Chapter 10

COMPLEXITY THEORY AND THE NO FREE LUNCH THEOREM

Darrell Whitley
Department of Computer Science
Colorado State University, Fort Collins, CO, USA

Jean Paul Watson
Sandia National Laboratories, Albuquerque, NM, USA

1. INTRODUCTION

This tutorial reviews basic concepts in complexity theory, as well as various No Free Lunch results and how these results relate to computational complexity. The tutorial explains basic concepts in an informal fashion that illuminates key concepts. No Free Lunch theorems for search can be summarized by the following result:

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

Note that No Free Lunch is often referred to simply as NFL within the heuristic search community (despite copyrights and trademarks held by the National Football League).

No Free Lunch relates to complexity theory in as much as complexity theory addresses the time and space costs of algorithms; complexity theory is also concerned with key classes of problems, such as the class of NP-complete problems that are also of interest to researchers designing search algorithms.

2. COMPLEXITY, P AND NP

The complexity classes denoted by P and NP are the most famous (or notorious) classes of problems in complexity theory. The problem class P is the set of problems that can be solved in polynomial time on a deterministic Turing machine. For current purposes, we can think of any computer as a surrogate for a Turing machine (except that Turing machines are assumed to have infinite memory). The P stands for polynomial. In practice, we generally think of P as representing those problems that are *tractable*, i.e. problems that can be solved in reasonable computation time (within one's lifetime, for example).

The problem class NP is the set of problems that can be solved in polynomial time on a nondeterministic Turing machine. The NP stands for nondeterministic polynomial (*not* to be confused with Not Polynomial). Nondeterminism is a little strange. In a nondeterministic machine, choices are allowed in the computation, so that some things need not be computed. In effect, the computation itself becomes a search tree. Each path in the tree represents a possible solution, but only certain paths yield an actual solution. We say that a problem is in NP if this search tree is polynomial in height, while the number of nodes in the search tree might be exponential. Thus, if we could explore all computational paths in parallel, we arrive at a solution in polynomial time. Alternatively, if we "magically" make the right choice at each decision node in the tree, then we again arrive at the desired solution in polynomial time. If we can *deterministically* find a path to a solution in polynomial time in every case, then the problem is in P. All problems in P are also in NP. Another characteristic of the class NP is that the correctness of solutions can be verified in *deterministic* polynomial time. Note that this is true, because if we have the solution in hand, we then know how to make the right choice at each decision node without needing any magical guidance.

Problems in NP that are not known to be in P are characterized by an *algorithm gap*. An algorithm gap exists when the proven difficulty of a problem (or a set of problems) has lower complexity than the best known algorithms for solving that problem. The complexity of the problem itself is algorithm independent and is a bound from below: *the problem can be proven to be at least this hard (but might be harder).* The complexity of the algorithm is a bound from above: *the best known algorithms solves the problem this fast (but might be done faster).*

The complexity of sorting has been proven to be $\mathcal{O}(N \log N)$, thus no algorithm can sort faster than $\mathcal{O}(N \log N)$ in the worst case. Of course, there exist algorithms that sort in $\mathcal{O}(N \log N)$ time, so sorting is said to be a *closed problem* because it does not have an algorithm gap.

If an algorithm sorts faster than $\mathcal{O}(N \log N)$ time, then that algorithm has been designed to work on special subclasses of problems: for example, if we know that we are sorting integers from ranging from 1 to

1000, and the expected distribution of the integers is uniform, we can use a bucket sort and sort in linear, i.e. $\mathcal{O}(N)$, time.

In contrast, an algorithm gap does exist in the well-known traveling salesman problem. Here, the only algorithm guaranteed to locate an optimal solution is, in effect, enumeration. Thus, the best known method in the worst case has complexity $\mathcal{O}(N!)$ for an N city problem. Yet, no one has proven that the *inherent* complexity of the traveling salesman problem is such that it cannot be solved in polynomial time. And note that a solution can be verified in polynomial time. If someone has a solution that is claimed to have a particular evaluation, then that evaluation can be verified in $\mathcal{O}(N)$ time—which is polynomial, of course.

Can all the problems that are solved by a Turing machine in NP time be solved by a deterministic Turing machine using another, more clever algorithm in polynomial time? What we are really asking is whether the complexity class $P = NP$. The answer is unknown and is considered to be one of the most important theoretical questions in Computer Science. It is an equally important question in Operations Research. While the answer is unknown, it is widely thought that $P \neq NP$.

Researchers have identified a very important subset of the class NP known as the class NP-complete. A problem, R, is NP-complete if (1) R is NP-hard and (2) $R \in NP$. Informally, a problem is NP-hard if it is *at least* as hard as any other problem in NP. More formally, a problem R is NP-hard if there exists an NP-complete problem R_0 such that every instance of R_0 can be "reformulated" into an instance of R in deterministic polynomial time. R must be just as hard as R_0 since R in some sense "includes" R_0.

In a renowned theorem, Cook (1971) established that Boolean satisfiability is NP-complete by showing it is in NP and by showing that *every* problem in NP can be expressed as a Boolean satisfiability problem (also just called "SAT"). Of course SAT is a member of the set of NP problems: the nondeterministic Turing machine just selects the right assignment to the Boolean variables to make the expression true, if it is possible to do so.

Other problems in NP have been shown to be NP-complete by showing that every SAT problem can be converted into an instance of that particular problem class. Thus, every instance of SAT can be converted into an instance of the 3-CNF-SAT problem, which can an be converted into an instance of a Hamiltonian circuit problem, which can an be converted into an instance of the traveling salesman problem. This means all of these problems are NP-hard. Showing that they are all also in the class NP makes them NP-complete. Technically, to be NP-complete, a problem must be a decision problem. A decision problem is a problem that has a yes or no answer. Therefore, the traveling salesman problem is "NP-complete" when expressed as a decision problem (i.e., Is there a tour with length $\leq$ X?), but the traveling salesman problem is still said to be "NP-hard" when expressed as an optimization problem.

Given the interrelated nature of the NP-complete problems, if researchers ever discover a polynomial-time algorithm for any NP-complete problem, then it would follow that *every* problem in NP could be solved in polynomial time. In an abstract sense, this means that all problems in the NP-complete problems are all of comparable difficulty, and that the NP-complete are the most difficult problems in the set made up of all problems in NP.

2.1 Complexity, Search and Optimization

Since we do not know how to compute the solution to NP-hard problems in polynomial time, we have to settle for approximate solutions (which sometimes can be computed exactly in polynomial time) or use search methods to find the best solutions possible. It can be useful to think of these search methods as exploring the same decision tree that is navigated by a nondeterministic Turing machine. The solutions that are found using search methods often are not optimal, but finding sufficiently good solutions can be important for many applications.

A basic distinction can be made between search problems that are discrete versus problems that are continuous. This distinction can also be related to the difference between integers and real-valued numbers. If we ask how many integers there are in the (inclusive) interval between 1 and 10, the answer is obviously 10 different and discrete values. But if we asked how many real-valued numbers there are between 1 and 10, the answer is infinitely many.

The nondeterministic Turing machine is clearly solving a discrete problem, because there are a fixed number of decisions that must be made to reach an optimal solution. By definition, the number of decisions that must be made by the nondeterministic Turing machine must be polynomial if it is solving an NP-hard problem.

Some problems cannot be solved in polynomial time by a nondeterministic Turing machines and therefore are not in NP; we can loosely think of such problems as requiring exponential time, although in complexity theory one must worry about both space (memory) and time and balance trade-offs between space and time costs.

Consider a *parameter optimization problem* such that there is a function f that takes k parameters as inputs and returns a single value that evaluates the usefulness or goodness of those k parameters. The space of possible inputs is known as the *domain* and the space of possible outputs as the range or *co-domain* of the function. For example, we might have a parameter optimization problem that used temperature and pressure as two input control parameters for a process that produces some material (e.g. paper), where the output of the function might be the cost of the material, or some measurement of its quality.

If a parameter can be assigned any continuous real-valued number, then the input space is theoretically infinite. We will limit our attention to problems that are discrete such that the domain and therefore the co-domain are finite. Discrete parameter optimization problems are part of a larger set of discrete problems referred to as *combinatorial optimization problems*. Combinatorial optimization problems include many different types of problems, such as scheduling and resource allocation, as well as problems in graph theory and Boolean logic.

For example, we might have a scheduling problem where we want to optimize the order in which tasks are carried out. The goal might be to minimize total processing time, or to maximize work done per unit of time. For N tasks, there could be $N!$ ways to order those tasks. Or, we might want to assign truth values (0 or 1) to a Boolean expression, in which case there are 2^k assignments if there are k Boolean variables in the expression. In the first case, an input could be a permutation of tasks of length N and the evaluation might be how long it takes to process all of the N tasks. In the second case, an input might be a bit-string of length k representing the assignments made to the k Boolean variables, and the output might be a true or false (0 or 1) evaluation of the overall Boolean expression. For classic NP-hard problems, the search space is typically modeled in a general way so that the search space is exponentially large in relationship to the size of an input.

Parameter optimization problems can also be discretized. For example, a single input parameter can be restricted to a value between 0.00 and 99.99 (inclusive) where we only consider values that are increments of 0.01. In this case, there are only 10 000 possible assignments for that particular input. If all of the parameters of a parameter optimization problem are discretized in this way, then the overall search problem is discrete as well. There are a number of reasons that one might want to look at parameter optimization problems as discrete search spaces. In some cases, sensors for the inputs and/or outputs have limited precision and it does not make sense to represent and reason about extremely high precision numbers: we simply cannot measure the world that precisely. And, in general, as soon as anything is represented in a computer program it is discrete. Infinite precision is a fiction, although it is sometimes a useful fiction. But as soon as we decide to represent a parameter using a fixed-length floating point representation, the optimization problem is discrete.

This leads to the following observation. If the set of possible inputs is discrete, we can enumerate the set of inputs and label each possible input with a unique integer. We will also sort the inputs in some principled manner, so that the ith possible input is uniquely identified. This is a familiar concept in complexity, since it allows us to count all of the inputs. Thus, any particular instance of a discrete search problem using any given discrete representation can be abstractly modeled by a function

$$f(i) = j$$

where i is an integer that labels the ith input (i.e. element i of the domain) and j is a member of the set of values that make up the co-domain. This perspective also provides a general foundation for discussing the concept of No Free Lunch.

3. NO FREE LUNCH

In 1995, a paper by David Wolpert and William Macready caused a good deal of excitement in the search community. An updated version of the original report appeared in 1997. The paper *No Free Lunch Theorems for Search* presents proofs that can be summarized by the following No Free Lunch result:

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

First, note that we only consider discrete functions. A performance measure includes any measurement of the quality of the solution (or set of solutions) found after sampling some fixed number of points in the search space, or how long it takes to find a solution of a particular quality. It is also implied that a performance measure is taken over the set of domain and associated co-domain values that have been sampled so far.

A key assumption behind this result is that resampling is ignored: this means that if a search algorithm samples point i and evaluates the objective function $f(i)$ then that point is never sampled again. In reality, heuristic search algorithms “focus” search toward particular regions of the search space: in other words, a focused search is one that spends more time sampling points that are near to one another in the search space. Consequently, a focused search is one that is more likely to resample previously visited points. Search algorithms that are more likely to resample points in the search space than others are in some sense "worse" than algorithms that resample less.

One of the most basic and least intelligent forms of search is random enumeration. Random enumeration means that we sample the search space randomly without replacement; this can be done using clever bookkeeping, or simply by keeping a list of visited points so that none are evaluated again. In practice, random sampling is typically unfocused, only a limited amount of the search space can be sampled, and it is reasonable to allow sampling with replacement because resampling is unlikely. When random sampling is used as a search algorithm, it provides a minimal baseline against which the performance of heuristic search algorithms can be judged. Clearly, we would expect any useful heuristic search algorithm to outperform random enumeration. However, a startling and powerful consequence of No Free Lunch is that *no* heuristic search algorithm is better than random enumeration when compared over all possible discrete functions.

Useful search algorithms do not exhaustively enumerate the entire search space. Wolpert and Macready (1995, 1997) model a search algorithm as a procedure that searches for **m** steps. However, this does not restrict any of the No Free Lunch results.

Another issue relating to No Free Lunch involves deterministic versus stochastic search algorithms. Some algorithms make deterministic decisions, such as a steepest ascent local search algorithm: when started from the same point, steepest ascent always yields the same solution. Genetic algorithms are often implemented as largely stochastic algorithms—meaning that the search involves many random or stochastic decisions and that different runs will often produce different solutions. Wolpert and Macready present arguments showing that the No Free Lunch theorems hold for both stochastic and deterministic search algorithms. Radcliffe and Surry (1995) also point out that in practice stochastic algorithms typically employ pseudo-random number generators. Thus, if we include the random number generator and initial seed in the specification of the search algorithm, then these "stochastic" algorithms, in effect, are also deterministic.

Immediately following its introduction, researchers had two general reactions to the No Free Lunch results.

Reaction 1: Many researchers simply dismissed No Free Lunch, arguing that results concerning the set of all possible discrete functions are not applicable in the real world because this set is not representative of real-world problems. Some researchers pointed out that the set of all possible discrete functions is infinitely large and most functions are *incompressible* in that there is not a representation whose size is significantly less than the size of the function when fully enumerated. For example, if there are N values in the co-domain of a function, then writing down all of these values requires $N \log_2 N$ bits (i.e. N values, $\log_2(N)$ bits per value). In effect, this representation of the function is just a look-up table where the ith entry is the co-domain value associated with $f(i)$. If there exists no representation of a function that uses less than $\mathcal{O}(N \log_2 N)$ bits, then that function is incompressible. Even if an evaluation function only returns 0 or 1, it still requires $\mathcal{O}(N)$ bits to construct a look-up table or to enumerate the function; in this case, the look-up table is still exponentially large when N is exponentially large in relationship to the size of an input string to the evaluation function.

Of course, there are more random functions than non-random functions (English, 2000a). Furthermore, most standard textbooks on computability discuss the well-known result that the set of all possible functions is uncountably infinite (as can be shown using diagonalization arguments), while the set of all possible programs (which are just bit-strings at the lowest level) is only countably infinite (Sudcamp, 1997). So the set of all possible cost functions that can be implemented on a computer is a tiny subset of the set of all possible functions. Thus, the space of all possible discrete functions is largely composed of incompressible functions. Given these observations, “No Free Lunch is No Big Deal” seemed to be the conclusion of this point of view.

Reaction 2: The other reaction to No Free Lunch was to acknowledge that researchers trying to develop the best possible algorithm for a particular application typically need to leverage extensive problem-specific knowledge. Consequently, the No Free Lunch result seemed to be an intuitive affirmation of the idea that there are no general-purpose search methods (at least none that are very effective) and that the business of developing search algorithms is one of building special-purpose methods to solve application-specific problems. This point of view echoes a refrain from the Artificial Intelligence community: “Knowledge is Power”.

Of course, there is truth in both of these views. It has taken several years for the research community to gain a deeper understanding of No Free Lunch. These investigations have led to some surprising and even fruitful results along the way. In 1998 Joe Culberson published an “algorithmic view” of No Free Lunch that added perspective to the debate; Culberson makes two important points.

First, No Free Lunch looks at search as a blind process. This means that the only information we have is the evaluation of particular points in the space. We do not have information about what a solution might look like or information about how the evaluation function is constructed that might allow us to search more intelligently. Blind search is extremely weak. Using an "adversarial argument" we can think of blind search as the process of asking an adversary to sample a point of some objective function and then return an answer. In the space of all possible discrete functions, however, the adversary is free to return any value whatsoever without regard to those values of the search space that have already been examined. In the worst case, sampled points from the search space tell us nothing about the remaining points in the search space.

Second, search is often not blind. If we construct an algorithm for the traveling salesman problem, for example, we often do exploit application-specific operators and representations. But this does not mean that we completely give up generality; our algorithms are designed to solve a particular problem, but should be general enough to solve different instances of that problem.

Radcliffe and Surry (1995) first formalized the idea that we can also include representations under No Free Lunch. That is, when we consider all possible representations of a function, No Free Lunch still holds: no search algorithm is better than another when applied to all possible representations of a function. In effect, a representation just transforms one function into another.

Not surprisingly, No Free Lunch also holds when comparing the set of possible representations under Gray codes and Binary bit encodings. However, Whitley et al. (1997) pointed out that if one selected particular subsets of problems of bounded complexity, then No Free Lunch no longer holds; Rana and Whitley (1997) and Whitley (199) provides proofs of this related to binary representations. Droste et al. (1999) also made similar observations, indicating that one can define sets of reasonable and interesting functions where one algorithm can consistently outperform another.

If we go back in time, No Free Lunch observations were made by Greg Rawlins at the *Foundations of Genetic Algorithms* (FOGA) workshops in 1990 and 1992. In the preface to the proceedings of the 1990 FOGA workshop Rawlins (1991) makes the following observations:

> [I]t is sometimes suggested that GAs [Genetic Algorithms] are universal in that they can be used to optimize any function. These statements are true in only a very limited sense; any algorithm satisfying [these] claims can expect to do no better than random search over the space of all functions. (Rawlins, 1991, p. 7)

> It is now apparent that for a *fixed universal* algorithm, restricted to [bit] strings . . . over the set of all possible domain functions . . . it does not matter which encoding we use, since for every domain function which the encoding makes easier to solve there is another domain function that makes it more difficult to solve. Thus, changing the encoding does not affect the *expected difficulty* of solving randomly chosen domain functions.
>
> Equivalently, assume that we have a *fixed* domain function f and suppose that we choose the encoding, e, at random.. . . Then, no search algorithm can expect to do better than random search, since no information is carried by e about f, except that for each string there is a value (Rawlins, 1991, p. 8.)

Rawlins anticipated several of the consequences of No Free Lunch. Nevertheless, it was Wolpert and Macready who not only provided the first proof of No Free Lunch, but also explored many of the ramifications of the No Free Lunch Theorem.

3.1 No Free Lunch: Variations on a Theme

Two other common variants of NFL are as follows:

- the aggregate behavior of any two search algorithms is equivalent when compared over all possible discrete functions;
- the aggregate behavior of all possible search algorithms is equivalent when compared over any two discrete functions.

At the root of these observations is another, more concise result. Consider any algorithm A_i applied to function f_j. Let *Apply* (A_i, f_j, m) represent a "meta-level" algorithm that outputs the order in which A_i visits m elements in the co-domain of f_j after m steps. For every pair of algorithms A_k and A_i and for any function f_j, there exists another function f_l such that

$$\mathit{Apply}\ (A_i, f_j, m) \equiv \mathit{Apply}\ (A_k, f_l, m)$$

The equivalence operator $\equiv$ denotes that the ordered sequence of co-domain values that is returned by "Apply" will be equivalent. We could interpret this result in 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). In fact, if we consider the algorithms and the functions as variables that are supplied to the Apply function, then when any three of the variables are known, the fourth is immediately determined.

This also implies that we can talk about No Free Lunch in a much smaller context: for example, we can talk about any two search algorithms applied to exactly two carefully chosen paired functions.

This perspective on No Free Lunch has some rather counterintuitive implications, which may be deeper and more profound than the general NFL result. Consider a best-first version of steepest ascent local search which restarts when a local optimum is encountered. Also consider a worst-first steepest ascent local search, also with restarts. We incorporate restarts so that these algorithms continue searching for an arbitrary number of steps. Then, for every function f_j there exists a function f_l such that

$$\textit{Apply}\ (\textit{best-first}, f_j, m) \equiv \textit{Apply}\ (\textit{worst-first}, f_l, m)$$

Virtually all researchers would accept that best-first local search is a reasonable search algorithm and that it is useful on many real-world problems. In other words, there is a subset of problems where best-first search is effective, relative to some performance measure. But there is a corresponding set of functions where worst-first local search is equally effective. What do these functions look like? They probably are "structured" in some sense, and might be compressible. Also note that if we are minimizing a function, then a worst-first local search is one that simply maximizes at each step, instead of minimizing. On the other hand, it seems reasonable that we might want to maximize one function and minimize another function. Why is best-first search generally viewed as a reasonable algorithm and worst-first as an unreasonable algorithm? This is a nagging question for which, at least formally, there are currently no good answers.

3.2 No Free Lunch and Permutation Closure

As has been noted, the set of all possible discrete functions is infinitely large. One easy way to see this is by considering all the functions that take K inputs: since K could be any integer from 1 to infinity, there must be infinitely many discrete functions. But even if there are exactly two inputs, the number of evaluations could be chosen from an infinite set of different possible values, resulting in infinitely many discrete functions.

Whitley et al. (1997) first explored the idea that permutations could be used to represent both algorithms and functions—and thus produce an NFL result over a finite set. This was further explored by Whitley (2000). Consider the following small example. Assume that the co-domain of our objective function consists of the set of values $\{A, B, C\}$. Let the permutation $\langle A, B, C\rangle$ represent a canonical ordering of these values. We can start by considering bijective functions, those that are one-to-one and onto: an important implication of this is that each value in the co-domain is unique. To construct a function, we need to assign values to $f(1), f(2)$ and $f(3)$. Exactly 3! bijective functions can be constructed given three possible co-domain values. Additionally, only 3! *behaviors* are possible for any search algorithm, assuming that an algorithm does not resample points. Let an algorithm's behavior be represented by a permutation over the set of numbers $\{1, 2, 3\}$ which will serve as indices into the canonical permutation of co-domain values $\{A, B, C\}$. Let s_i be the ith value sampled by a search algorithm. Thus, the permutation $\langle 2, 1, 3\rangle$ defined with respect to the canonical ordering $\langle A, B, C\rangle$ represents a search algorithm whose behavior can be described by the following sampling behavior: $s_1 = B, s_2 = A, s_3 = C$. Note that we do not need to specify a particular function to talk about behavior, we just need to define the co-domain values. In the following table, we enumerate all possible permutations over all possible functions over the co-domain $\{A, B, C\}$ as well as all possible permutations over the set of algorithm behaviors over the set of indices denoted by $\{1, 2, 3\}$.

POSSIBLE BEHAVIORS	POSSIBLE FUNCTIONS
B1: < 1, 2, 3 >	F1: < A, B, C >
B2: < 1, 3, 2 >	F2: < A, C, B >
B3: < 2, 1, 3 >	F3: < B, A, C >
B4: < 2, 3, 1 >	F4: < B, C, A >
B5: < 3, 1, 2 >	F5: < C, A, B >
B6: < 3, 2, 1 >	F6: < C, B, A >

The implications of No Free Lunch start to become clear when one asks basic questions about the set of behaviors and the set of functions.

If we apply any two sets of behaviors to all functions, each behavior generates a set of 3! possible search behaviors which is the same as the set of all possible functions. If we apply all possible search behaviors to any two functions, for each function we again obtain a set of behaviors which, after the indices are translated into co-domain values, is the same as the set of all possible functions.

We need to be careful to distinguish between algorithms and their behaviors. There exist many algorithms (perhaps infinitely many) but once the values of the co-domain are fixed, there are only a finite number of behaviors.

Schumacher (2000) and Schumacher et al. (2001) sharpened the No Free Lunch theorem by formally relating it to the *permutation closure* of a set of functions. Let $\mathcal{X}$ and $\mathcal{Y}$ denote finite sets and let f: $\mathcal{X} \longrightarrow \mathcal{Y}$ be a function where $f(x_i) = y_i$. Let σ be a permutation such that

$\sigma : \mathcal{X} \longrightarrow \mathcal{X}$. 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 co-domain (the output values) of the objective function.

We next define the permutation closure $P(F)$ of a set of functions F:

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

Informally, $P(F)$ is constructed by taking each function in F and re-ordering its co-domain values to produce a new function. This process is repeated until no new functions can be generated. This produces *closure* since every re-ordering of the co-domain values of any function in $P(F)$ will produce a function that is already a member of $P(F)$. Therefore, $P(F)$ is closed under permutation. This provides the foundation for the following result.

THEOREM 10.1 *The No Free Lunch theorem holds for a set of functions if and only if that set of functions is closed under permutation.*

Proofs are given by Schumacher et al. (2001). Intuitively, that NFL should hold over a set closed under permutations can be seen from Culberson's adversarial argument: any possible (remaining) value of the co-domain can occur at the next time sample. Proving that the connection between algorithm behavior and permutation closure is an *if and only if* relationship is much stronger than the observation that No Free Lunch holds over the permutation closure of a function. But if every remaining value is not equally likely at each time step, the set of functions we are sampling from is not closed under permutation and No Free Lunch does not hold. Similar observations have also been made by Droste et al. (2002).

It is useful to view the permutation closure of a function as a table, where each row of the table is a permutation representing a function. Each row in the table also corresponds to the behavior of some optimization algorithm on some function. The *behavior* of an optimization algorithm with respect to some objective function describes the order in which the optimization algorithm samples the values that make up the co-domain of the objective function. Schumacher et al. (2001) refer to this as the *performance vector*.

This tabular representation makes it clear when NFL results hold and makes it clear why making a general declaration that one algorithm is better than another is in some sense meaningless.

Consider the following table representing the permutation closure over a function defined over a co-domain of three values.

```
< 1, 2, 3 >
< 1, 3, 2 >
< 2, 1, 3 >
< 2, 3, 1 >
< 3, 1, 2 >
< 3, 2, 1 >
```

Each column of the table represents the set of possible results at a particular time step; the rows represent all possible performance vectors. But each column is identical in its composition. The notion of robustness implies that some algorithm yields relatively good performance over a broad range of problems compared to another algorithm. This would suggest that relatively good solutions are found within some fixed (e.g. polynomial) number of time steps. Yet, if NFL holds over a set of problems, the set of co-domain values returned over all functions in the permutation closure is identical at each time step. Thus, not only are all measures of performance the same after m steps; every step of the search yields exactly the same set of co-domain samples when behavior is aggregated over all possible functions in any permutation closure.

We can now make a more precise statement about the "zero-sum" nature of No Free Lunch. If algorithm **K** outperforms algorithm **Z** on any subset of functions denoted by β, then algorithm **Z** will outperform algorithm **K** over $P(\beta) - \beta$. Differences in aggregate measures of performance such as the total number of steps taken to find a particular evaluation or the sum of the evaluations after m steps will be zero. Aggregate versus average measures of performance can be different, because the subsets are of different size. This means that No Free Lunch theorems for search apply to finite sets. These sets can in fact be quite small.

English (2000a) first pointed out that NFL can hold over sets of functions such as needle-in-a-haystack functions. A needle-in-a-haystack function is one that has the same evaluation for every point in the space except one; in effect, searching a needle-in-a-haystack function is necessarily random since there is no information about how to find the needle until after it has been found.

In the following example, NFL holds over just three functions:

$$\begin{aligned} f &= \langle 0,0,3\rangle \\ P(f) &= \{\langle 0,0,3\rangle, \langle 0,3,0\rangle, \langle 3,0,0\rangle\} \end{aligned}$$

Clearly, NFL does not just hold over sets that are incompressible. All needle-in-a-haystack functions have a compact representation of size $\mathcal{O}(\log N)$, where $N = |\,\mathcal{X}\,|$. In effect, the evaluation function needs to indicate when the needle has been found and return a distinct evaluation.

Generally, we like to construct evaluation functions that are capable of producing a rich and discriminating set of outputs: that is, we like to have evaluation functions that tell us point i is better than point j. But it also seems reasonable to conjecture that if NFL holds over a set that is compressible, then that set has low information measure.

Schumacher et al. (2001) also note that the permutation closure has the following property:

$$P(F \cup F') = P(F) \cup P(F')$$

Given a function f and a function g, where $g \notin P(f)$, we can then construct three permutation closures: $P(f), P(g), P(f \cup g)$. For example, this implies that NFL holds over the following sets which are displayed in table format:

```
Set 1: {< 3, 0, 0 >,
        < 0, 3, 0 >,    Set 3: {< 3, 0, 0 >,
        < 0, 0, 3 >}            < 0, 3, 0 >,
                                < 0, 0, 3 >,
Set 2: {< 1, 3, 2 >,            < 1, 3, 2 >,
        < 2, 1, 3 >,            < 2, 1, 3 >,
        < 2, 3, 1 >,            < 2, 3, 1 >,
        < 3, 1, 2 >,            < 3, 1, 2 >,
        < 3, 2, 1 >}            < 3, 2, 1 >}
```

We can also ask about NFL and the probability of sampling a particular function in $P(f)$. For NFL to hold, we must insist that all members of $P(f)$ for a specific function f are uniformly sampled. Otherwise, some functions are more likely to be sampled than others, and NFL breaks down. For NFL to hold over $P(g)$ the probability of sampling a function in $P(g)$ must also be uniform. But Igel and Toussaint (2004) point out that we can also have a uniform sample over $P(g)$ and a (different) uniform sample over $P(f)$ and NFL still holds. Thus, sampling need not be uniform over $P(f \cup g)$.

3.3 Free Lunch and Compressibility

Whitley (2000) presents the following observation (the current form is expanded to be more precise).

Theorem 10.2 *Let $P(f)$ represent the permutation closure of the function f. If f is a bijection, or if any fixed fraction of the co-domain values* *of f are unique, then $|P(f)| = \mathcal{O}(N!)$ and the functions in $P(f)$ have a description length of $\mathcal{O}(N \log N)$ bits on average, where N is the number of points in the search space.*

The proof, which is sketched here, follows the well known proof demonstrating that the best sorting algorithms have complexity $\mathcal{O}(N \log N)$. We first assume that the function is a bijection and that $|P(f)| = N!$. We would like to "tag" each function in $P(f)$ with a bit string that uniquely identifies that function. We then make each of these tags a leaf in a binary tree. The tag acts as an address that tells us to go left or right at each point in the tree in order to reach a leaf node corresponding to that function. But the tag also uniquely identifies the function. The tree is constructed in a balanced fashion so that the height of the tree corresponds to the number of bits needed to tag each function. Since there are N! leaves in the tree, the height of the tree must be $\mathcal{O}(\log N!) = \mathcal{O}(N \log N)$. Thus $\mathcal{O}(N \log N)$ bits are required to uniquely label each function. (Standard binary labels can be compressed somewhat, but lexicographically ordered bit labels can be used, which cannot be compressed, so that the complexity is still $\mathcal{O}(N \log N)$.)

To construct a lookup table or a full enumeration of any permutation of N elements requires $\mathcal{O}(N \log N)$ bits, since there are N elements and $\log N$ bits are needed to distinguish each element. Thus, most of these functions have exponential description.

This is, of course, one of the major concerns about No Free Lunch theorems. Do No Free Lunch theorems really apply to sets of functions which are of practical interest? Yet this same concern is often overlooked when theoretical researchers wish to make mathematical observations about search. For example, proofs relating the number of expected optima over all possible functions (Rana and Whitley, 1998), or the expected path length to a local optimum over all possible functions (Tovey, 1985) under local search are computed with respect to the set of $N!$ functions.

Igel and Toussaint (2003) formalize the idea 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 are a vanishing small percentage. The problem with this observation is that the *a priori* probability of *any* subset of problems is vanishingly small—including any set of applications we might wish to consider. On the other hand, Droste et al. (2002) have also shown that for any function for which a given algorithm is effective, there exist related functions for which performance of the same algorithm is substantially worse. This is expressed in the *Almost No Free Lunch* (ANFL) theorem.

THEOREM 10.3 *ANFL Theorem: Let H be a randomized search strategy and $f : \{0,1\}^n \rightarrow \{0,1,\ldots,N-1\}$. Then there exists at least $N^{2^{n/3}-1}$ functions $f* : \{0,1\} \rightarrow \{0,1,\ldots,N\}$ which agree with f on all but at most $2^{n/3}$ inputs such that H does find the optimum of f^* within $2^{n/3}$ steps with a probability bounded above by $2^{-n/3}$. Exponentially many of these functions have the additional property that their evaluation time, circuit size representation, and Kolmogorov complexity is only by an additive term of $\mathcal{O}(n)$ larger than the corresponding complexity of f.*

Even search algorithms designed for specific problem classes could be subject to ANFL kinds of effects.

3.4 No Free Lunch and NP-completeness

No Free Lunch has not been proven to hold over the set of problems in the complexity class NP. This is rather obvious if one considers the following: if No Free Lunch holds for any NP-complete problem, then it immediately follows that no algorithm is better than random enumeration on the entire class of NP-complete problems (because of the existence of a polynomial-time transformation between any two NP-complete problems). However, this would also prove that $P \neq NP$, since it would prove that no algorithm could solve all instances of an NP-complete problem in polynomial time. This means that proofs concerning No Free Lunch do not apply to NP-complete problems unless the proofs also show (perhaps implicitly) that $P \neq NP$.

The description length of all NP-complete problems must also be polynomial, since we need to reformulate one problem into another in polynomial time. This means that an NP-complete problem class (such as NK-landscapes: Kauffman, 1989) *cannot* be used to generate all $N!$ functions of $P(f)$ when f is a bijection, since on average the set of all possible bijective functions over a set of co-domain values do not have polynomial space descriptions.

The existence of ratio bounds for certain NP-complete problems also shows that NFL theorems do not hold for certain NP-complete problems. For example, a greedy polynomial time approximate algorithm exists for the Euclidean traveling salesman problem which is guaranteed to yield a solution that is no worse than $2C$, where C is the cost of an optimal solution (Cormen et al., 1990). (In fact, even tighter bounds exist.) Branch and bound algorithms (Horowitz and Sahni, 1978) can use this information to compute bounds such that no solution with a cost greater than $2C$ is examined. Thus, the existence of a ratio bound means that algorithms can select which performance vectors to explore, and this excludes some search behaviors (i.e. performance vectors) that are part of the permutation closure of the objective function.

3.5 Evaluating Search Algorithms

From a theoretical point of view, comparative evaluation of search algorithms is a dangerous, if not dubious, enterprize. But the alternative to testing is to give up and say that all algorithms are equal—which means we have no way of recommending one algorithm over another when a search method is required to solve a problem of practical interest. The best we can do is build test functions that we believe capture some aspects of the problems we actually want to solve. But this highlights a critical question. Do benchmarks really test what we want to test? If an algorithm does well on a very simple problem—such as a linear objective function—is that good or bad? Many people have used the ONEMAX test function for testing search algorithms that use a binary representation. The objective function for ONEMAX is to maximize the number of bits set to **1** in a bit string. But should we really believe that an algorithm that does well on ONEMAX generalizes to other problems of practical interest? Theory would suggest extreme caution.

A set of benchmarks, denoted by β where $S = |\beta|$, i is really just a subset of functions. If algorithm K is better than algorithm Z on β, then algorithm Z is equally and identically better on another set of S functions drawn from $P(\beta)$.

So what does it mean to evaluate an algorithm on a set of benchmarks and compare it to another algorithm? Given the NFL theorems, comparison is meaningless unless we prove (which virtually never happens) or assume (an assumption which is rarely made explicit) that the benchmarks used in a comparison are somehow representative of a particular subclass of problems.

Benchmarks are commonly used for testing both optimization and learning algorithms. Often, the legitimacy of a new algorithm is "established" by demonstrating that it finds better solutions than existing algorithms when evaluated on a particular benchmark or collection of benchmarks. Alternatively, the new algorithm may find high-quality solutions faster than existing algorithms for one or more benchmarks.

What are some of the dangers associated with the use of benchmarks? Algorithms can be tuned such that they perform well on specific benchmarks, but fail to exhibit good performance on benchmarks with different characteristics. More importantly, there is no guarantee that algorithms developed and evaluated using synthetic benchmarks will perform well on more realistic problem instances. Furthermore, simple algorithms can often provide excellent performance on more realistic benchmarks (Watson et al., 1999).

While the dangers associated with benchmarks are well-known, most researchers continue to use benchmarks to evaluate their algorithms. This is because researchers have few alternatives. How can one algorithm be compared to another without some form of evaluation? Evaluation requires the use of either synthetic or real-world benchmarks, or at least the use of test problems drawn from problem generators so that algorithms can be compared on sets of problem instances that have similar characteristics. Researchers who develop new algorithms and do not demonstrate their merit through some form of comparative testing can expect their work to be ignored. The compulsion to develop "a new method" has resulted in the literature being full of new algorithms, most of which are never used or analyzed by anyone other than the researchers who created them.

Hooker (1995) discusses the "evils of competitive testing" and points out the difficulty of making fair comparisons of algorithm performance. Implementation details can significantly impact algorithm performance, as can the values selected for various tuning parameters. Some algorithms have been refined for years. Other algorithms have become so specialized that they only work well on specific benchmarks. Hooker argues that the evaluation of algorithms should be performed in a more scientific, hypothesis-driven manner. Barr et al. (1995) suggest guidelines for the experimental evaluation of heuristic methods. Such guidelines are for the most part useful, although rarely followed.

While evaluation is difficult, it is also important. Too many experimental papers (especially conference papers) include no comparative evaluation; researchers may present a hard problem (perhaps newly minted) and then present an algorithm to solve the problem. The question as to whether some other algorithm could have done just as well (or better!) is ignored.

4. CONCLUSIONS

As in many other areas of life, extreme reactions are likely to lead to extreme errors. This is also true for No Free Lunch. It is clearly wrong to say "NFL doesn't apply to real world problems, so who cares?" It is also an error to give up on building general purpose search algorithms.

A careful consideration of the No Free Lunch theorems forces us to ask what set of problems we want to solve and how to solve them. More than this, it encourages researchers to consider more formally whether the methods they develop for particular classes of problems actually are better than other algorithms. This may involve proofs about performance behavior. In some ways, we are just starting to ask the right questions. And yet, researchers working in complexity and NP-completeness have long been concerned with algorithm performance for particular classes of problems.

Few researchers have attempted to formalize their assumptions about search problems and search algorithm behavior. But if we fail to do this, then we become trapped in a kind of empirical and experimental treadmill that leads nowhere: algorithms are developed that work on benchmarks, or on particular applications, without any evidence that such methods will work on the next problem we might wish to solve.

5. TRICKS OF THE TRADE

No Free Lunch is a theoretical result about search algorithms. As such there are no specific methods or algorithms that directly follow from NFL. Several pieces of advice do follow from No Free Lunch.

1 In most practical applications one must trade-off generality and specificity. Using simpler off-the-shelf search methods reduces time effort and cost. Simple but reasonably effective search methods, even when implemented from scratch, are often easier to work with than complex methods. Using custom-designed search methods that only work for one application will usually yield better results: but generally, one must ask how much time and money one wishes to spend and how good does the solution need to be.

2 Exploit problem-specific information when it is simple to do so. Most NP-complete problems, for example, have been studied for years and there are many problem specific methods that yield good near-optimal solutions.

3 For discrete parameter optimization problems, one has a choice of using standard binary encodings, Gray codes or real-valued representations. Gray codes are often better than binary codes when some kind of neighborhood search is used either explicitly (e.g., local search) or implicitly (e.g., via a random bit flip operator). The use of Gray codes versus real-valued is less clear, and depends on other algorithm design choices.

4 Do not assume that a search method that does well on classic benchmarks will work equally well on real-world problems. Sometimes algorithms are overly tuned to do well on benchmarks and in fact do not work well on real-world applications.

6. CURRENT AND FUTURE RESEARCH DIRECTIONS

Another area of research is the construction of algorithms that can provably beat random enumeration on specific subsets of problems. Christensen and Oppacher (2001) prove that No Free Lunch does not hold over sets of functions that can be described using polynomials of a single variable of bounded complexity. This also includes Fourier series of bounded complexity. (Also see the paper by English (2000a) about polynomials and No Free Lunch). They define a minimization algorithm called "SubMedian-Seeker." The algorithm assumes that the target function, f, is one-dimensional and bijective and that the median value of f is known and denoted by $med(f)$. The actual performance depends on $M(f)$, which measures the number of submedian values of f that have *successors* with supermedian values. They also define M_{crit} as the critical value of $M(f)$ such that when $M(f) < M_{crit}$ SubMedian-Seeker is better than random search. Christensen and Oppacher then prove:

> If f is a uniformly sampled polynomial of degree at most k and if $M_{crit} > k/2$ then SubMedian-Seeker beats random search.

The SubMedian-Seeker is not a practical algorithm. The importance of Christensen and Oppacher's work is that it sets the stage for proving there are algorithms that are generally (if perhaps weakly) effective over a very broad class of interesting, nonrandom functions. More recently Whitley et al. (2004) have generalized these concepts to outline conditions which allow local neighborhood bit climbers to display "SubTheshold-Seeker Behavior" and then show that in practice such algorithms spend most of their time exploring the best points in the search space on common benchmarks and are obviously better than random search.

ADDITIONAL SOURCES OF INFORMATION

The classic textbook *Introduction to Algorithms* by Cormen et al. (1990) has a very good discussion of NP-completeness and approximate algorithms for some well-studied NP-hard problems.

Joe Culberson's 1998 paper *On the Futility of Blind Search: an Algorithmic View of No Free Lunch* helps to relate complexity theory to No Free Lunch in simple and direct terms.

Tom English has contributed several good papers to the NFL discussion (English, 2000a, 2000b). C. Igel and M. Toussaint have also contributed notable papers. Chris Schumacher's 2000 Ph.D. dissertation, *Fundamental Limitations on Search Algorithms*, deals with various issues related to No Free Lunch.

Recent work by Ingo Wegener and colleagues has focused on showing when particular methods work on particular general classes of problems, (e.g., Storch and Wegener, 2003; Fischer and Wegener, 2004) or showing the inherent complexity of particular problems for black-box optimization (Droste et al., 2003).

References

Barr, R., Golden, B., Kelly, J., Resende, M. and Stewart Jr., W., 1995, Designing and reporting on computational experiments with heuristic methods, *J. Heuristics* **1**:9–32.

Christensen, S. and Oppacher, F., 2001, What can we learn from No Free Lunch?, in: *Proc. Genetic and Evolutionary Computation Conference, GECCO-01*, Morgan Kaufmann, San Mateo, CA, pp. 1219–1226.

Cook, S., 1971, The Complexity of Theorem Proving Procedures, in: *Proc. 3rd ACM Symposium on Theory of Computing,* pp. 151–158.

Cormen, T., Leiserson, C. and Rivest, R., 1990, *Introduction to Algorithms,* McGraw-Hill, New York.

Culberson, J., 1998, On the futility of blind search, *Evol. Comput.* **6**:109–127.

Droste, S., Jansen, T. and Wegener, I., 1999, Perhaps not a free lunch, but at least a free appetizer, in: *Genetic and Evolutionary Computation Conference (GECCO-99),* Morgan Kaufmann, San Mateo, CA, pp. 833–839.

Droste, S., Jansen, T. and Wegener, I., 2002, Optimization with randomized search heuristics; the ANFL theorem, realistic scenarios and difficult functions, *Theor. Comput. Sci.* **287**:131–144.

Droste, S., Jansen, T., Tinnefeld, K. and Wegener, I., 2003, A New framework for the valuation of algorithms for black-box optimization, in: *Foundations of Genetic Algorithms (FOGA-7),* Morgan Kaufmann, San Mateo, CA.

English, T., 2000a, Practical implications of new results in conservation of optimizer performance, in: *Parallel Problem Solving from Nature,* Vol. 6, Springer, Berlin, pp. 69–78.

English, T., 2000b, Optimization is easy and learning is hard in the typical function, in: *Proc. Congress on Evolutionary Computation (CEC-2000)*, pp. 924–931.

Fischer, S. and Wegener, I., 2004, The Ising model on the ring: mutation versus recombination, in: *Genetic and Evolutionary Computation Conference, GECCO-04*, Springer, Berlin, pp. 1113–1124.

Rawlins, G., ed., 1991, *Foundations of Genetic Algorithms,* Morgan Kaufmann, San Mateo, CA.

Hooker, J. N., 1995, Testing heuristics: we have it all wrong, *J. Heuristics* **1**:33–42.

Horowitz, E. and Sahni, S., 1978, *Fundamentals of Computer Algorithms,* Computer Science Press, Rockville, MD.

Igel, C. and Toussaint, M., 2003, On classes of functions for which No Free Lunch results hold, *Inform. Process. Lett.***86**:317–321.

Igel, C. and Toussaint, M., 2004, A no-free-lunch theorem for non-uniform distributions of target functions, *J. Math. Model. Algor.*

Kauffman, S. A., Adaptation on Rugged Fitness Landscapes, 1989, in: *Lectures in the Science of Complexity,* D. L. Stein, ed., Addison-Wesley, New York, pp. 527–618.

any update for Igel and Toussaint, 2004?

Radcliffe, N. J. and Surry, P. D., 1995, Fundamental limitations on search algorithms: Evolutionary computing in perspective, in: *Lecture Notes in Computer Science,* Vol. 1000, J. van Leeuwen, ed., Springer, Berlin.

Rana, S. and Whitley, D., 1997, Representations, search and local optima, in: *Proc. 14th National Conference on Artificial Intelligence (AAAI-97)*, MIT Press, Cambridge, MA, pp. 497–502.

Rana, S. and Whitley, D., 1998, Search, representation and counting optima, in: *Proc. IMA Workshop on Evolutionary Algorithms,* L. Davis, K. De Jong, M. Vose and D. Whitley, eds, Springer, Berlin.

Rana and Whitley 1997 not cited.

Schumacher, C., 2000, Fundamental limitations of search, *Ph.D. Thesis,* University of Tennessee, Department of Computer Sciences, Knoxville, TN.

Schumacher, C., Vose, M. and Whitley, D., 2001, The no free lunch and problem description length, in: *Genetic and Evolutionary Computation Conference (GECCO-01),* Morgan Kaufmann, San Mateo, CA, pp. 565–570.

Storch, T. and Wegener, I., 2003, Real Royal Road Functions for Constant Population Size, in: *Genetic and Evolutionary Computation Conference (GECCO-03),* Springer, Berlin, pp. 1406–1417.

Sudcamp, T., 1997, *Languages and Machines,* 2nd edn, Addison-Wesley, New York.

Tovey, C. A., 1985, Hill climbing and multiple local optima, *SIAM J. Algebr. Discr. Methods* **6**:384–393.

Watson, J. P., Barbulescu, L., Whitley, D. and Howe, A., 1999, Algorithm performance and problem structure for flow-shop scheduling, in: *Proc. 16th National Conference on Artificial Intelligence.*

Whitley, D., 1999, A free lunch proof for gray versus binary encodings, in: *Genetic and Evolutionary Computation Conference (GECCO-99),* Morgan Kaufmann, San Mateo, CA, pp. 726–733.

Whitley, D., 2000, Functions as permutations: regarding no free lunch, Walsh analysis and summary statistics, in: *Parallel Problem Solving* *from Nature,* Vol. 6, Schoenauer, Deb, Rudolph, Lutton, Merelo, and Schwefel, eds., Springer, Berlin, pp. 169–178.

Whitley, D., Rana, S. and Heckendorn, R., 1997, Representation issues in neighborhood search and evolutionary algorithms, in: *Genetic Algorithms and Evolution Strategies in in Engineering and Computer Science,* C. Poloni, D. Quagliarella, J. Periaux, and G. Winter, eds, Wiley, New York, pp. 39–57.

Whitley, D., Rowe, J. and Bush, K., 2004, Subthreshold seeking behavior and robust local search, in: *Genetic and Evolutionary Computation Conference (GECCO-04),* Springer, Berlin, pp. 282–293.

Wolpert, D. H. and Macready, W. G., 1995, No free lunch theorems for search, *Technical Report* SFI-TR-95-02-010, Santa Fe Institute, Santa Fe, NM.

Wolpert, D. H. and Macready, W. G., 1997, No free lunch theorems for optimization, *IEEE Trans. Evol. Comput.* **4**:67–82.