Focused No Free Lunch Theorems

Darrell Whitley
Computer Science Department
Colorado State University
Fort Collins, CO USA 80534
whitley@cs.colostate.edu

Jonathan Rowe
School of Computer Science
University of Birmingham
Birmingham, Edgbaston, B15 2TT, UK
J.E.Rowe@cs.bham.ac.uk

ABSTRACT

Proofs and empirical evidence are presented which show that a subset of algorithms can have identical performance over a subset of functions, even when the subset of functions is not closed under permutation. We refer to these as *focused sets.* In some cases focused sets correspond to the orbit of a permutation group; in other cases, the focused sets must be computed heuristically. In the smallest case, two algorithms can have identical performance over just two functions in a focused set. These results particularly exploit the case where search is limited to m steps, where m is significantly smaller than the size of the search space.

Category and Subject Descriptors: I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, Search

General Terms: Theory, Algorithms

Keywords: Theory, Algorithms

1. INTRODUCTION

This paper presents both theoretical and empirical results which demonstrate that there can exists a "Focused No Free Lunch" when a small set of algorithms are compared on a small set of functions; we will define this set of functions to be the *focused set.* Focused No Free Lunch results sometimes exploit the fact that practical search algorithms are limited to a small number of sample evaluations, denoted by m, in a search space that is typically exponentially large. In the smallest case, Focused No Free Lunch results holds over two algorithms limited to m steps and the resulting focused set contains only two functions.

Focused No Free Lunch theorems build on the Sharpened No Free Lunch theorem which shows that No Free Lunch holds over sets that are closed under permutation. However, unlike the Sharpened No Free Lunch theorem, Focused No Free Lunch theorems can hold over sets that are a subset of the permutation closure. In some cases, a closure exists which corresponds to the orbit of a permutation group. In this case, we leverage mathematical concepts from permutation groups to bound the maximal size of the focused closure. In other cases, particularly when search is limited to m steps, there can be many focused sets and we construct a focused set heuristically.

Ultimately, Focused No Free Lunch theorems are concerned with how researchers and practitioners use and compare search algorithms. If algorithm $A1$ is better than algorithm $A2$ on a benchmark β, the Sharpened No Free Lunch theorem tells us that if we compute the permutation closure of β (denoted by $P(\beta)$), then algorithm $A2$ is equally better than $A1$ on the set $P(\beta) - \beta$ in the aggregate. The problem is that usually the size of β is small and $P(\beta) - \beta$ is enormous. Focused No Free Lunch results show that there can exists a *focused set* denoted by $C(\beta)$ such that when $A1$ is better than $A2$ on β, then $A2$ is better than $A1$ on $C(\beta) - \beta$. In some cases, $C(\beta)$ can be quite small.

These results also address the concerns raised by Igel and Toussaint [3] and Streeter [6] who show many broad classes of problems (for example, consider ONEMAX, MAXSAT, trap functions, or N-K Landscapes) are not closed under permutation. Focused No Free Lunch theorems demonstrate that a subset of algorithms can also display identical performance over focused sets that are smaller than the permutation closure.

1.1 Background

Wolpert and Macready's original "No Free Lunch Theorems for Search" [9] can be summarized as follows:

> *For all possible metrics, no search algorithm is better than another when its performance is averaged over all possible discrete functions.*

Schumacher et al. [5] sharpened the No Free Lunch theorem by formally relating it to the **permutation closure** of a set of functions. Let X and Y denote finite sets and let f: $X \longrightarrow Y$ be a function where $f(x_i) = y_i$. Let σ be a permutation such that $\sigma : X \longrightarrow X$. We can permute functions as follows:

$$\sigma f(x) = f(\sigma^{-1}(x))$$

Since $f(x_i) = y_i$, the permutation $\sigma f(x)$ can also be viewed as a permutation over the values that make up the codomain (the output values) of the objective function.

We can now define the permutation closure $P(F)$ of a set of functions F.

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

$$P(F) = \{\sigma f : f \in F \text{ and } \sigma \text{ is a permutation}\}$$

This provides the foundation for the following result.

THEOREM 1. *For any two arbitrary algorithms A and B, the performance of A and B will be identical over a set of functions if and only if that set of functions is closed under permutation.*

Proofs are given by Schumacher et al. [5]. Related proofs are given by by Droste, Jansen and Wegener [1]. One thing that should be clearly stated about current No Free Lunch theorems is that these theorems addresses questions about *all* possible algorithms and all possible functions, or all possible functions in a permutation closure. Focused No Free Lunch theorems address questions about the specific behavior of a specific subset of algorithms.

Schumacher et al. [5] and Whitley [8] also point out that when No Free Lunch holds over sets of function *closed under permutation*, the permutation closure may be compressible or the permutation closure may be incompressible. English [2] noted that NFL can hold over sets of functions such as needle-in-a-haystack functions, which are both closed under permutation and compressible.

Igel and Toussaint [3] show that if one considers all the possible ways that one can construct subsets over the set of all possible functions, then those subsets that are closed under permutation is a vanishing small percentage of all possible subsets.

On the other hand, Droste et al. have also shown that for any function for which a given algorithm is effective, there exists related functions for which the performance of the same algorithm is substantially worse [1].

So when is one algorithm really better than another? What do experimental evaluations of algorithm really tell us about comparative performance?

Over the next few sections, this paper presents examples of Focused No Free Lunch results.

1.2 Gray and Binary Representations

The section presents an example of a Focused No Free Lunch result that relates to Binary and Gray representations. This result is already well known.

Consider a benchmark test function, $f_b : \{0, 1, \ldots, 2^L - 1\} \rightarrow R$, where R is some finite subset of real numbers and L a positive integer. Let algorithm A_G be a search algorithm using a bit encoding with a Gray code representation. Let algorithm A_B use the same search algorithm except A_B uses a Binary representation. (It does not matter what search algorithm is used, as long as it uses a bit representation of a parameter optimization problem.)

We now ask a normal No Free Lunch question: over what sets of functions do the two algorithms have the same performance?

The Sharpened No Free Lunch theorem proves that A_G and A_B will have the same performance when compared over the permutation closure of f_b. The same result also follows from Whitley's NFL for binary and gray representations [7], and from Radcliff and Surry's representation results [4].

However, because we are talking about two specific algorithms, not just any two arbitrary algorithms out of the space of all possible algorithms, there is a more focused result. Let string s_b be any binary string; we can also treat s_b as a column vector. There exists matrix M such that $(s_b)^T M = s_g$ where the addition is mod 2 and s_g is the Gray encoding corresponding to s_b. We can also "Gray" a string multiple times. Thus $(((s_b)^T M)M) = (s_b)^T M^2$ produces a string to which Gray coding has been applied twice.

It is straight forward to prove for strings of length L that there exists an integer $L \leq i < 2L$ such that $M^i = I$ where I is the identity matrix.

This means that for *any* function f_b there exists a focused set of functions (which are a subset of the permutation closure), with less than $2L$ members that can be generated by repeated application of Gray code.

The following is an example for a 3 bit representation

s_b :	000	001	010	011	100	101	110	111
$(s_b)^T M^1$:	000	001	011	010	110	111	101	100
$(s_b)^T M^2$:	000	001	010	011	101	100	111	110
$(s_b)^T M^3$:	000	001	011	010	111	110	100	101
$(s_b)^T M^4$:	000	001	010	011	100	101	110	111

where $(s_b)^T M^4 = s_b$.

If we then interpret all of the bit strings as standard binary strings, then we can use the following permutation of indices into f_b to create a subset of 4 functions.

P1:	0	1	2	3	4	5	6	7
P2:	0	1	3	2	6	7	5	4
P3:	0	1	2	3	5	4	7	6
P4:	0	1	3	2	7	6	4	5

We can convert these permutations into functions as follows. Let σ_i be the i^{th} element in a permutation σ. A new function g is generated by the mapping $g(i) = f_b(\sigma_i)$.

This produces a focused subset of 4 functions, such that the performance of A_G and A_B will be identical when compared over all functions in the focused subset. Technically, a *group* has been defined that is closed with respect to the application of M to the bit representation.

This group has two cycles of length 1, a cycle of length 2, and a cycle of length 4, which can be denoted by:

(0)(1)(2 3)(4 6 5 7), or more concisely (2 3)(4 6 5 7)

Closure is achieved when all of the cycles synchronize. The orbit of the permutation cycle defines both the size of the closure and the time required for the permutation cycles to synchronize. Since 2 and 1 are factors of 4, the orbit is 4.

Note that the group action just described can be viewed in three ways: a subset of functions is generated; a subset of different representations is generated, or a subset of different algorithms is generated. When this process is viewed as a change in representation, the Focused No Free Lunch result holds even if the search algorithms are nondeterministic.

This result can be extended to any pair of algorithms whose only difference is a choice of representation (of whatever the search space happens to be). Since a change in representation is the same as applying a bijection σ to the domain, this means that such a pair of algorithms $A1$ and $A2$ have the property that running $A1$ on f is the same as running $A2$ on σf, for all functions f. Then given any particular function f, we get a focused set $f, \sigma f, \sigma^2 f, \ldots$, whose size is the same as the order of σ (i.e. the smallest integer k such that σ^k is the identity). In the Gray-Binary case, σ is calculated using the M matrix, whose order is $< 2L$.

2. FOCUSED NO FREE LUNCH

The thesis of this paper is that many such examples of "Focused No Free Lunch" exist. In particular, when comparing one algorithm $A1$ to another algorithm $A2$, what we should care about is the not the permutation closure, but rather what other smaller closures or *focused sets* may exist.

Permutation groups are one mechanism that leads to focused sets. Permutation cycles can show up in various ways when comparing algorithms. We next look at this on a special class of algorithms.

2.1 Path-Search Algorithms

We first define a special class of algorithms called *path-search algorithms*.

DEFINITION 1. *A* path-search *algorithm searches a function using a predefined sequence of points in the domain X. The sequence of points determines a permutation of X.*

This means a path-search algorithm does not use any information about the set of codomain values that are observed during search. Given N values in the domain of a function, there are N! search path algorithm.

Path-search algorithms are strongly deterministic in as much as their behavior (and therefore their performance) is predetermined before search begins. This also means the path of a path-search algorithm and the resulting trace of the codomain values is strongly coupled as well.

Given two path-search algorithms, there is a unique set of cycles that describes the difference in the behaviors of the two path-search algorithms. This is not always true for other search algorithms. For now, this simplifies our discussion of permutation cycles.

We next construct an example using the domain $X = \{1, 2, 3, 4, 5, 6, 7\}$ and codomain $Y = \{a, b, c, d, e, f, g\}$ where a to g will represent any arbitrary real values. Consider the following path-search algorithms:

$$A_x =< 1\ 2\ 3\ 4\ 5\ 6\ 7 >$$
$$A_y =< 2\ 3\ 4\ 1\ 6\ 5\ 7 >$$

There again exists permutation cycles that define a closure: (1 2 3 4)(5 6)(7).

Thus, there are three cycles in the two permutations that define the two path-search algorithms. Since all three cycles are synchronized after a period of 4, there exists an orbit of 4 functions that define a closure. Given any test function f_1 we can define three more functions f_2 and f_3 and f_4 such that the performance of algorithm A_x and A_y will be identical over these four functions.

For example, if $f_1 =<$ a b c d e f g $>$ then:

$$f_1 =<\ \text{a b c d e f g}\ >$$
$$f_2 =<\ \text{b c d a f e g}\ >$$
$$f_3 =<\ \text{c d a b e f g}\ >$$
$$f_4 =<\ \text{d a b c f e g}\ >$$

Path-search algorithms A_x and A_y will have identical behaviors when compared over this set of functions.

We next formalize these ideas.

2.1.1 Path Search and Closure

Given finite sets X and Y, then Y^X is the set of all functions from X to Y. Given a deterministic black box search algorithm A for such functions then let $\mathrm{tr}_A(f)$ (the *trace* of A on f) be the sequence of codomain values generated as A searches through X. If we let T be the set of all possible sequence of codomain values, then one of the consequences of the NFL theorem is that $\mathrm{tr}_A : Y^X \to T$ is a bijection for all algorithms A. It follows that, given any two algorithms A and B that the function $(\mathrm{tr}_A)^{-1} \circ \mathrm{tr}_B$ is a permutation of the set of functions Y^X. There are, of course, many possible permutations of this set, and they form a *group* under function composition.

DEFINITION 2. *Let $\pi : X \to X$ be a bijection. Then the map $v_\pi(f) = f \circ \pi$ is a permutation of Y^X called a* value-permutation.

The set of all value-permutations forms a subgroup of permutations of Y^X. This kind of permutation appears in the Sharpened NFL, which states that any set $S \subseteq Y^X$ is closed under value-permutations if and only if $\mathrm{tr}_A(S) = \mathrm{tr}_B(S)$ for all algorithms A, B. However, if we only consider some subset of algorithms, then S needn't be closed under value-permutations for the result to hold.

LEMMA 1. *Given a subset $\mathcal{A}$ of algorithms, let G be the group of permutations of Y^X generated by maps of the form $(tr_A)^{-1} \circ tr_B$, for pairs of algorithms $A, B \in \mathcal{A}$. Then for a given function $f : X \to Y$, the* orbit *of f under G, defined by*

$$Gf = \{f' | f' = g(f) \text{ for some } g \in G\}$$

has the property that $tr_A(Gf) = tr_B(Gf)$ for all $A, B \in \mathcal{A}$. Moreover, it is the smallest subset containing f for which this is the case.

PROOF. $\mathrm{tr}_A(Gf) = \mathrm{tr}_B(Gf)$ follows immediately from the definition of the group G. Now let $S \subseteq Y^X$ with $f \in S$ such that $\mathrm{tr}_A(S) = \mathrm{tr}_B(S)$ for all $A, B \in \mathcal{A}$. It follows that $S = g(S)$ for all $g \in G$. Therefore $g(f) \in S$ for all $g \in G$ which shows that $Gf \subseteq S$. □

The orbit of a function f (corresponding to some set of algorithms $\mathcal{A}$) is the smallest set of functions containing f for which all the algorithms in $\mathcal{A}$ have equal performance. The orbits corresponding to different functions partition the set Y^X. The union of two such orbits is another set for which the algorithms have equal performance. We therefore get a lattice (under set inclusion) of sets of functions for which NFL results hold for a given set of algorithms.

Of course, the orbit Gf must be contained within the value-permutation closure of f, as the following lemma verifies.

LEMMA 2. *Given a permutation of the form $(tr_A)^{-1} \circ tr_B$ and a function $f : X \to Y$, there is a value-permutation v_π such that $(tr_A)^{-1} \circ tr_B(f) = v_\pi(f)$.*

PROOF. Let $g = (\mathrm{tr}_A)^{-1} \circ \mathrm{tr}_B(f)$. Then the trace of A on g is the same as the trace of B on f. Since the trace lists all the codomain values of a function, it follows that g and f are value-permutations of each other. □

From this result, the Sharpened No Free Lunch theorem follows.

COROLLARY 1 (SHARPENED NFL). *If G is the group generated by all deterministic black box algorithms on Y^X and V is the group of value-permutations, then $G(F) = V(F)$ for any set of functions $F \subseteq Y^X$. In other words, the NFL result holds over a set of functions if and only if it is closed under (value-)permutations.*

It may well be the case that the orbit is by no means the whole of the value-permutation closure. Consider the example where there are just two algorithms $A, B \in \mathcal{A}$. Then the group G is generated by the single element $z = (\mathrm{tr}_A)^{-1} \circ \mathrm{tr}_B$ and is a cyclic group. If f is a one-one function then the size of the orbit of f is the order of the element z. That is, it is the smallest integer k for which z^k is the identity. Examples can be given where this is considerably smaller than the size of the value-permutation closure, as follows.

We now turn again to *path-search* algorithms.

LEMMA 3. *Given two path-search algorithms A, B the permutation $(tr_A)^{-1} \circ tr_B$ is a value-permutation.*

PROOF. Let π_A and π_B be the permutations of X that give the sequence of points searched by algorithms A and B respectively. Let $\pi = \pi_B \circ \pi_A^{-1}$. Without loss of generality, we think of the points of X as indexing the elements of the traces. Now given any f, let

$$f' = v_\pi(f) = f \circ \pi_B \circ \pi_A^{-1}$$

Then

$$\mathrm{tr}_A(f')_k = f'(\pi_A(k)) = f \circ \pi_B(k) = \mathrm{tr}_B(f)_k$$

Therefore $v_\pi(f) = (\mathrm{tr}_A)^{-1} \circ \mathrm{tr}_B(f)$ for all f as required. □

Now suppose our two algorithms are path-search algorithms. Then the generator of G is a value-permutation (that is, a permutation of X).

The maximum order of such a permutation is given by Landau's function. For example, when $|X| = 10$, then the biggest orbit is of size 30. In general, Landau's function grows with upper bound $O(e^N)$ which is considerably smaller than the size of the value-permutation closure which is $N!$. Since Landau's function is an upper bound, the actual orbit can be much smaller in many cases.

Lemma 3 looks at what happens when we compare just two path-search algorithms. It is reasonable to ask how many algorithms need to be considered before the orbit is in fact the same as the whole value-permutation closure. This depends, of course, on the choice of algorithms. Consider the domain $X = \{1, 2, 3, \ldots, n\}$. We define three path-search algorithms, A1, A2, A3, as follows:

A1: Search in order $1, 2, 3, \ldots, n$.

A2: Search the same as A1, but exchange the first two elements; i.e., search in order $2, 1, 3, 4, \ldots, n$.

A3: Search in shifted order $2, 3, 4, \ldots, n, 1$.

The group generated by these algorithms is the entire symmetric group of all permutations of X. The orbit of any function is therefore the full value-permutation closure.

There can also exist large collections of algorithms for which we *do not* get the full value-permutation closure. For example, one can use $n!/2$ different path-search algorithms corresponding to elements of the alternating group; by selecting this set of algorithms, an orbit results which is still half the size of the full value-permutation closure (assuming the function f we begin with is a one-one function).

2.2 Deterministic Search Algorithms

Clearly path-search algorithms are not practical search algorithms. Real search algorithms have a history of the points visited so far, and they use this information to decide where to sample next. In the remainder of this paper, we limit our attention to deterministic search algorithms.

This then raises two questions: 1) do cycles occur in real search algorithms that use history information, and 2) how do we determine if there is a closure that is smaller than the full permutation closure? Does it make a difference if we assume a search algorithm exhaustively explores the entire space (which is also not realistic) so that the search algorithm produce a trace over the entire codomain?

We know cycles occur in real search algorithms. If we compare any two search algorithms that are identical except one uses a Gray code representation and the other uses a Binary code, then there will exist a closure of not more than 2L functions over these two algorithms given an L bit representation. These cycles exist even if we exhaustively explore the entire search space.

We can induce similar cyclic patterns over other forms of local search. Assume that we have a deterministic local search algorithm A_1 that uses a regular neighborhood of size k. We then construct a second algorithm A_2 which is identical to A_1 except when A_1 samples neighbors in the order 1 to k, then algorithm A_2 will sample the neighbors in the order (2, 3, ... k, 1). After the first move, there will be $k-1$ unvisited neighbors (since the incumbent point was just visited by both algorithms). Algorithms A_1 and A_2 will induce cycles of length k and $k-1$ in the traces that are generated. The closure must be of at least size $k(k-1)$; it can be larger if the number of unvisited points in any particular neighborhood is less than $k-1$. But the orbit that is induced must be less than $k!$.

2.3 Algorithms limited to m steps.

What happens when deterministic algorithms are limited to m steps? Given the pragmatic constraints on search, we will assume that for an arbitrarily chosen problem the search space is exponentially large relative to the problem representation size and that m is polynomial. The size of the search space is denoted by N.

Sometimes (perhaps most of the time) algorithms that use history do not induce simple cycles. The *focused sets* over which two algorithms have identical performance for only m steps may not correspond to a formal closure, and there may exist focused sets that are not minimal in size. How do we then ask if there exists a Focused No Free Lunch result?

Consider any algorithm A_i applied to any function f_j. Let $Apply(A_i, f_j, m)$ represent a meta-algorithm that outputs a trace: it executes A_i on f_j and outputs the order in which A_i visits m elements in the codomain of f_j after m steps. We assume a step does not count previously sampled points. For every pair of algorithms A_k and A_i and for any function f_j, there exists another function f_l such that

$$Apply(A_i, f_j, m) \equiv Apply(A_k, f_l, m).$$

Viewed another way, for every pair of functions f_j and f_l and for any algorithm A_i, there exists another algorithm A_k such that $Apply(A_i, f_j, m) \equiv Apply(A_k, f_l, m)$. When $m = N$ and any 3 of the "variables" are determined, the 4th is also immediately determined.

This meta-algorithm can be used as an operator to generate traces. Under some circumstances, the traces also fully define a function. In other cases, the traces do not fully define a function, but rather specify a set of constraints that define a family of functions.

Assume we compare algorithm A_1 to algorithm A_2 on function f_0. This yields two traces which are generated by

$$Apply(A_1, f_0, m) \quad \text{and} \quad Apply(A_2, f_0, m)$$

For example, given A_1, A_2 and f_0 there exists at least two functions f_1 and f_2 such that:

$$Apply(A_1, f_0, m) \equiv Apply(A_2, f_1, m).$$

$$Apply(A_2, f_0, m) \equiv Apply(A_1, f_2, m).$$

If algorithms A_1 and A_2 exhaustively explores the space (i.e., $m = N$), then $Apply(A, f, m) = \mathrm{tr}_A(f)$ for function f. However, if $m < N$, then we only know that the traces are the same for the first m steps.

If by coincidence $f_1 = f_2$ then this subset of functions is closed under the *Apply* operator when $A1, A2$ and f_0 are fixed. Thus if $A1$ is better than $A2$ on function f_0 (or f_1) then $A2$ has identical but opposite performance relative to $A1$ on f_1 (or alternatively, f_0). Of course, we do not normally expect this to happen. But we might ask under what circumstances this does happen?

We will also construct a function called $APPLY$ which returns a trace of a function such that when $m = N$:

$$f_2 = APPLY(A, B, f_0, m) \iff Apply(B, f_0, m) \equiv Apply(A, f_2, m).$$

Again, if the entire space is exhaustively explored, then we have two notations that yield the same results: if $g = (\mathrm{tr}_A)^{-1} \circ \mathrm{tr}_B(f)$ then the trace of A on function g is the same as the trace of B on function f. However, when limited to m steps we can no longer have functions, but rather traces that define a family of functions.

When $m \neq N$ we allow $APPLY$ to accept a trace, t_0 as input. Thus, we want

$$t_1 = APPLY(A, B, t_0, m)$$

to be well-defined.

$APPLY(A, B, t_0, m)$ will sometimes assign a specific codomain value to the i^{th} element of trace t_1 and sometimes the value will be undetermined. Undetermined values can occur because algorithm A visits a point in the domain that has not been previously seen by algorithm B. We will assume that $APPLY$ assigns a variable which acts as a place holder for an unspecified codomain value to the i^{th} element of the trace if the value is undetermined. The assignment of a value to a variable that appears in multiple traces must be consistent across traces. For the reader wishing to see an example, this process is illustrated in subsection 2.5.

We will define a *potential function* as a set of traces which may contain variable representing undetermined values, as well as constraint information sufficient to determine the behavior of two or more algorithms given those traces. This means that $APPLY$ must execute algorithms on traces (acting as proxies for functions) with undetermined values, given that constraint information is provide sufficient to determine the behavior of the algorithms.

There are two ways we might do this. 1) We could define $APPLY$ to be a nondeterministic algorithm. In this case, $APPLY$ would nondeterministically selects *functions* which yield the desired traces; as a nondeterministic function, it can select these functions so that it produces the smallest focused set possible. This view allows us to again blur the distinction between traces and functions, and we do not have to worry about undetermined trace values. 2) We can make $APPLY$ a heuristic procedure that decides where a search algorithm is going to sample next given the trace it is trying to generate. To do this, $APPLY$ will need to add additional constraint information to the trace to allow the execution of

$$t_1 = APPLY(A, B, t_0, m)$$

to be both well defined and replicable. In the worst case, the heuristic may need to assign codomain values to the undetermined values, but it need never define a complete function. In other cases, it may only need to indicate information about the relative magnitude of undetermined values in the trace.

We can now define an algorithm to search for focused sets. BUILD.TRACES generates T, a set of traces; i and j index the set of traces in T.

BUILD.TRACES

```
1  i = 1;
2  j = 2;
3  t_i = Apply(A_2, f_1, m);
4  t_j = Apply(A_1, f_1, m);
5  T = t_1 ∪ t_2;
6  not-closed = CHECK(A_1, A_2, f_1, m, T);
7  While (not-closed)
8        {t_{i+2} = APPLY(A_1, A_2, t_i, m);
9         t_{j+2} = APPLY(A_2, A_1, t_j, m);
10        i = i + 2;
11        j = j + 2;
12        T = T ∪ t_i ∪ t_j;
13        not-closed = CHECK(A_1, A_2, f_1, m, T);
14       }
```

$APPLY$ adds enough additional information to the trace sufficient to allow an algorithm's behavior to be both deterministic and replicated. However, $APPLY$ has only a local point of view. A variable representing an undetermined codomain value may be propagated from one trace to another. CHECK executes the two algorithms over all of the traces to make sure all of the traces are feasible and compatible; it never needs to correct APPLY, but it can reduce the number of variables corresponding to undetermined values; this has the side-effect of changing T and the traces.

CHECK can decide that current traces t_i and t_j could be merged into one trace if the determined and undetermined codomain values are compatible and there are no conflicting constraints. If CHECK merges traces t_i and t_j into one trace, then sets the flag to terminate the loop.

CHECK can decide that t_i and t_j cannot be merged, but there exists some function g such that Algorithms A_1 and A_2 yield traces t_i and t_j when executed on g; in this case, CHECK also sets the termination flag.

After termination the number of traces in T can be odd or even. T will contain a set of traces representing behaviors over potential functions, such that the performance of algorithm A_1 and A_2 over the set of potential functions will be identical. Note that since there are two algorithms, each potential function must be compatible with two traces.

The *CHECK* and $APPLY$ functions must be constructed based on specific properties of the search algorithms.

2.4 Benignly Interacting Algorithms

In some cases, we do not require all of the machinery of BUILD.TRACES and can show that a focused set of just 2 traces exists. In this case, BUILD.TRACES terminates before executing the while loop. This result is independent of the algorithms that are used.

Let A_1 and A_2 be two deterministic algorithms that are executed on function f_1. We assume a method exists for indexing the domain. Assume each algorithm samples m points, with each domain sample indexed by $i = 1$ to m.

For the current discussion, a key aspect of deterministic algorithms is that the search paths explored by these algorithms may "intersect" in the sense of sampling the same domain values. However, given that m is dramatically smaller than the size of the search space, it is also possible that two algorithm do not "intersect" after m steps. What does this mean for the comparison of the two algorithms?

We construct two arrays: Array D stores the domain values $D_{i,j}$ sampled by algorithm j at step i. Array V stores the codomain values $V_{i,j}$ sampled by algorithm j at step i.

We next construct two new arrays, D^* and V^*. Note that V and V^* contain information about *two* traces constructed from the codomain values. Our goal is to combined the information contained in V^* into a single potential function. We construct V^* as follows.

$$\text{for } i = 1 \text{ to } m, \qquad V^*_{i,2} = V_{i,1} \quad \text{and} \quad V^*_{i,1} = V_{i,2}.$$

To assign values to D^*, we define a function to extend the domain trace (denoted by XDT) such that

$$D^*_{i,j} = XDT(i, j, D^*, V^*).$$

The function $XDT(i, j, D^*, V^*)$ executes algorithm A_j, using the provided domain values and codomain values from steps 1 to i-1. It returns the domain value that algorithm A_j will visit at step i. In effect, $XDT(i, j, D^*, V^*)$ simulates algorithm A_j; it uses the $i-1$ values in D^* and V^* to determine $D^*_{i,j}$.

We then iteratively construct D^* as follows:

$$\text{for } i = 1 \text{ to } m, \; D^*_{i,j} = XDT(i, j, D^*, V^*)$$

Note that D and D^* stores m elements for each algorithm. We will treat the elements of D and D^* as sets, and apply intersection and union operations.

$$D_{A1} = \bigcup D_{i,1} \quad \text{and} \quad D^*_{A2} = \bigcup D_{i,2}$$

$$V_{A1} = \bigcup V_{i,1} \quad \text{and} \quad V^*_{A2} = \bigcup V_{i,2}$$

For example, this denotes that D_{A1} is the union of the m domain values sampled by A_1 at each time step i.

Definition 3. *Two algorithms A_1 and A_2* benignly interact *with respect to a function f_1 and the construction of V^* if one of the following conditions hold:*

1) $(D^*_{A1} \cap D^*_{A2}) = \emptyset$.

2) if ($d \in (D^*_{A1} \cap D^*_{A2})$ and $d = D^*_{x,1} = D^*_{y,2}$) then $V^*_{x,1} = V^*_{y,2}$.

Note that condition 2 for benign interaction can occur in two ways.

If f_1 is a bijection then

$$((D^*_{A1} \cap D^*_{A2}) \neq \emptyset \text{ and } V^*_{x,1} = V^*_{y,2}) \iff x = y$$

Thus, if the function is a bijection and both algorithm sample the same domain value, they must visit that domain value at time step $x = y$ in order to produce the same codomain value at the same time in the traces V and V*. This produces a permutation cycle of one element in the two trace V and V*.

If f_1 is not a bijection and $x \neq y$, then as long as $V^*_{x,1} = V^*_{y,2}$ when $(D^*_{A1} \cap D^*_{A2}) \neq \emptyset$ then the same codomain values will still occur at the same time step in V and V^*.

We will use D^* and V^* to construct a new function f_2. The notion of benign interaction is a way of guaranteeing that either the algorithms A_1 and A_2 do not visit the same domain values, or if they do visit the same domain values, they also find the same codomain values at those locations.

Lemma 4. *Assume two algorithms A1 and A2 are executed for a polynomial number of time steps denoted by m on function f_1. If A1 and A2 "benignly interact" there exists a function f_2 such A_1 and A_2 have two equal but opposite traces when executed for m steps on f_1 and f_2; if f_1 is compressible, then there exists at least one function f_2 which is compressible. In general, when f_1 is a bijection and N is the size of the search space, there are $(N-m)!$ ways to construct f_2.*

Proof. Array V defines the behavior of algorithms A_1 and A_2 on function f_1. Using D^* and V^* we construct a second function f_2; when D^* does not contain a domain value we can make an assignment to that domain value randomly or we use f_1 to make the assignment.

By construction $V^*_{i,2} = V_{i,1}$ and $V^*_{i,1} = V_{i,2}$.

The construction is feasible and V^* captures the correct traces of A_1 and A_2 executed on f_2 given that A_1 and A_2 benignly interact on function f_1. If $(D^*_{A1} \cap D^*_{A2}) = \emptyset$ then then we are free to assign values to f_2 at all domain values using V* without conflict. If $(D^*_{A1} \cap D^*_{A2}) \neq \emptyset$ but $V^*_{x,1} = V^*_{y,2}$ because A_1 and A_1 benignly interact, then there is no conflict in the construction since the codomain values that appear in both V and V* occur at exactly the same time steps (and thus act as a permutation cycle of length 1).

In the construction of f_2 if domain points that are not in D^* are assigned codomain values randomly using codomain values that are not in V^*, then for a search space of size N, there are (N-m)! possible constructions when f_1 is a bijection.

In the construction of f_2 if f_1 is used to make the assignment to domain points not in D^*, f_2 is compressible when f_1 is compressible, since we only need a copy of f_1 and the arrays D^* and V^* which are of length m to construct f_2.

□

2.5 Constructing Other Sets

Assume two algorithms do not benignly interact and that it is not possible to construct a focused set of size 2. We can still then use BUILD.TRACES to find sets of potential functions where algorithms A_1 and A_2 produce exactly the same set of traces; thus, these potential functions define a

focused set given a specific function as a seed. In this section, we also need a heuristic version of *APPLY* to guide local neighborhood search.

We must worry about common (interacting) elements in the traces of algorithms A_1 and A_2 that represent the same codomain values at different positions in the trace.

Elements that do not interact provide flexibility in the construction of potential functions with the desired behavior. Heuristically, we want *APPLY* to construct a set of functions such that the traces produced by algorithms A_1 and A_2 move away from the interacting elements toward traces composed of undetermined codomain values.

In the following example we start with function f_1 which is a 6 bit Gray coded 1-D version of Schwefel's function. The function is a bijection, and the actual function values have been replaced by permutation values 1 to 64. Algorithm B is best-first local search and algorithm W is worst-first local search. The neighborhood size is 6 ($2^6 = 64$); both algorithms sample the same neighborhood in the same order. Algorithm B was started at location 000101 and algorithm W was started at location 001101. Starting at adjacent bit locations results in traces such that the two traces "interact" in non-benign ways. There are permutation cycles in the first few steps of the search.

We use BUILD.TRACES to construct a focused set of potential functions. However, as traces are built, the algorithms visit points in the domain that have not previously been visited when the algorithms were executed on f_1. The codomain values at these points are undetermined. We will represent these undetermined codomain values as free variables: there is nothing in the two original functions that forces these particular undetermined variables to take on any particular codomain value from among those that are not already assigned to some specific domain value. In this example, there are initially 39 free variables representing undetermined codomain elements (labeled a to z, and aa to nn). The traces produced are as follows.

```
                  Traces and Free Variables
 B   W        T2   T4   T6   T7   T5   T3   T1
==  ===     ================================
 1   3        3    8    a    i    z    2    1

 2   1        1    3    8    a    i    z    2+
 3   8        8    a    i    z    2    1    3
 4  13       13    b    j   aa    q   y-    4
 5  14       14    c    k   bb    r    x    5
 6  15      15+  17-   l-  cc-  hh-   w+    6
 7  16      16-   d+   e+   m+  ii+    8   7-

 8   7        7   15   17    l   jj    v    8
 9  17       17    e    m   dd    8    6    9
10  18       18    f    n   ee   kk    u   10
11  19       19    g    o   ff   ll    t   11
12  20       20    h    p   gg   nn    s   12
=============================================
B traces     f2   f3   f4   f5 <-f6 <-f7 <-f1
W traces     f1-> f2-> f3-> f4   f5   f6   f7

+ indicates best in neighborhood
- indicates worst in neighborhood
```

This table shows the original traces for best-first (B = T1) and worst-first search (W =T2) as columns. It also shows 5 additional traces (T3 to T7) generated by BUILD.TRACES. The last two rows assigns potential functions to traces. The last two rows also show the order in which BUILD.TRACES constructs the traces and potential functions (as indicated by the -> symbols). The potential functions are labeled f2 to f7. Each potential function must have a best-first trace (B) and a worst-first trace (W). Each trace must be preserved by two potential functions; thus each potential function shares traces with two other potential functions. Thus, potential function f1 has constraints that must also be preserved by potential functions f2 and f7. Note that f4 and f5 share trace T7, which is composed entirely of undetermined variables.

BUILD.TRACES converges on trace T7. In this case, BUILD.TRACES generates two fully undetermined traces, which CHECK merges together to form trace T7; thus the codomain values of T7 are undetermined. CHECK must still make sure the algorithms can execute correctly on the (underspecified) potential functions which are being constructed from the traces.

If traces contain undetermined codomain values, the traces may also need to 1) assign a candidate set of values to the traces to allow the algorithms to execute, or 2) indicate constraints over the undetermined codomain values to allow the algorithms to execute. The traces shown here indicate the best (+) and worst (-) neighbors. "Best" and "worst" were heuristically chosen in this example so as to (usually) produce the most undetermined neighbors possible in the output traces: this meant that generally the last two neighbors in the first neighborhood were chosen because these had neighbors farther away from the known codomain values.

For example, on potential function f5, best-first search and worst-first search impose the following constraints:

$$\text{m} < \text{a, z, aa, bb} < \text{cc}$$
$$\text{ii} < \text{i, 2, q, r} < \text{hh.}$$

In addition, note that element 2 appears as the best neighbor in f1, but not in f5. However, if element 2 is a global optimum, then there does not exist a codomain value better than element 2 and function f5 is not feasible; if 2 were a global optimum, we would need to define a deterministic restart (or backtrack) and continue generating traces.

From the Schwefel function, and considering only the domain and codomain values seen by the two algorithms B and W, we have the following correspondence to actual codomain values.

```
1 = 28      6 = 11      11 = 15     15 = 94
2 = 0.9     7 = 59      12 = 8      17 = 121
3 = 47      8 = 37      13 = 50     18 = 104
4 = 21      9 = 3.4     14 = 42     19 = 83
5 = 36     10 = 0.8     15 = 4      20 = 57
```

We can assign these codomain values to undetermined variables as long as none of the new assignments appear in the same potential function twice. Thus, the following assignment can be used to fill in most of the undetermined values in the traces. The 4 remaining undetermined values are denoted by a, b, c and d. This is illustrated in the following table.

```
      Traces and Free Variables
  T2     T4     T6     T7     T5     T3     T1
=============================================
  47     37     21     36     83    0.9     28

  28     47     37     21     36     83    0.9+
  37     21     36     83    0.9     28     47
  50     11    3.4     15    3.4    104-    21
  42     15     50     47     50     94     36
   4+   121-    94-   104-   121-     4+    11
  94-   0.9+   0.8+    11+   0.8+    37     59-

  59      4    121     28     59     37     37
 121      d     42      4      8     11    3.4
 104      c     59      c     42      c    0.8
  83      b      8      b      d      b     15
  57      a     57      a     57      a      8
=============================================
  f2     f3     f4     f5  <- f6  <- f7  <- f1
  f1 ->  f2 ->  f3 ->  f4     f5     f6     f7
```

Note that the table constructed using T is similar to array V used in the last section, except there are more than 2 potential functions. V can be generalized to be multi-dimensional. Note that the domain values that are visited by the algorithms are also known and can be used to construct a similar multi-dimensional structure D. Using the previously defined methods, and the new arrays V and D, we can easily construct a focused set of 7 compressible functions (compared to the 64! functions in the permutation closure) by using f_1 to fill in missing values.

3. DISCUSSION AND CONCLUSIONS

This paper provides both theoretical proofs and empirical examples that demonstrate a Focused No Free Lunch result can hold when comparing a subset of algorithms. Depending on the nature of the algorithm and the number of calls to the objective function used by the algorithm, a small set of algorithms will produce identical traces on a set of functions which may be much smaller than the permutation closure over those functions.

In the smallest case, two algorithms can have equal but opposite performance over just two functions; if one function is compressible, both functions can be compressible. Or, one can select the second function from a set of potentially exponentially many function which suffice.

One objection to only looking at m steps is the following. If we construct a focused set based on m time steps, and then run algorithms A_1 and A_2 for $2m$ time steps on these functions, their performance will almost certainly not still be the same after $2m$ time steps. This is true. But either we don't have that access to that information, or we can construct another new focused set using the $2m$ time steps.

This might seem like a flaw. But in practice, we can never have complete traces for any algorithm on any non-trivial (exponentially large) problem. Also consider the following. There are many examples of reasonable search algorithms where the performance of A_1 is better than A_2 after 100,000 evaluations, but A_2 is better than A_1 after 200,000 evaluations. It is not just a matter of one algorithm being better than another, but rather being better than another given some amount of effort such as number of function evaluations. Thus, arguments about the traces an algorithm would have produced if it had exhaustively explored the search space are not practical given that all comparisons of algorithms are based on a limited view of the function based on m samples.

4. ACKNOWLEDGMENTS

This research was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number FA9550-07-1-0403. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. We would also like to thank Dagstuhl for giving us the chance to meet and exchange ideas.

5. REFERENCES

[1] S. Droste, T. Jansen, and I. Wegener. Optimization with randomized search heuristics; the (A)NFL theorem, realistic scenarios and difficult functions. *Theoretical Computer Science*, 2002.

[2] T. English. Practical Implications of New Results in Conservation of Optimizer Performance. In Schoenauer, Deb, Rudolph, Lutton, Merelo, and Schwefel, editors, *Parallel Problem Solving from Nature, 6*, pages 69–78. Springer, 2000.

[3] C. Igel and M. Toussaint. On classes of functions for which No Free Lunch results hold. *Information Processing Letters*, 2003.

[4] N. Radcliffe and P. Surry. Fitness variance of formae and performance predictions. In D. Whitley and M. Vose, editors, *FOGA - 3*, pages 51–72. Morgan Kaufmann, 1995.

[5] C. Schumacher, M. Vose, and D. Whitley. The No Free Lunch and Problem Description Length. In *GECCO-01*, pages 565–570. Morgan Kaufmann, 2001.

[6] M. Streeter. Two broad classes of functions for which a No Free Lunch result does not hold. In J. D. Schaffer, editor, *GECCO 2003*. Springer LNCS, 2003.

[7] D. Whitley. A Free Lunch Proof for Gray versus Binary Encodings. In *GECCO-99*, pages 726–733. Morgan Kaufmann, 1999.

[8] D. Whitley. Functions as Permutations: Regarding No Free Lunch, Walsh Analysis and Summary Statistics. In Schoenauer, Deb, Rudolph, Lutton, Merelo, and Schwefel, editors, *Parallel Problem Solving from Nature, 6*, pages 169–178. Springer, 2000.

[9] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 4:67–82, 1997.