No-Free-Lunch Theorems and the Diversity of Algorithms

Mario Köppen
Dept. of Security and Inspection Technologies
Fraunhofer IPK
Pascalstr. 8-9, 10587 Berlin, Germany
Email: mario.koeppen@ipk.fhg.de

***Abstract*— In this paper, the No-Free-Lunch theorem is extended to subsets of functions. It is shown that for algorithm a performing better on a set of functions than algorithm b, there has to be another subset of functions on which b performs better in average than a. To achieve a performance evaluation for an algorithm, it is not sufficient to demonstrate its better performance on a given set of functions. Instead of this, the diversity of an algorithm will be considered in this paper in more detail. The total number of possible algorithms will be computed and compared with the number of algorithm instances that a random search or a population-based algorithm can have. It comes out that the number of different random searches is very small in comparison to the total number of algorithms. On the other hand, population-based algorithms are principally able to cover the set of all possible algorithms. The smaller variance of algorithm performance, measured by the repeated application of the algorithm under different settings on different random sets of functions, comes out to be a value reflecting the higher count of instances.**

I. Introduction

The No-Free-Lunch theorem [9][10] states that without any structural assumption on a search or optimization problem, no algorithm can perform better than blind search. Several discussions on this theorem were provided in the last decade, including discussion of information theoretic aspects (e.g. [2]), alternate proofs and notations [6][4], and extension to other problem classes (as multi-objective optimization [1][5]). The general viewpoint taken by the proof of this theorem is to average the performance over *all* possible quality functions. The quality functions are assumed to be uniformly distributed. Once the averaging is performed over a subset of such functions, this theorem will only hold if and only if this subset is closed under permutation (this is the so-called Generalized No-Free-Lunch theorem [7]). Recent discussion claimed that since the majority of function subsets is not closed under permutation, the No-Free-Lunch theorem does not seem to have much practical importance [3]. The averaging over all functions involves a vast majority of functions that will never appear in a practical optimization scenario (an approach that was used e.g. to point out the usability of gray encoding in [8]). However, in this paper we will continue in this matter and show that the Generalized No-Free-Lunch theorem gives raise to a further theorem, stating the equal average of average performances over all subsets of functions that are not closed under permutation. In this context, demonstrating the better performance of an algorithm a compared to algorithm b for a set of functions is doing nothing more than showing the existence of another set of functions, for which b will perform better than a. Nothing more is known about this other function set than its existence, as long as the function set is assumed to be uniformly distributed, and alternate viewpoints are not taken into account (as the possible elimination of high complexity functions). However, there is another aspect on algorithm performance that has not been considered so far in more detail: the diversity of an algorithm. So, by speaking about an algorithm like ”genetic algorithm”, we usually do not refer to a single algorithm but to a distribution of algorithms, according to different configurations, runtime parameter settings and pseudo-random number sequences. The question, which will be studied in this paper and considered an alternate viewpoint on algorithm performance is: how many different algorithms can be provided by such an algorithm class, and how does this number behave with respect to the total number of possible algorithms? The answer gives us a ranking for algorithms according to their smaller or larger number of instances. It comes out that by such a ranking, random search is worst, while evolutionary approaches are (at least theoretically) able to provide *any* search sequence that is possible. To see this, we are going to use the framework of the No-Free-Lunch theorem itself to compute the total number of algorithms and the number of instances that a certain algorithm can have.

Section II of the paper states and proves the No-Free-Lunch theorem for sets of functions. Then, in section III, we derive the total number of algorithm instances and prove the correctness of this derivation. The alternate viewpoint on the number of algorithm instances is introduced and discussed in section IV, followed by a discussion of its relation to algorithm performance variance in section V. Section V also provides a procedure to estimate the number of instances by using the variance of algorithm performance.

II. No-Free-Lunch Theorems for Sets of Functions

The following notation will be used. Be X and Y two finite sets, and f_i a mapping from X to Y. There are $N = |Y|^{|X|}$ such mappings, thus $i = 1, \ldots, N$.

Now we consider a deterministic, non-repeating algorithm a. Applying algorithm a for m steps to a function f, it samples an ordered set $(y_1, y_2, \ldots, y_m)$ of function values of f, and

a performance measure p assigns a performance value to this set of function values. We will indicate this performance value as $p_{a,m}(f_i)$ in the following.

With Π we indicate a permutation of the numbers $(1, 2, \ldots, |X|)$ and we write

$$\Pi(1, 2, \ldots, |X|) = (\Pi(1), \Pi(2), \ldots, \Pi(|X|))$$

Now we define the permutation of a function f as follows: when f maps x_k to y_k, then $\Pi(f)$ maps $x_{\Pi(k)}$ to y_k.

Further we specify some sets. The symbol s_a denotes the set of *all* functions f mapping X to Y. With s_n we denote a subset of s_a with exactly n elements $s_n = (f_1, f_2, \ldots, f_n)$.

Definition 1. The set s_n is *closed under permutation* (c.u.p.) iff

$$\forall_\Pi (f \in s_n \rightarrow \Pi(f) \in s_n)$$

The performance measure p is extended to sets of functions:

$$p_{a,m}(s_n) = \frac{1}{n} \sum_{i=1}^{n} p_{a,m}(f_i)$$

which stands for the average performance of algorithm a after m steps on the set s_n of n functions.

The symbol T_{nl} denotes a set of l sets s_n, i.e. subsets of s_a of n elements. A set T_{nl} is said to be c.u.p. if and only if each element of T_{nl} is c.u.p. and the *total performance* of a on T_{nl} is given by

$$p_{a,m}(T_{nl}) = \sum_{s_n \in T_{nl}} p_{a,m}(s_n)$$

(note that the total performance is *not* divided by l).

With T_{na} we denote the set of *all* subsets of s_a of size n, with T_{nc} the set of all subsets of s_a of size n that are c.u.p. and with T_{nn} the set of all subsets of s_a of size n that are not c.u.p. (thus having $T_{na} = T_{nc} + T_{nn}$).

Now we can formulate the Generalized No-Free-Lunch Theorem [7] as:

Theorem 1. *We have $p_{a,m}(s_n) = p_{b,m}(s_n)$ for all algorithms a and b and each number of steps $m \leq |X|$ if and only if s_n is c.u.p.*

This directly gives that from T_{nl} being c.u.p. it follows that $p_{a,m}(T_{nl}) = p_{b,m}(T_{nl})$ and as a special case:

Lemma 1. *The total performance of any two algorithms a and b and any number of steps $m \leq |X|$ on the set of all subsets of s_a that are c.u.p. is equal.*

$$p_{a,m}(T_{nc}) = p_{b,m}(T_{nc})$$

Now we consider the total performance on the set of *all* subsets of size n of s_a and show that this also does not depend on the algorithm:

$$\begin{aligned} p_{a,m}(T_{na}) &= \sum_{s_n} p_{a,m}(s_n) \\ &= \sum_{s_n} \frac{1}{n} \sum_{f \in s_n} p_{a,m}(f) \end{aligned}$$

In this sum, each term $p_{a,m}(f)$ appears $\binom{N-1}{n-1}$ times, since there are exactly $\binom{N-1}{n-1}$ subsets of s_a of size n that contain f (each of these subsets is the set $\{f\}$ unified with a subset of $(n-1)$ elements of $s_a \setminus \{f\}$, which has $(N-1)$ elements). So, we may continue

$$\begin{aligned} p_{a,m}(T_{na}) &= \frac{1}{n} \binom{N-1}{n-1} \sum_f p_{a,m}(f) \\ &= \frac{N}{n} \binom{N-1}{n-1} p_{a,m}(s_a) \end{aligned}$$

The set s_a of all functions is c.u.p., thus

$$p_{a,m}(s_a) = p_{b,m}(s_a)$$

which gives $p_{a,m}(T_{na}) = p_{b,m}(T_{na})$ in consequence:

Lemma 2. *The total performance of any two algorithms a and b and any number of steps $m \leq |X|$ on the set of all subsets of s_a is equal.*

$$p_{a,m}(T_{na}) = p_{b,m}(T_{na})$$

Finally we consider the set of all subsets of s_a that are not c.u.p. Using the two lemmas, we get for any two algorithms a and b and any number of steps $m \leq |X|$:

$$\begin{aligned} p_{a,m}(T_{nn}) &= p_{a,m}(T_{na}) - p_{a,m}(T_{nc}) \\ &= p_{b,m}(T_{na}) - p_{b,m}(T_{nc}) \\ &= p_{b,m}(T_{nn}) \end{aligned}$$

So, for any n the total performance on all sets of n functions that are not c.u.p. does not depend on the algorithm. Finally, if K is the number of all subsets of s_a that are not c.u.p. we get the No Free Lunch Theorems for subsets of functions:

Theorem 2. *For any two algorithms a and b and any number of steps $m \leq |X|$ it holds*

$$\frac{1}{K} \sum_{i=1}^{N} p_{a,m}(T_{in}) = \frac{1}{K} \sum_{i=1}^{N} p_{b,m}(T_{in}) \tag{1}$$

The average of the average performances of an algorithm a over *all* subsets of functions that are not closed under permutation does not depend on the algorithm a. The alternative formulation is that if an algorithm a shows better average performance on a set of functions than an algorithm b (this can only happen if this set of functions is not closed under permutation) than nothing more was shown as the existence of another set of functions (not necessarily of same size, but not closed under permutation) for which b will show better average performance than algorithm a. In no way, the "superiority" of a has been demonstrated.

To make the concept behind the proof more clear, we may consider an example. We choose for X the set $X = (x_1, x_2)$ and for Y the set $Y = (0, 1)$. With f_{ij} $(i, j \in Y)$ we denote the function that maps x_1 to i and x_2 to j (e.g. f_{01} maps x_1 to 0 and x_2 to 1).

There are $2^2 = 4$ possible functions f: $s_a = (f_{00}, f_{01}, f_{10}, f_{11})$. This set has 16 subsets. Out of these subsets, one subset has 0 elements, 4 subsets have one element, 6 subsets have two elements, 4 subsets have 3 elements and one subset has four elements.

For $|X| = 2$ we have two permutations of $(1,2)$: $\Pi_1 = (1,2)$ and $\Pi_2 = (2,1)$. For example, $\Pi_2(f_{01}) = f_{10}$ and $\Pi_1(f_{00}) = \Pi_2(f_{00}) = f_{00}$.

There are 8 subsets closed under permutation: $\emptyset$, (f_{00}), (f_{11}), (f_{00}, f_{11}), (f_{01}, f_{10}), (f_{00}, f_{01}, f_{10}), (f_{01}, f_{10}, f_{11}) and $(f_{00}, f_{01}, f_{10}, f_{11})$. The remaining 8 subsets are not closed under permutation: (f_{01}), (f_{10}), (f_{00}, f_{01}), (f_{00}, f_{10}), (f_{01}, f_{11}), (f_{10}, f_{11}), (f_{00}, f_{01}, f_{11}) and (f_{00}, f_{10}, f_{11}). For example, for the set (f_{10}, f_{11}) the permutation Π_2 transforms f_{10} into f_{01} but f_{01} is not an element of the set.

For simplicity we write $p(f)$ instead of $p_{a,m}(f)$ in the following. So, the average of the average performances of algorithm a after m steps over all subsets of functions that are not closed under permutation is given by the expression:

$$\begin{aligned} P &= \frac{1}{8} \times \Big[p(f_{01}) + p(f_{10}) + \\ &+ \frac{1}{2}\big(p(f_{00}) + p(f_{01})\big) + \frac{1}{2}\big(p(f_{00}) + p(f_{10})\big) + \\ &+ \frac{1}{2}\big(p(f_{01}) + p(f_{11})\big) + \frac{1}{2}\big(p(f_{10}) + p(f_{11})\big) + \\ &+ \frac{1}{3}\big(p(f_{00}) + p(f_{01}) + p(f_{11})\big) + \frac{1}{3}\big(p(f_{00}) + \\ &+ p(f_{10}) + p(f_{11})\big) \Big] \end{aligned}$$

Now the generalized No Free Lunch theorems gives that the average performances on subsets that are closed under permutations are constant values with respect to the algorithm (but they depend on m, of course). This gives the equations:

$$\begin{aligned} p(f_{00}) &= c_1 \\ p(f_{01}) + p(f_{10}) &= c_2 \\ p(f_{11}) &= c_3 \end{aligned}$$

with c_1, c_2 and c_3 the corresponding constants. We use this to resort the terms and evaluate the expression for P:

$$8P = c_2 + c_1 + \frac{1}{2}\,c_2 + c_3 + \frac{1}{2}\,c_2 + \frac{2}{3}\,c_1 + \frac{2}{3}\,c_3 + \frac{1}{3}\,c_2$$

Thus it can be easily seen that the expression for P will also not depend on the algorithm.

III. Number of algorithms

The foregoing section demonstrated that the testing of algorithms by applying them onto a set of given functions (usually called a benchmark) does not prove so much about the superiority of an algorithm. We are going to introduce a different viewpoint on algorithm performance here, related to the diversity of an algorithm.

The framework of the NFL theorems allows for the computation of the number of different algorithms for given set sizes $|X|$ and $|Y|$. Assume that we have indexed all possible mappings $f : X \to Y$ by $f_1, f_2, \ldots, f_N$ with $N = |Y|^{|X|}$. The set $X = \{x_1, x_2, \ldots, x_n\}$ with $n = |X|$ is the domain of all f_i and $Y = \{y_1, y_2, \ldots, y_{|Y|}\}$ the codomain. Related to this (fixed) indexing, the k-th *function column* fc_k is the set of all function values, to which x_k is mapped: $fc_k = \{f_1(x_k), f_2(x_k), \ldots, f_N(x_k)\}$. The set of all function values of a function f_i will be denoted by $y(f_i) = \{f_i(x_1), f_i(x_2), \ldots, f_i(x_n)\}$.

With a we denote a deterministic, non-repeating algorithm, which is applicable to any f_i mapping X to Y. The specification is given by a suite of mappings from partial sampling sequences to x-values not sampled so far. In more detail: initially, an algorithm, if applied to a function f, starts with the (deterministic) choice of an element x_{a_1} of X. Then, f provides the function value $y_{a_1} = f(x_{a_1})$. Thus, algorithm a initially builds the sampling list $S_1 = ((x_{a_1}, y_{a_1}))$ and computes the next sampling value $x_{a_2} \neq x_{a_1}$ from S_1 by $x_{a_2} = a[S_1]$. With $y_{a_2} = f(x_{a_2})$ we can extend the sampling list $S_2 = S_1 + (x_{a_2}, y_{a_2})$ with "+" standing for list appending. The next sampling point is a function of S_2, which is given by the specification of the algorithm: $x_{a_3} = a[S_2]$. In the $(k+1)$-th step ($k < N$) of the algorithm, we append to the partial sampling list $S_k = ((x_{a_1}, y_{a_1}), \ldots, (x_{a_k}, y_{a_k}))$ the next sampling point $x_{a_{k+1}}$, derived by the algorithm from S_k, and which is different from all x_{a_i} with $i = 1, 2, \ldots, k$ evaluated so far, and the corresponding function value $y_{a_{k+1}} = f(x_{a_{k+1}})$. Once $k = N - 1$, the algorithm terminates in the next step. Now, the set of y values $\{y_{a_1}, y_{a_2}, \ldots, y_{a_n}\}$ has to be a permutation of the function values $\{f(x_1), f(x_2), \ldots, f(x_n)\}$ of f. In that sense, an algorithm can be specified by stepwise constructing a permutation of the (unknown) function values. We denote the ordered set of function values sampled by the algorithm a when applied to f with $y_a(f)$.

An important fact is that for different functions f_a and f_b, these permutations have to be different. Assume that an algorithm a samples the y-values of two functions f in the same order. By induction, we can show that in such a case the functions are equal (among other, see [5] for the proof).

Lemma 3. *For any algorithm a and any two functions f_a and f_b $y_a(f_a) = y_a(f_b)$ iff $f_a = f_b$.*

This lemma is a precursor for the Generalized No Free Lunch Theorem. Thus, once an algorithm a is applied to *all* N functions f_i ($i = 1, \ldots, N$) step by step, the ordered set of sampling sequences $\{y_a(f_1), \ldots, y_a(f_n)\}$ is a permutation of the ordered set of function mappings

$$Y/X = \{y(f_1), y(f_2), \ldots, y(f_N)\}.$$

It comes out that an algorithm is fully specified either

- by the choice of x_{a_1} and the complete set of mappings $x_{a_{k+1}} = a[S_k]$ for $k = 2, ..., n-1$, or
- by the permutation of Y/X.

Using the first specification, we can compute the total number of different algorithms, which can be applied to all mappings $f_i : X \to Y$ with $1 \leq i \leq N$. For doing so, we simply

follow a scheme of stepwise assignments. Consider the case $X = \{x_1, x_2, x_3, x_4\}$ and $Y = \{0, 1\}$ and the scheme:

x_{a_1}	$x_{a_{21}}$	$x_{a_{31}}$	$x_{a_{41}}$
			$x_{a_{42}}$
		$x_{a_{32}}$	$x_{a_{43}}$
			$x_{a_{44}}$
	$x_{a_{22}}$	$x_{a_{33}}$	$x_{a_{45}}$
			$x_{a_{46}}$
		$x_{a_{34}}$	$x_{a_{47}}$
			$x_{a_{48}}$

In the first step, an algorithm has to specify an index a_1. Then, the value of $f(x_{a_1})$ can be either 0 or 1. In the second step, the algorithm selects one of the three remaining x-values, depending on the value of $f(x_{a_1})$. If it is 0, the algorithm selects as second x-value $x_{a_{21}}$, if it is 1 then $x_{a_{22}}$. In the third step, the algorithm a selects one of the two remaining x-values. Now, there are the four cases that the first two function values sampled were either $(0, 0)$, $(0, 1)$, $(1, 0)$ or $(1, 1)$. In each of these four cases, the algorithm may select a different x-value. Then, for the fourth step, only one x-value remains, so the selection is unique.

So, in this case we can completely specify an algorithm a by providing the 7 indices $a_1, a_{21}, a_{22}, a_{31}, a_{32}, a_{33}$ and a_{34}. For a_1 there are four choices, since $|X| = 4$. For a_{21} as well as for a_{22} we can choose one out of the three remaining x-values (including the case that $a_{21} = a_{22}$), giving $3 \cdot 3 = 9$ possibilities, and finally 2^4 possibilities for the third step. So, the total number of assignments is $4 \cdot 9 \cdot 16 = 576$.

This scheme can be easily extended to the general case. In the first step, we have $n = |X|$ choices for the index a_1 of the first x-value. According to the $|Y|$ possible values of $f(x_{a_1})$ there are $|Y|$ independent cases, for which a second index a_2 from the $(|X| - 1)$ remaining indices has to be selected. This gives $(|X| - 1)^{|Y|}$ choices for the second step. For the k-th step, one has $|Y|^{k-1}$ cases, and in each case one can choose one from the remaining $(|X| - (k - 1))$ indices of x-values. So, we may formulate the following theorem:

Theorem 3. *The number of different algorithms that can be applied to all functions mapping a finite set X to a finite set Y is given by*

$$N_a = \prod_{k=0}^{|X|-1} (|X| - k)^{\left(|Y|^k\right)}. \tag{2}$$

Proof. That equation 2 is an upper bound for the number of algorithms has been demonstrated in the foregoing paragraph. It remains to show that any two different assignments of indices to

$$a_1, a_{21}, \ldots, a_{a|Y|}, a_{31}, \ldots, a_{3|Y|^2}, \ldots, a_{|X|1}, \ldots, a_{|X||Y|^{|X|-1}}$$

will indeed give two different permutations of Y/X, hence two different algorithms.

For showing this, consider the first k elements $\Pi = (\pi_1, \pi_2, \ldots, \pi_k)$ of a permutation of $(x_1, x_2, \ldots, x_n)$ and a set $(y_1, y_2, \ldots, y_k)$ of k elements of Y. Now, i and j shall be two different of the remaining indices, i.e. $x_i \notin \Pi, x_j \notin \Pi, i \neq j$ (therefore $k < n - 1$).

Now consider the set

$$M = \{l \,|\, f_l(\pi_1) = y_1, f_l(\pi_2) = y_2, \ldots, f_l(\pi_k) = y_k\}$$

(with $1 \leq l \leq N$) and for the index i and a fixed $y_a \in Y$ its subset

$$M_1 = \{l \,|\, l \in M \wedge f_l(x_i) = y_a\}.$$

The set M_1 has exactly $|Y|^{|X|-(k+1)}$ elements l, and for all of them is $f_l(x_i) = y_a$ per definition.

Next, since $k \leq n - 1$, we consider the set

$$M_2 = \{l \,|\, f_l(\pi_1) = y_1, \ldots, f_l(\pi_k) = y_k, f_l(x_i) = y_a\}$$

(again $1 \leq l \leq N$) Obviously $M_1 = M_2$, but M_2 contains only $|Y|^{|X|-(k+2)}$ elements l, for which $f_l(x_j) = y_a$.

Defining for any index set I the operator C_i with

$$C_i(I) = \{f_j(x_i) \,|\, j \in I\}$$

then this means $C_i(M_1) \neq C_j(M_2)$ and using $M_2 = M_1 \subseteq M$ it follows

$$C_i(M) \neq C_j(M)$$

However, while specifying the algorithm a we had to assign the next testing point x_i for any such M, so different assignments for i and M will give different (partial) outcomes $C_i(M)$. Thus in general, different assignments will specify different algorithms and the number of algorithms equals the number of possible assignments. □

To illustrate the proof, we take as an example $X = \{x_1, x_2, x_3\}$ and $Y = \{0, 1\}$, thus having eight possible functions that are listed below.

l	x_1	x_2	x_3
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1
5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

The table shows the case $k = 1, \pi_1 = x_2, y_1 = 1$ and $i = 1, j = 3$. The set M of all indices l for which $f_l(x_2) = 1$ is $M = \{3, 4, 7, 8\}$. Then, M_1 is the subset of indices l of M for which also $f_l(x_1) = 1$ holds. This is the set $M_1 = \{7, 8\}$. The set M_2 is the set of indices l for which $f_l(x_2) = 1$ and $f_l(x_1) = 1$, which is also the set $M_2 = \{7, 8\}$. As we can see, the set $C_1(M_1) = \{1, 1\}$ contains two times the value 1 $(2 = |Y|^{|X|-(k+1)} = 2^1)$, while the set $C_3(M_2) = \{0, 1\}$ only contains one times the value 1 $(1 = |Y|^{|X|-(k+2)} = 2^0)$. So, any algorithm specified by selecting x_2 at first, and then

x_1 in case $f(x_2) = 1$ will have a different outcome of y-values than an algorithm, which selects x_2 first and then x_3 in case $f(x_2) = 1$. The proof basically generalizes this approach.

IV. On algorithm instances

In the foregoing section, we computed the number N_a of different samplings of algorithms, when applied to all possible functions f. Now, if we consider a particular algorithm, usually such an algorithm can provide more than one of such samplings, depending on its configuration and runtime parameters. We will refer to such different samplings as algorithm *instances* and are looking for the number of instances that a given algorithm can have, in comparison to the maximum number N_a of possible instances.

In particular we are considering the class of algorithms, which are using pseudo-random numbers to perform their internal computations. Among them we find random search, algorithms using mutation operation (like hill-climbing) and algorithms using recombination (or crossover) operations, like evolutionary algorithms. Such algorithms can be modeled in similarity to the concept of a very simplified Turing machine: assume that there is a linear tape with a sequence of numbers (referring to configuration parameters, runtime parameters or pseudo-random numbers). The tape is read out and moved forward each time the algorithm needs one or more pseudo-random numbers to specify one of its operations (e.g. to apply mutation onto a bitstring). Thus it will be said that an algorithm performs an operation "randomly". If we consider different tapes, the application of the algorithm may achieve different instances, and if there is a distribution function for the values on the tape, there will be a distribution function for the algorithm instances as well.

We may reformulate our question about the number of algorithm instances by using a design viewpoint: if we want to achieve a given instance of an algorithm, can we provide an appropriate sequence of numbers on the tape?

The simplest case here is the random search: in the decision rule $x_{k+1} = a[S_k]$, the value of a will only depend on k and never on y_k. This means, a random search is not directed by the y-values sampled so far. So, it can be easily seen that random search has exactly $n!$ instances, each of which corresponds to a permutation of the sample points $\{x_1, x_2, \ldots, x_n\}$. This is obviously a small number, if compared to the maximum number N_a of possible instances. So, the question is if we can obtain a larger number of instances or even all, and the positive answer will come from the evolutionary approach.

To show this, we will consider the following model situation: given is a set of functions that has not been distinguishable so far by an algorithm a for k steps, but for step $(k+1)$, for the first time we will get different results. We assume f_1 and f_2 to be two such functions, i.e. $S_k(f_1) = S_k(f_2)$, thus following $x_{k+1}|_{f_1} = x_{k+1}|_{f_2}$ but $f_1(x_{k+1}) \neq f_2(x_{k+1})$. So, if we want to have an algorithm a to be able to provide all possible instances, it is necessary that the algorithm has to be able to decide on a different x_{k+2} for any different value $f(x_{k+1})$. The more different sample points x_{k+2} can be selected by a out of the remaining x-values, the more instances the algorithm will have.

We are considering the class of *population-based* algorithms. Such algorithms maintain a varying set of x-values (the population) and base all decisions for the next sampling point and on the next population on the y-values of these x-values and the pseudo-random numbers on the tape alone. The problem of initializing such a population will be discussed below.

For our model situation, after k steps we have a population of size l with elements $(x_{a_1}, x_{a_2}, \ldots, x_{a_l})$. Since we are considering a function set that was not distinguishable by the algorithm so far, the algorithm will decide on the same next sampling point $\tilde{x}$ for all these functions as well. However, we assumed to have two f_1 and f_2 in our function set with $f_1(\tilde{x}) \neq f_2(\tilde{x})$. Altogether, there are $|Y|$ possible values for $f(\tilde{x})$.

Now we introduce three operators:

1) *Selection*: By using the next number(s) from the tape, and the values $f(x_{a_i})$ and $f(\tilde{x})$, an index $1 \leq i \leq l+1$ is computed and used to select x_{a_i}, or $\tilde{x}$ in case $i = l+1$.
2) *Mutation*: Given any x_a, by using next number(s) from the tape this sample point is transformed into any other $x_b \in X$.
3) *Recombination*: Given any pair of x-values x_a and x_b, by using next value(s) from the tape a value x_c is assigned to x_a and x_b. In contrary to mutation, the set of attainable values x_c can be restricted (as e.g. the crossover operation on two bitstrings can not result into any bitstring).

Based on such operation, we consider two algorithms: *hill climbing* (HC) as the iterated application of selection and mutation, and *evolutionary algorithm* (EA) as the combination of selection, selection, recombination and mutation[1]. Now, we can discuss both algorithms to learn about their attainable number of instances.

- Case HC: Here, the selection can only depend on the value $f(\tilde{x})$, since everything else went equally so far for two functions f_1 and f_2 and this will give one out of $\min(|Y|, l+1)$ different x-values (either one from the population, or $\tilde{x}$, but not more than possible function values of $f(\tilde{x})$). Then, mutation can transform the selected x-value into any other x value (including the avoidance of mutating into any x-value that was sampled already). Thus, an hill climbing algorithm can maximally decide on one of $l+1$ possible choices for the next step, but there are in general $|Y|$ possible cases to consider. This means, once $|Y| > l+1$, an hill climber can not provide all algorithm instances anymore.
- Case EA: For two selections, there are $\min^2(|Y|, (l+1))$ possible results, each of which may give a different next sample point after applying the recombination operation. However, not all possible x-values can be obtained this

[1]Yes, in such a framework we ignore the fitness function completely. Here, fitness has to be considered a technical means to perform the operations.

way, so a further mutation is needed to transform them into any other desired sample point (again excluding repetitions). It follows that a necessary condition for an EA to provide all possible algorithm instances is $|Y| \leq (l+1)^2$. As we can see, this is a smaller effort than for the case of an HC, who needs at least a population of size $|Y| - 1$ to achieve the same goal.

To obtain a sufficient criteria, we have to consider the initialization of an algorithm: the standard approach is to perform a random search (RS) for l steps. It can be easily seen that this procedure makes some algorithm instances inaccessible by the algorithm, and, for large sizes of $|X|$ and $|Y|$, the number of instances effectively vanishes against the total number of algorithms. Instead of this, we may consider a "smart" *diversity initialization* (DI) to maintain a higher diversity (despite of the fact that it does not seem to be useful for HC or EA itself): select first element of the population x_{a_1} randomly, then make the selection of the next element depending on $f(x_{a_1})$ and do the same for the next initial population x-values until population has grown to size l. This makes sure that we will not loose any sampling sequence at the beginning. Once using DI and making sure that $(l+1)^2 \geq |Y|$, an EA becomes able to provide (by chance, or by providing the needed values on the tape) *all* possible instances (as well as for an HC with $(l+1) \geq |Y|)^2$.

So, we may conclude this section by providing a kind of ranking among three famous algorithm classes according to their number of different possible instances:

$$N_{RS} \ll N_{HC} \leq N_{EA} \leq N_a$$

From the foregoing discussion it can also be seen that the ability of an algorithm to provide more instances can be increased by a higher "socialization" of the operators, i.e. by using higher order operators than mutation (order 1) or recombination (of order 2).

V. Discussion

Finally, we will shortly discuss the reasonable question: Is it useful for an algorithm to have more instances than another algorithm? To approach an answer here, we consider all algorithm instances a_i of a particular algorithm a. Caused by the distribution of its configuration parameters and pseudo-random numbers, the algorithm will instantiate any a_i with a probability w_i with $\sum_i w_i = 1$. From the No-Free-Lunch theorem, it can be easily seen that different values of w_i will not have any influence of the performance of a, even if it is taken over all of its instances. However, the variance of the performances can be different! Even more, the minimum variance can only be achieved when all w_i are equal. In case an algorithm has a limited number of instances, it also achieves its minimum variance when all instances are equally likely, but any algorithm with more instances can achieve a smaller variance. So, if we measure variance e.g. by the expression $\sigma = \bar{x^2} - \bar{x}^2$, and a achieves a maximum of k instances, then minimum variance is

$$\sigma_{min} = \frac{1}{k \cdot N_a} - \frac{1}{N_a}$$

which is achieved iff all k probabilities w_i are equal to $1/k$. The higher k, the smaller the minimum variance. Thus, by comparing variances of algorithms we get, basically, the same ranking as for the number of instances. While there is no procedure to estimate the number of algorithm instances, the variance of its performance can be estimated by repeated application of the algorithm with random "tapes" to a random set of functions.

Some final remarks:

1) The No-Free-Lunch theorems points out the equal average performance of any algorithm, once applied to any possible function. In a practical sense, it may be even of interest to have an algorithm, which is coming close to this average at all. If there is a smaller variance in the algorithm outcome, it is more likely that the algorithm is closer to this average. On the opposite, the "worst" algorithm, random search will have a high variance in the results, with a few excellent results and lots of failures. This is reflected by the variance measure.
2) The viewpoint used in this paper on discussing the number of algorithm instances can be extended to other algorithms. However, the use of pseudo-random numbers in algorithms like EA and HC comes out to be of an advantage to be (at least theoretically) able to provide a large number of instances.
3) An alternate performance measure for algorithms can thus be described as follows: apply the algorithm under varying setups and differing pseudo-random numbers, each time on a random set of quality functions, and get an estimate for the variance of the algorithm performance. The fewer this variance, the "better" the algorithm (i.e. the higher the number of instances that can be expected, or the more the algorithm is differing from random search, or the more it is likely that the algorithm can approach the average performance).

[2]It has to be noted that for such a viewpoint the size of the population is related to the size of Y and not the size of X

References

[1] D. W. Corne and J. D. Knowles, "No Free Lunch and Free Leftovers Theorems for Multiobjective Optimization Problems" in *Evolutionary Multi-Criterion Optimization (EMO 2003) Second International Conference*, Faro, Portugal, April 2003, Proceedings, pp. 327-341, Springer LNCS, 2003.

[2] T. M. English, "Optimization is Easy and Learning is Hard in the Typical Function," in *Proceedings of the 2000 Congress on Evolutionary Computation (CEC 2000)*, A. Zalzala et al., Eds., La Jolla, CA, USA, pp. 924-931, 2000.

[3] Ch. Igel and M. Toussaint, "On Classes of Functions for which No Free Lunch Results Hold," *Information Processing Letters 86*, pp. 317-321, 2003.

[4] M. Köppen and D. W. Wolpert and W. G. Macready, "Remarks on a Recent Paper on the No Free Lunch Theorems," *IEEE Trans. Evol. Comp.*, vol. 5(3), pp. 296-296, 2000.

[5] M. Köppen, "On the Benchmarking of Multiobjective Optimization Algorithms," in *Knowledge-Based Intelligent Information Systems, 7th Intl. Conference, KES2003*, Oxford, UK, September 2003, V. Palade et al., Eds., Proceedings, Springer LNAI 2773, pp. 379-385, 2003.

[6] N. J. Radcliffe and P. D. Surry, ”Fundamental Limitations on Search Algorithms: Evolutionary Computing in Perspective,” in *Computer Science Today: Recent Trends and Development*, J. van Leeuwen, Ed. LNCS vol. 1000, pp. 275-291, 1995.

[7] C. Schumacher and M. D. Vose and L. D. Whitley, ”The No Free Lunch and Description Length,” in *Genetic and Evolutionary Computation Conference (GECCO 2001)*, L. Spector et al, Eds. San Francisco, CA, 2001.

[8] D. Whitley, ”A Free Lunch Proof for Gray versus Binary Encodings,” in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, W. Banzhaf et al., Eds., Orlando, FL, USA, pp. 726-733, 1999.

[9] D. W. Wolpert and W. G. Macready, ”No Free Lunch Theorems for Search,” *Technical Report SFI-TR-05-010, Santa Fe Institute*, Santa Fe, NM, USA, 1995.

[10] D. W. Wolpert and W. G. Macready, ”No Free Lunch Theorems for Optimization,” *IEEE Trans. Evol. Comp.*, vol. 1(1), pp. 67-82, 1997.