of steps required to reach them, as a result of which representation is used. This view of representation is used to determine the effect of the various random representations.

In this paper, we omit the proofs of the various Propositions and Corollaries because of space restrictions. The proofs of these facts will be given in an upcoming longer version of this paper.

2.1 Extended Diversifications

We assume in the following work that the systems to which we apply this formalism have the following properties: (1) The algorithm maintains a population, which is a set of vectors, that are stored in memory. The vectors are stored in memory and acted on by operators which define the evolutionary algorithm. The state of the evolutionary algorithm is defined by the population, which is a set of individual vectors stored in memory, and the current operator being employed. (2)The diversification operator(s) introduce new elements to the population using a combination of crossover and mutation events. Diversification operators do not include selection. (3) Selection culls the population by replacing some elements with others, preferentially replacing lower scoring individuals with higher scoring individuals. (4) We assume we are working with a finite search space Γ. The details of these behaviors are not important for the arguments made below, though the arguments made from here on out will be true as long as these assumptions are true.

There are, of course, two radically different types of populations. In the first type, the population is finite (which is required for most practical applications involving EAs), and in the second, the population is infinite. In this paper, we will explore the finite population case. The population has a specific number of elements which is maintained by the evolutionary algorithm. Additions of new elements to the population must be accompanied by removals of elements from the population.

The diversification operator generally operates in the following way: based on the population (or in some cases independently of the population), the diversification creates new vectors for consideration. These vectors either immediately become part of the population, replacing vectors in the population, or are subject some culling. The methods for including these are varied. They may include finding a vector whose score is lower than the new vector and replacing it, or randomly choosing a new vector to replace. When this step is combined with selection one typically finds a lower-scoring vector to replace with the new vector. If one cannot be found, the new vector is not added to the population. Generational selection typically consists of a removal of the lower scoring individuals from the population and replacement either with an individual produced by a diversification event or by an individual which is a copy of an existing individual in the population.

Often times, the diversification operator is limited in the sense that a single application of this operator cannot transform any given vector into any other given vector in the search space. As an example diversification operators derived from single point mutation and any finite-point crossover operations cannot change all binary vectors into any other binary vector. In this case, it is the repeated action of the diversification operator which allows the vector to be changed completely from one vector to another vector. In population-based search algorithms, it is possible for diversification steps to act additively, extending the capability of the diversification operator. We call such a pseudo-operator an ***extended diversification operator*** and each sequence of connected diversifications an ***extended diversification event***.

Let us now take the original population of N elements and enumerate them. We represent these as $\{x_1, \dots, x_N\}$. Then, each element discovered by the extended diversification operator can be added to this list of elements. Thus, if M extended diversification events have occurred, then the sequence of elements is given by $\{x_1, \dots, x_{N+M}\}$. We may think of the current population as being the subset of elements from this sequence of elements that is being considered when the next element is developed. Thus, we may write the next element as a function of the current sequence, with an emphasis on the current population. That is, we may write a recursion relation as

$$x_{N+M+1} = f\left(P; x_1, \dots, x_{N+M}\right) \tag{1}$$

where f is the diversification operator. Let us designate the entire set of numbers a sequence X.

What this means, then, is that the entire evolutionary algorithm can be likened to a method for generating an infinite sequence. The sequence of numbers is complex to analyze, but it is still a *deterministic* sequence of numbers.

The main problem is that the sequence is often times a *repeating* sequence of vectors containing multiple copies of many of the elements. Simply because an element has been removed from the population does not require this element to never again reappear in the population. Let us now consider the subsequence of elements of X which do not appear earlier in the sequence. Let us designate this sequence of elements as $Y = \{y_i\}_{i=1}^{Max}$ where Max represents the final unique discovery in the sequence X. For finite spaces, Max is finite; for infinite spaces, Max may be infinite.[1]

We can then define different types of algorithms. An optimization algorithm can be defined as *eventually stagnant* if $Max < |\Gamma|$ where Γ is the number of elements in the entire search space. An example of an algorithm that is eventually stagnant no matter the starting point is a hillclimbing algorithm, no matter the number of starting vectors or the position of the starting vectors.

[1]Note that this sequence is identical to the one used in the arguments for the No Free Lunch theorems.

One important question is whether or not evolutionary algorithms are eventually stagnant. This is dependant on the evolutionary algorithm's diversification operator. The following proposition addresses the future of optimization algorithms whose diversification operators or extended operators have no limitation in their reach. Let us define the probability of an extended diversification operator D changing vector $\overrightarrow{v_1}$ to vector $\overrightarrow{v_2}$ as $p_{D(\overrightarrow{v_1},\overrightarrow{v_2})}$.

PROPOSITION 1 *Suppose that an evolutionary algorithm has an extended diversification operator D such that given any two vectors $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ in the search space Γ, $p_{D(\overrightarrow{v_1},\overrightarrow{v_2})} > 0$. Then the evolutionary algorithm is not eventually stagnant.*

The importance of this proposition comes from its application to the optimization, and forms the motivation for the use of evolutionary algorithms. The following Corollary illustrates its use in optimization.

COROLLARY 1 *An evolutionary algorithm which has an extended diversification operator d such that given any two vectors $\overrightarrow{v_1}$ and $\overrightarrow{v_2}$ in the search space Γ, $p_{D(\overrightarrow{v_1},\overrightarrow{v_2})} > 0$ will always find the space's optimum.*

Corollary 1 provides a clear description of the motivation for using evolutionary algorithms. It is clear that, using evolutionary algorithms, the optimum will eventually be visited by the algorithm. In order to do this, it is merely necessary to either construct a diversification operator that, in one step, has a p which is nonzero for all possible mutations, or an extended diversification operator which does the same.

2.2 Evolutionary Pathways

Once a problem encoding and a fitness function have been chosen, the fitness of each of the points in the space is defined. Once the encoding of the diversification operator has been defined, the connectivity of the search space is defined. Therefore, each point has a well-defined number of positive and negative transitions. That is, a single diversification event applied to any single point will generate a new vector whose fitness value is likely to differ from the original one. A reduction in fitness value is viewed as a negative transition, while an increase in fitness is a positive transition. We define a point as a ***local maximum (minimum)*** if the point has no positive (negative) transitions. Note that these definitions mirror similar ones given earlier.

The set of all positive transitions defines a structure in the search space. This structure defines the way a hillclimbing algorithm might progress. Evolutionary algorithms tend to follow the same paths that hill- climbing algorithms follow if the improvements are relatively direct and quick. If, however, the improvements require more time, the evolutionary algorithm will tend to spread out in many directions, choosing another optimization directon from those built into this structure. Therefore, this structure defines a scaffolding upon which the evolutionary optimization algorithm can work. More rigorously, we define a ***path*** to be a finite ordered set of points in which any given point could give rise to the next point using a single diversification yielding a positive transition. We define a ***path bundle between point a and b*** to be the set of all paths through state space which follow positive transitions only, begin at **a**, and end at **b**. Let **b** be fixed. We define the set of all trajectories ending at **b** to be the ***heap of trajectories leading to b***. Let us represent this as $H(b)$. Finally, suppose that **b** is a local maximum (minimum). Then, we call the heap of trajectories leading to **b** the ***maximal (minimal) heap of trajectories leading to b***. We denote this as $H_m(b)$. Note that $\partial H_m(b)$, ***the boundary of the maximal heap of trajectories leading to b***, is a set such that at each point at least one transition is part of a path that leads to **b** and at least one is a transition that is not part of such a path.

Note that the boundary regions can be located within a basin of attraction. Since individual basins of attraction can contain smaller basins of attraction, they can also contain boundary regions. The regions, we will see, are locations in which "decisions" are made by the algorithm about which path bundle to climb.

PROPOSITION 2 *Suppose that $H_m(b) \cap H_m(b') \neq \emptyset$. Then at each point in the set $H_m(b) \cap \partial H_m(b') \cup \partial H_m(b) \cap H_m(b')$ a transition is possible which excludes (except with a backward transition) the local maximum b or b'.*

Because of this, we may designate the set $H_m(b) \cap \partial H_m(b') \cup \partial H_m(b) \cap H_m(b')$ ***the decision region for the sets $H_m(b)$ and $H_m(b')$***. Note that if there are only two local maxima, this region will always choose between the two different maxima.

The decision region is an important part of the space, as it makes an ascending evolutionary algorithm choose between the quick transitions that lead it to a single maxima. As the population progresses through the search space, it passes through the transition regions, effectively "choosing" between future maxima. Once the population has made a decision, it becomes increasingly unlikely to return to the decision region and to make a new decision.

The search space can be characterized by the number of disparate starting points and decision regions a population will pass through as it moves toward a local optimum. As each decision region forces the population to choose between at least two local maxima, the number of potential end points must be at least $m2^k$ where m represents the number of disparate starting populations possible in the space [2] and k represents the number of transition regions that the population is likely to encounter during the optimization. This forms a lower limit on the number of times an unbiased evolutionary algorithm would have to be restarted in order to guarantee that the population visited the optimum. The value of m is very problem-dependent, and may be infinite. Moreover, k is highly dependent on the path bundle taken. These two factors make this type of analysis rather unreliable in determining a solid lower bound on the number of attempts one must make.

3. MERGING APPROACHES

In the preceding analysis, we've examined properties of the system that lead to two competing approaches to the optimization. In the first approach, a single long run is attempted. Given enough time, we have seen that the extended diversification operator will produce vectors that will move the population from one basin of attraction to another, meaning that this approach will eventually succeed.

The second approach is generally to start and re-start the algorithm so as to traverse as many path bundles as possible. This approach will generally allow the algorithm to make it to basins of attraction that cannot be easily attained by simply using a single run. As we have seen, when the basins of attraction are far apart, then the single run has an extremely hard time finding its way between basins of attraction. When the number of path bundles is small compared to the number of evaluations required to transfer from one basin of attraction to the next, then it is expected to be quicker to start the algorithm over in order to improve the performance rather than to wait for the population to make the desired transition.

A third option that has rarely been utilized is to use a completely exhaustive algorithm. The reason this has been overlooked is that it is generally assumed that the search space is so vast that the use of an exhaustive algorithm would be prohibitive. However, as with an evolutionary algorithm, a well-made exhaustive algorithm might be exhaustive in design, but still exploit the pertinent parts of the search space in the same way the an evolutionary algorithm or other algorithm might.

What we have seen is that the search space is made up of path bundles leading to local maxima which are separated by differing step distances. The evolutionary algorithm will generally explore the search space along these path bundles, and traverse the intermediate locations with a much lower probability. Therefore, our algorithm must be designed to do the same thing. Our design must have the property that it can search ex-

[2]Normally the initial population is distributed across the search space, making m 1.

haustively through the space, but do so in such a way that it can apply the same advantages one might find in an evolutionary algorithm.

Our algorithm is called *Directed Pseudoexhaustive Search (DPS)* [3]. This algorithm performs a *pseudoexhaustive search*, or a search which does not reach the entire space, but systematically exhaustively searches a subspace, which might include the entire space. This algorithm is meant to utilize the strengths of searches such as evolutionary algorithms, while avoiding the sometimes debilitating repetition of evolutionary algorithms.

DPS assumes that the space is discrete, as is the set of possible variations. Moreover, since the set of diversifications is discrete and therefore enumerable, it is assumed that the diversification operator used can be invoked using a vector and a chosen diversification ranging from 1 to the maximum mutation. This makes it possible to keep track of which mutations have been attempted with a single vector and which have yet to be attempted.

Our algorithm utilizes two main data structures. The first data structure is a linked list. The linked list contains the vectors that are currently being examined. These vectors are ordered according to a decreasing fitness value. The linked list is augmented with a list pointer which determines which element in the list will be utilized at a specific moment. The list pointer's control strongly influences the search. We return to it presently.

Each node in the linked list is a record that contains three pieces of information. The first is the vector. The second is an integer known as a variation pointer indicating which variation was last attempted on this vector. The integer is incremented with every mutation operation on a given vector. The third is the fitness value for the vector.

The second data structure used is a lexicographically-ordered binary tree. The binary tree stores the vectors that have been visited by DPS. This is utilized so as to avoid revisiting the vectors. The binary tree is source of greatest memory consumption, and so care must be taken to avoid storing too many vectors. This is done by periodically purging the tree. We describe how this is done without losing the efficacy of the algorithm shortly.

DPS makes use of a reset threshold, which is a floating point number initially set to 1, and which controls the likelihood that the algorithm will "spontaneously" reset the linked list pointer to the head of the list.

DPS starts by creating a predetermined set of binary vectors. This initial population is then inserted into the linked list. Once the initial population is inserted in the linked list, the list pointer is reset to the top of the listand DPS enters continual cycles.

During each cycle, the vector to which the list pointer is pointing is varied, calling on the variation indicated by its variation pointer. The old vector's variation pointer is incremented once the variation occurs. If the old vector's variation pointer exceeds the maximum variation number, it is removed from the linked list. If the new vector does not exist in the binary tree, it is stored in the binary tree. If it also has a fitness that exceeds that of the old vector, it is stored in the linked list with a newly initialized variation pointer set to 1. The linked list pointer is reset to the top of the list and the next cycle begins.

If the new vector does exist in the binary tree or if its fitness is lower than the old vector's fitness, then the list pointer is incremented, and the new vector discarded. If the list pointer reaches the end of the list, it is reset to the beginning of the list.

The binary tree will eventually expand to fill all the memory on a computer, even for moderately sized problems, without maintenance. As a result, we've developed a procedure for moderating its size. Each time an element is found in the binary tree (it has been produced by a mutation earlier), a counter in its node is incremented. Periodically, the binary tree is updated by creating a new tree from the old one, while deleting the old one. The new tree only contains those elements from the old one that have been visited recently. This not only removes many of the nodes, but also serves to include only those that need to be included. Many of the nodes can only be created by going through other nodes. If these other nodes kept in the binary tree, the nodes created using them can be safely removed from the tree without fear that they will be added again to the linked list.

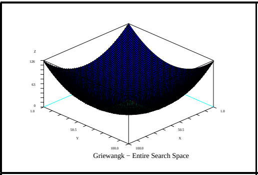
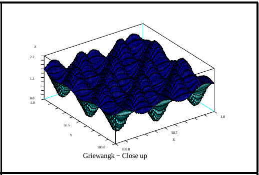
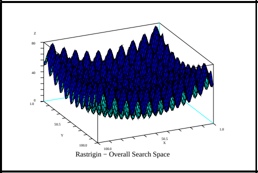
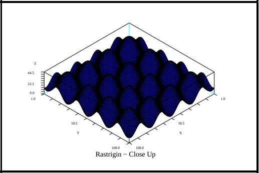
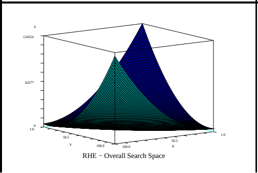
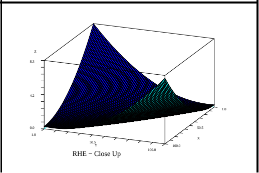
4. TEST PROBLEMS

To examine the effectiveness of the DPS, we have selected three standard problems from the literature: the Griewangk function, the Rastrigin function, and the Rotated Hyper Ellipsoid function. These three functions are noteworthy for the shape of their search spaces [5]. Observed from a distance, each search space appears to be have very little structure, but closer inspection reveals a large amount of structure in the form of small bumps covering the surface on two of the functions. These are significant obstacles that an evolutionary algorithm might have difficulty overcoming. In this situation, EAs are able to quickly progress to the approximate region of the optimum, then encounter a large amount of local optima that quickly halt progess. Like the evolutionary algorithms it is based on, the DPS is able to quickly reach the same region, and has similar difficulties in making progress to the local optima.

As these functions are real-valued functions, and DPS can function only on discrete functions, it is necessary for us to use a discretization of the search space. Therefore, all of the vectors considered consist of $4 * N$ components, where N represents the overall dimension, and

[3]This is similar to the algorithm detailed in [3, 4].

Table 1: These figures illustrate the three test functions taken from the literature. The first two functions have a great deal of structure which must be overcome, while the third, the RHE function does not.

each element is made up of 4 digits running from $0-9$. This means that a vector, say, of two elements would have 8 digits, as

$$(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8) \to (a_1.a_2a_3a_4, a_5.a_6a_7a_8)\,. \quad (2)$$

In order to make this form more symmetric, we use the form $(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8) \to$

$$\frac{Max}{4.999} \times (a_1.a_2a_3a_4 - 5.0, a_5.a_6a_7a_8 - 5.0)\,. \quad (3)$$

We use N values of 5, 10, and 20 for each function. We chose to run each program for ten million evaluations[4].

These simulations use a genetic algorithm and the DPS described previously in the paper. Both use the same mutation operator. The GA has a populations of 100 initially randomly assigned individuals. The mutation probability is 0.05. The algorithm utilizes an elitest mechanism so as to stabilize the population against variations that tend to reverse progress. We use a proportional generational reproduction operator which chooses copies of the current generation to populate the next generation. The mutation operator is designed to mutate single or double digits at a time. We use the DPS described previously in the paper.

When we run the first set of test problems from the literature, we obtain the following data. Note that we report the average performance from thirty independent runs of each model.

Typical runs of both algorithms produce data sets with the following forms.

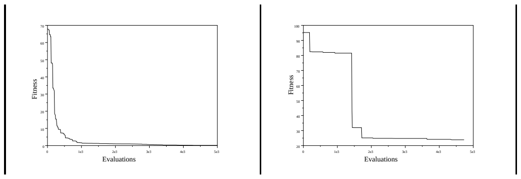


Figure 1: These figures illustrate the best individuals during a GA (left) and a DPS (right)run.

We also use a second problem borrowed from [2]. This problem involves the construction of connectionist networks. Each network is a signal processor, taking in data from the environment, processing it, and producing data at the output[7]. We use a genetic algorithm and a DPS to construct networks that are capable of carrying out the binary addition task. The networks consist of nodes and connections with each node processing the sum of the inputs from other nodes and sending the processed information along to other nodes. Each node is synchronized with the other nodes in the sense that they all update at the same time. The evolutionary process consists of adding, deleting, and changing nodes and connections to the network while facilitating the resultant competition between the various networks. DPS, of course, cycles through the various designs resultant from the mutations described above, eventually generating the desired structure.

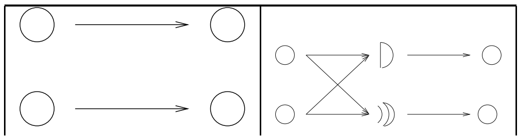


Figure 2: These figures illustrate the three initial and final states of a design problem for a 1-bit adder.

When the binary adder design problems are run, we obtain the following data. Note also that the averaged values are over thirty runs, and those that do not report a value have as yet been unable to produce an optimal design. DPS does not require multiple runs as the algorithm in this case is initialized identically and therefore the outcome is precisely identical each time it is run

5. DISCUSSION

Optimization algorithms such as evolutionary algorithms behave differently according to the part of the search space that they are currently exploring. However, the "magical" part of the evolutionary algorithm is often thought to be tied up in the randomness of the mutation and crossover operators. It is this randomness, it is often times argued, that makes evolutionary algorithms capable of doing things that are outside of

[4]Note that these are evaluations and not iterations. For a moderately sized population of 100 a mutation probability of 0.1 and crossover probability of 0.9, with an estimated 90 changes per iteration, this amounts to the same as 110,000 iterations.

Table 2: This table gives the performance of the various algorithms on the test problems and the design problem.

Function	Dim	Avg. GA Eval	Avg. DPS Eval	Avg. GA Fit	Avg. DPS Fit
Rastrigin	5	2739166±2983373	1051485±1705417	0.007325±0.003759	0.007087±0.004386
	10	2648504±2968023	1006328±1874768	0.021389±0.016817	0.015788±0.013251
	20	1722886±1971145	311682±286100	0.054299±0.043063	0.075424±0.041994
Griewangk	5	2504±612	11071±26077	0.000500±0.000215	0.000448±0.000236
	10	5892±1180	14401±2447	0.001012±0.000325	0.001036±0.000330
	20	12998±1965	58908±9379	0.001996±0.000472	0.002154±0.000456
RHE	5	131954±361036	117360±79634	0.000704±0.000684	0.001178±0.001316
	10	1624788±866692	3346373±1656765	0.001205±0.000826	0.002216±0.005317
	20	8703510±1503281	9954956±41063	65.115463±82.555588	1082±626

Table 3: This table reports the performance of DES and the GA on the binary adder design problem. Problems for which >100,000,000 are reported have run for greater than this amount of evaluations and have yet to produce an optimal design. Note that these data report the first instance of an optimal individual.

Adder	EA Eval.	DPS Eval.
1-bit	113,437.84 ±93,088.298	18,441
2-bit	>100,000,000	8,403,412
3-bit	>100,000,000	31,729,355

more standard optimization algorithms, worthy of an entirely discrete research center.

Recent theoretical work on the No Free Lunch (NFL) Theorem and its consequences [8, 9, 10] has begun to overturn this conceptual thinking. While many people in the evolutionary algorithm community still incorrectly apply this theorem, its correct interpretation is a very important design consideration for evolutionary algorithms. Typical interpretations of the NFL tend to assign equality among all problems to every optimization algorithm. However, this is neither implied nor explicitly stated by the NFL theorem as it does not include any information about the repetitive nature of the evolutionary algorithm. In fact, as all evolutionary algorithms exhibiting *eventual stagnation* are not addressed by the NFL Theorem, and many evolutionary algorithms are, in practice, eventually stagnant, this analysis does not apply to any of these.

What the NFL theorem does correctly do is to indicate that evolutionary algorithms capable of being implemented on a computational device and not eventually stagnant are, in fact, not random at all. Their apparent randomness is a result of a complex set of equations that exhibit many of the properties of randomness, but are not really random. Nonetheless, these algorithms have an impressive ability to yield the global optimum of a search space. As a result, it must be concluded that these algorithms are capable of yielding good results because of how they process the search space. Their randomness would seem to be irrelevent.

The search space processing capability of the algorithm can be clarified by examining the search space structure imposed by the diversification operators (mutation and crossover). This structure defines the distance in diversification steps between different vectors, the probability of each of these steps, and indirectly defines the expected amount of time that will pass between the various transitions. Therefore, an algorithm that processes the search space in the same way as evolutionary algorithms should be able to perform similarly, even if it is not evolutionary in nature.

In our simulations, we have taken care to generate algorithms that are fashioned from the same genetic operators used in our evolutionary algorithms. Because of the nature of crossover and the difficulty in implementing it in an exhaustive way, we have bypassed the use of crossover in our evolutionary algorithms. However, despite the limited capability of our evolutionary algorithms, both they and the versions of DPS formed from each EA instantiation performed quite well on the problems they were applied to.

As expected, limitation of the repetition inherant in an EA and simultaneous path exploration produced improvements in much of the performance of the EAs. The DPS applied to the Griewangk function appears to outperform the evolutionary algorithm. For the Rastrigan function, the DPS once again appears to outperform the evolutionary algorithm. The Rotated Hyper Ellipsoid function's results, however, show that the evolutionary algorithm's performance is superior to the DPS's performance. However, an examination of the search space, given in Table 1, demonstrates that the search space is incredibly smooth, non-deceptive, and has a single minimum. In such a space, the machinery of DPS would seem to be a hinderance. However, a hillclimbing algorithm would also be expected to do just as well as or better than the GA in this case.

We have also applied EAs to the design problem taken from [2]. This particular problem is significantly different from the previous ones in that it is a design problem which builds structures incrementally. The construction task is very heavily affected by the path that is taken; the wrong path *cannot* lead to correct construction without complete deconstruction of the current design. An EA will *never* succeed in this kind of design if the deconstruction required has too great a required backstep in fitness.

Indeed, we have found in these runs that the EA was capable of finding the correct design in these searches for the smaller design problems. However, it utterly failed in hundreds of attempts to design a three bit adder, despite the fact that the DPS algorithm based on this EA was able to design it. This reinforces our assertion that the path and distance between search space components is perhaps the most important of considerations. Algorithms not able to reconnect search space components that are too far away to be realistically reached cannot succeed.

Knowing this is particularly important for a problem like this one because it would seem to by archetypal for the kind of use of evolutionary algorithms that is most exciting. The eventual design of complex structures and machines using search algorithms to sample the design space is compelling. However, in such a problem domain, these considerations are paramount. It is important that they become part of the discussion.

6. SUMMARY AND CONCLUDING REMARKS

In this paper, we've explored some of the theoretical issues concerning evolutionary algorithms. The theoretical issues are interesting in the sense that they validate some of the expected or oft observed but heretofore unexplained behaviors of EAs. The ability of the genetic algorithm, or for that matter any EA with a diversification operator that isn't eventually stagnant, to find the global optimum has never before (to our knowledge) been demonstrated theoretically. This serves to solidify our understanding of the scope and design of the EA, and is related to the design of both the variational operators and the selection operators.

Of prime importance in the design of any search algorithm is the representation of the various vectors and operators. Our theoretical results have demonstrated that, quite suprisingly, random representations of these operators can be very effective in generating improvements in performance. This lends credence to several existing studies that have demonstrated this fact, but have been unable to explain it theoretically. Moreover, this is similar to the use of uniform crossover, which has various advantages over single point and dual point crossover.

Moreover, our examination of the search space and characterization of it as a set of paths has allowed us to understand how one might utilize an exhaustive or pseudoexhaustive algorithm which seems to behave similarly to the EA. Our analysis has indicated that while the EA is an interesting paradigm that seems to perform well with a modicum of machinery and using random operators, the part of the EA which makes it useful and important is its ability to process information gleaned during the search and use it to direct its later actions. When this information processing capability was utilized by our pseudoexhaustive algorithm, the algorithm was able to replicate the performance of the earlier optimized memetic algorithms, though it wasn't itself optimized.

This bodes well for exhaustive and pseudoexhaustive algorithms. The subset of the space that they can explore can be significant enough to rival the performance of EAs, using the strengths of EAs.

7. REFERENCES

[1] E. Cantu-Paz and D. Goldberg. *Are Multiple Runs of Genetic Algorithms Better than One?* E. Cantu Paz et al. (eds.). **Proceedings of Gecco 2003 Conference**. Chicago, IL., 2003.

[2] S. Kazadi, Y. Qi, I. Park, N. Huang, P. Hwu, B. Kwan, W. Lue, and H. Li. *Insufficiency of Piecewise Evolution.* **Proceedings of the Third NASA/DoD Workshop on Evolvable Hardware**, Long Beach, CA, 2001.

[3] S. Kazadi, D. Johnson, J. Melendez, and B. Goo. *Exhaustive Directed Search.* **Proceedings of the Genetic and Evolutionary Computation Conference, 2004**, Seattle, WA, USA, 2004.

[4] S. Kazadi, M. Lee, and L. Lee *A Case for Exhaustive Optimization.* **Proceedings of Gecco 2005 Conference, Late Breaking Papers**, Washington D.C., USA, June 2005.

[5] B. Manderick and M. Weger. *The Genetic Algorithm and the Structure of the Fitness Landscape.* R. Belew and L. Booker, (eds), **Proceedings of the Fourth International Conference on Genetic Algorithms**. San Mateo, CA: Morgan Kaufmann, 1991.

[6] J. Schaffer and L. Eshelman. *Spurious Correlations and Premature Convergence in Genetic Algorithms.* G. Rawlins (ed). **Foundations of Genetic Algorithms**, vol. 1. San Mateo, CA: Morgan Kaufmann, 1991.

[7] A. Thompson and C. Wasshuber. *Design of Single Electron Systems through Artificial Evolution.* **Int. J. Circuit Theory and Applications**, 109-116, 2000.

[8] D. Wolpert and W. Macready. *No Free Lunch Theorms for Search.* **Technical Report, SFI-TR-05-010**, available at www.santafe.edu, 1995.

[9] D. Wolpert and W. Macready. *No Free Lunch Theorms for Optimization.* **IEEE Trans. Evolutionary Computation** 1(1): 67-82, 1997.

[10] D. Wolpert and W. Macready. *Coevolutionary Free Lunches.* **IEEE Trans. Evolutionary Computation** 9(6): 721-735, 2005.