Chapter 16
Sharpened and Focused No Free Lunch and Complexity Theory

Darrell Whitley

16.1 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 measures, 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). Two more recent variants of NFL, the Sharpened NFL, and the Focused NFL are also reviewed. There has been a significant amount of confusion in the literature about the meaning of No Free Lunch, and differences between Sharpened NFL and Focused NFL have not been well understood in the literature. This tutorial attempts to resolve some of this confusion. The reader familiar with basic complexity theory might wish to skip to Sect. 16.3 on No Free Lunch. Section 16.4 explains new results based on the distinction between Sharpened and Focused No Free Lunch.

16.2 Complexity: P and NP

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.

D. Whitley (✉)
Department of Computer Science, Colorado State University, Fort Collins, CO, USA

E.K. Burke and G. Kendall (eds.), *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*, DOI 10.1007/978-1-4614-6940-7_16,

The complexity classes denoted by P and NP are the most famous 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. What we generally mean when we say that a problem is not tractable is that the computational costs grow exponentially with problem size and that relatively modest sized problems can result in computation times involving years, or hundreds of years, or trillions of years.

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 (which should *not* be confused with "not polynomial"). Nondeterminism is a bit 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 with paths and decision points in the computation. Each path in the tree corresponds to a computation that results in one possible solution, but only one path or a small number of paths yields an exact and optimal 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, because a nondeterministic Turing machine can do all the computations in polynomial time that can be done by a deterministic Turing machine in polynomial time.

Another hallmark 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.

Can all the problems that are solved by a nondeterministic Turing machine in polynomial time be solved by a deterministic Turing machine in polynomial time using another, more clever algorithm? 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 class, R, is NP-complete if (1) it 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. More formally, it is said that R_0 reduces to R in polynomial time. Therefore R must be just as hard as R_0 since R includes R_0 in some sense.

In a famous 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. A 3-CNF-SAT problem is a satisfiability problem where the Boolean expressions is made up of conjunctive normal form clauses. Each clause contains three Boolean literals, where a literal is a variable or it negation, such as x_1 or $\neg x_1$. All of the clauses of a 3-CNF-SAT problem must be satisfied for the Boolean expression to be satisfied. Every instance of a 3-CNF-SAT, in turn, can be converted into an instance of a Hamiltonian circuit problem, and every Hamiltonian circuit can be converted into an instance of the traveling salesman problem (TSP) (Cormen et al. 1990). 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 TSP is NP-complete when expressed as a decision problem (i.e. is there a tour with length $\leq k$?), but the TSP 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 are all of comparable difficulty, and that the NP-complete problems are (within a polynomial difference) the most difficult problems in the set made up of all problems in NP.

16.2.1 Complexity, Search and Optimization

Since we don't know how to compute the exact solution to NP-hard problems in polynomial time, we sometimes have to settle for approximate solutions. In some cases approximate methods can guarantee a solution which is within some ratio constant of the optimal solution; in other cases this is not possible and we use heuristics search methods to find the best solutions possible. It can be useful to think of these search methods as exploring the decision tree that is magically navigated by a nondeterministic Turing machine. The solutions that are found using heuristic search methods often are not optimal, but finding satisfactory or 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 *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 1 and 100 (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 parameter. 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 can't 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 often an extremely useful fiction (for example, when we can still use mathematical methods that exploit properties of continuous functions). But as soon as we decide to represent a parameter using a fixed-length floating point representation, the optimization problem is actually 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 *i*th 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. the i element 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.

16.3 No Free Lunch

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

> For all possible performance measures, 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.

An updated version of the original report appeared in 1997. 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 concentrate search in particular regions of the search space: in other words, in a concentrated search more time is

spent sampling points that are near to previously observed good solutions. This is sometimes described as intensification or exploitation. Consequently, a concentrated search is one that is more likely to also 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 intelligence 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 explores only a limited amount of the search space, and it is reasonable to allow sampling with replacement because resampling is unlikely if the sampling is random. 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. Other search algorithms are stochastic—meaning that the search utilizes random numbers and makes stochastic decisions and therefore different runs will typically 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 $O(N \log_2(N))$ bits, then that function is incompressible. Even if an evaluation function only returns 0 or 1, it still requires $O(N)$ 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, all of this looks at search as a blind process. This means that we are doing black-box optimization and 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, the previously sampled points from the search space tell us nothing about the remaining points in the search space. (As we will see later in this paper, for Focused No Free Lunch, we are not restricted to blind black-box optimization.)

Second, Culberson points out that search is often not blind. If we construct an algorithm for the TSP, for example, we usually exploit application-specific operators

and representations. But we do not 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 and Rana (1997) pointed out that if one selected particular subsets of problems of bounded complexity, then No Free Lunch no longer holds; Whitley (1999) provided proofs of this for 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 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 a randomly chosen domain function.
>
> Equivalently, assume that we have a *fixed* domain function f and suppose that we choose the encoding, e, at random. That is, we pick one of the ... possible encodings. 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 provided the first detailed proof of No Free Lunch for search.

16.3.1 No Free Lunch: Variations on a Theme

Two 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 that share the same co-domain values.

At the root of these observations is another, more concise result. Consider any algorithm A_i applied to function f_j. Let $\text{Apply}(A_i, f_j, m)$ 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

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

The equivalence operator $\equiv$ denotes that the ordered sequence of co-domain values that is return by "Apply" will be equivalent. We could interpret this result in another way. For every pair of functions f_j and f_l that share the same co-domain values and for any algorithm A_i, there exists another algorithm A_k such that $\text{Apply}(A_i, f_j, m) \equiv \text{Apply}(A_k, f_l, m)$. 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, assuming we restrict the functions to the same set of co-domain values.

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

This perspective on No Free Lunch has some rather counterintuitive implications. 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

$$\text{Apply}(\text{Best-First}, f_j, m) \equiv \text{Apply}(\text{Worst-First}, f_l, m).$$

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. So on some functions, we find a good *minimal* solution by using an algorithm that maximizes. 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 except that we expect functions representing real applications to have a structure that is better explored by Best-First search.

16.3.2 No Free Lunch and Permutation Closure

Whitley et al. (1997, 2000) 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. However, this idea is also implicit in the work of Radcliffe and Surry (1995) on NFL and representations.

Consider the following 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

$$s_1 = f(2) = B, \quad s_2 = f(1) = A, \quad s_3 = f(3) = C.$$

Note that we don't actually 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) make the No Free Lunch theorem more precise by formally relating it to the *permutation closure* of a set of functions. The result is what is now referred to as the Sharpened No Free Lunch theorem. 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:

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

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.

The Sharpened No Free Lunch theorem is often informally expressed by saying that when comparing search methods, the No Free Lunch theorem holds if and only if the set of functions used to compare the algorithms is closed under permutations. However in Sect. 16.4 we will see that we should really be more precise about what the Sharpened No Free Lunch theorem really means.

Sharpened NFL does make it clear that No Free Lunch theorems for search apply to finite sets. These sets can in fact be quite small. 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 that has not yet been sampled can occur at the next time step of search. To see why this is true, assume we have a bijective function, and we place the N values of the co-domain in a grab-bag. By drawing values of the grap-bag we can construct $N!$ different functions. Denote this set $P(F)$ since it is the permutation closure of some seed function. Assume f_j is one of these functions. Next (using our Apply meta-function) execute algorithm A_1 on function f_j:

$$\text{Apply}(A_i, f_j, m).$$

Next, assume we want to compare the behavior of algorithm A_2 against that of A_1. For every $f_j \in P(F)$, there exists a function $f_l \in P(F)$ such that

$$\text{Apply}(A_1, f_j, m) = \text{Apply}(A_2, f_l, m).$$

It also follows that for every $f_l \in P(F)$, there exists a function $f_i \in P(F)$ where this statement also holds. This is the essence of Sharpened No Free Lunch. The statement remains true even if the functions are not bijections. But this only establishes one direction of the *if and only if*.

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, then the set of functions we are sampling from is not closed under permutation and No Free Lunch is not guaranteed to hold for arbitrarily chosen search methods. Similar observations have also been made by Droste et al. (2002). As we will see later, the fact that algorithms A_1 and A_2 can be chosen arbitrarily is critical to Sharpened No Free Lunch.

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 $P(\beta) - \beta$.

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:

$$f = \langle 0,0,3 \rangle$$
$$P(f) = \{\langle 0,0,3 \rangle, \langle 0,3,0 \rangle, \langle 3,0,0 \rangle\}.$$

Clearly, NFL does not just hold over sets that are incompressible. All needle-in-a-haystack functions have a compact representation of size O(lg 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 >,
        < 0, 0, 3 >}

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

```
Set 3: {< 3, 0, 0 >,
        < 0, 3, 0 >,
        < 0, 0, 3 >,
        < 1, 3, 2 >,
        < 2, 1, 3 >,
        < 2, 3, 1 >,
        < 3, 1, 2 >,
        < 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 $P(f \cup g)$.

16.3.3 Free Lunch and Compressibility

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

> Theorem: 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 $|P(f)| = O(N!)$ and the functions in $P(f)$ have a description length of $O(N \lg N)$ 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 $O(N \log N)$. 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 $O(\log N!) = O(N \log N)$. Thus $O(N \log N)$ bits are required to uniquely label each function. Standard binary labels can be compressed by dropping leading zeros, but only 1/2 of the strings can be compressed, so the complexity is still $O(N \log N)$ on average.

To construct a lookup table or a full enumeration of any permutation of N elements requires $O(N \log N)$ 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 which calculate 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 neighborhood 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. This 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.

16.4 Sharpened NFL and Focused NFL

We might express the Sharpened No Free Lunch theorem more precisely as follows:

> The aggregate behaviors of any two arbitrarily chosen search algorithms are guaranteed to be equivalent if and only if the algorithms are compared on a set of functions that are closed under permutation.

It is important to stress that the algorithms are arbitrarily chosen. If we do not know which algorithms are being compared, then the best we can do to ensure that the algorithms display identical behaviors is to compare them on a set of functions that are closed under permutation.

What if we wish to compare two specific search algorithms, A_i and A_k, algorithms that we have knowledge about? Does the "Sharpened No Free Lunch" result still apply?

We will again use the function $\text{Apply}(A_i, f_j, m)$. Recall that this meta-level algorithm outputs the order in which A_i visits m elements in the co-domain of f_j after m steps. We can also reconfigure Apply to be a function generator. We will call the function generator **APPLY** such that

$$f_{\text{out}} = \textbf{APPLY}(A_i, A_k, f_{\text{in}}, m) \iff \text{Apply}(A_i, f_{\text{in}}, m) \equiv \text{Apply}(A_k, f_{\text{out}}, m).$$

We can now define a set that is closed with respect to the operation of the **APPLY** function. Assume that we will be given some as yet unknown algorithms A_i and A_k, and we start with a set F which contains a single function f_1. We assign $f_{\text{in}} = f_1$ and we generate a function $f_{\text{out}} = f_2$.

Define the set $C(F)$ such that the set F is a subset of $C(F)$ and if f_{in} is a member of $C(F)$ then $f_{\text{out}} = \textbf{APPLY}(A_i, A_j, f_{\text{in}})$ is also a member of $C(F)$.

Can we define $C(F)$ in advance so that any two arbitrarily chosen algorithms A_k and A_i are guaranteed to have the same behaviors? The Sharpened No Free Lunch theorem states that $C(F)$ must be a set that is closed under permutation if A_k and A_i are arbitrarily chosen (as yet unknown algorithms) and we require that algorithms A_k and A_i have identical performance when compared on all in the functions in $C(F)$.

But what if we wish to compare exactly two algorithms, A_1 and A_2, and we are told in advance what algorithms are going to be compared? In this case we can potentially find a closure $C(F)$ defined with respect to the **APPLY** function such that the set $C(F)$ need not be closed under permutation in order for algorithms A_1 and A_2 to display the same aggregate performance over the set of function in $C(F)$ for all possible comparative measures.

Using these ideas, Whitley and Rowe (2008) present the key ideas behind the Focused No Free Lunch theorem:

> Let A_1 and A_2 be two predetermined algorithms and let F be a set of functions. The aggregate performance of A_1 and A_2 are equivalent over the set $C(F)$; furthermore, the set $C(F)$ need not be closed under permutation.

Actually, we might also compare 3 or 4 or 20 predetermined algorithms and ask if a set exists where all of the selected algorithms have the same behavior. But for now, looking at just two algorithms is enough to establish the behavior in which we are interested. We will look at two different ways in which Focused No Free Lunch can hold.

First, assume that the two algorithms are deterministic. The search behaviors of A_1 and A_2 when executed on a function f_1 can induce a permutation group such that the orbit of the group is smaller than the permutation closure. For example, assume that A_1 is a local search algorithm that uses a binary bit encoding, and A_2 is a local search algorithm that uses a Gray code bit encoding; assume that both algorithms use the same restart mechanism. Otherwise A_1 and A_2 apply exactly the same search strategy; the only difference is that one uses the binary representation and the other uses a Gray code. Then we can prove that when $F = \{f_1\}$, then the size of $C(F)$ is less than or equal to $2L$ where L is the number of bits used to encode the search space.

Whitley and Rowe (2008) show that this happens because repeated application of Gray encoding induces a group whose orbit is always made up of a set of between L and $2L$ functions. To see why this is true, consider a bit string of length 3. We will take a binary string, then Gray code it, and Gray code it again. Let G^1 denote one application of Gray code, G^2 will denote two applications, and G^x will denote x applications of Gray code:

Binary =	000	001	010	011	100	101	110	111
G^1 =	000	001	011	010	110	111	101	100
G^2 =	000	001	010	011	101	100	111	110
G^2 =	000	001	011	010	111	110	100	101
G^4 =	000	001	010	011	100	101	110	111

By the fourth application of Gray code, the encoding has cycled back to the same as the original binary encoding. Thus the performance of any algorithm using G^4 as a representation is identical to the performance of the same algorithm using the binary coding. The first 4 sets of bit strings form a group with an orbit of 4.

We will generate four functions in the following way: Gray the original function four times, then assume that the bit pattern that is produced is actually the binary encoding of a new function. In other words, each bit string in each representation will be treated as if it is a binary string, b, and the function $f(b) = i$ will return the integer corresponding to the bit string b. (Without loss of generality, we can represent the seven co-domain values as integers.) The binary representation and the Gray code representations are transformed into the following four functions:

	$f_i(1)$	$f_i(2)$	$f_i(2)$	$f_i(3)$	$f_i(4)$	$f_i(5)$	$f_i(6)$	$f_i(7)$
$f_1 =$	0	1	2	3	4	5	6	7
$f_2 =$	0	1	3	2	6	7	5	4
$f_3 =$	0	1	2	3	5	4	7	6
$f_4 =$	0	1	3	2	7	6	4	5
$f_5 =$	0	1	2	3	4	5	6	7

Since $f_1 = f_5$ there are only four distinct functions. One can show by construction that the Gray code representation of f_i induces exactly the same search space as the binary representation of f_{i+1}.

Let A_b be any algorithm that uses a binary coding; let A_g be exactly the same algorithm except that it uses a Gray encoding. On the set of four functions we have defined, algorithms A_b and A_g will have identical performance. This implies

$$\text{Apply}(A_b, f_2, m) = \text{Apply}(A_g, f_1, m)$$

$$\text{Apply}(A_b, f_3, m) = \text{Apply}(A_g, f_2, m)$$

$$\text{Apply}(A_b, f_4, m) = \text{Apply}(A_g, f_3, m)$$

$$\text{Apply}(A_b, f_5, m) = \text{Apply}(A_g, f_4, m).$$

And since $f_5 = f_1$ we have constructed a set $C(F) = \{f_1, f_2, f_3, f_4\}$ that is not closed under permutation. Therefore a Focused No Free Lunch holds. Whitley and Rowe (2008) generalize this result to show that when comparing algorithm A_g and A_b for inputs of L bits, the size of the set $C(F)$ is always less than $2L$. By contrast, the size of the permutation closure $P(F)$ is $2^L!$ for a single seed function. Whitley and Rowe also show that Focused No Free Lunch that exploit the orbits of groups also holds for other classes of search algorithms. In this case Focused No Free Lunch holds even if the entire search space is exhaustively explored.

So, when comparing two specific algorithms we can sometimes look at sets of functions smaller than the permutation closure, and still observe identical performance for the two algorithms we are comparing. Furthermore, while Sharpened No Free Lunch holds for the black-box optimization method, Focused No Free Lunch does not require that the optimization method be a black-box optimizer. For example, for our algorithms A_b and A_g we can have various kinds of information about the

functions we are optimizing. We can know how many parameters there are, where the parameter boundaries are, and we might exploit domain-specific knowledge so that we apply different search strategies for different parameters. Algorithms A_b and A_g can in fact use *any* information we might want to include about the set of functions as long as the only difference between the two algorithms is that one uses a binary encoding and the other uses a Gray encoding. Search need not be blind or black box for Focused No Free Lunch results to hold.

There is a second way in which Focused No Free lunch results can occur. In all real applications the number of points that we sample, denoted by m, is polynomial with respect to input size of the problem, while the search space is exponential. Let N denote the size of the search space. Reconsider the computation using the **APPLY** function where $m << N$:

$$f_{\text{out}} = \textbf{APPLY}(A_i, A_j, f_{\text{in}}, m).$$

There now can be exponentially many functions that can play the role of f_{out} because the behavior of f_{out} is defined at only m points in the search space, and the other points in the search space can be reconfigured in any of $(N-m)!$ ways, all of which are unique if the function f_{in} is a bijection. Intuitively, we no longer need the entire permutation closure to obtain identical over some set of functions when only a tiny fraction of the search space is explored. In fact, under certain conditions one can prove that given two predetermined algorithms A_1 and A_2 there can exist functions f_1 and f_2 such that

$$\text{Apply}(A_1, f_1, m) \equiv \text{Apply}(A_2, f_2, m)$$

$$\text{Apply}(A_1, f_2, m) \equiv \text{Apply}(A_2, f_1, m)$$

so that a Focused No Free Lunch result holds over a set of only two functions such that $C(F) = \{f_1, f_2\}$. This can occur, for example, if the two search algorithm never sample the same domain values or co-domain values on any test function. (It should be noted that this requirement is sufficient, but not strictly necessary; Whitley and Rowe (2008) present a more general result.)

For example, consider the following functions:

$$f_1 = \langle 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7 \rangle.$$

Assume that search algorithm A_1 samples $f_1(4) = 3$ and $f_1(8) = 7$ and halts. Assume that search algorithm A_2 samples $f_1(3) = 2$ and $f_1(6) = 5$ and halts. We can construct a second function f_2 on the fly. We will assume that each algorithm starts at the same domain value. To make their behaviors the same, we require that A_1 samples $f_2(4) = 2$ and that A_2 samples $f_2(3) = 3$. This has the following implications for function f_2:

$$f_2 = \langle ?,\ ?,\ 3,\ 2,\ ?,\ ?,\ ?,\ ? \rangle.$$

In this case, a ? symbol means that the co-domain value of the function at that location has not yet been determined, and the function is under-specified. Next we

see where the algorithms sample next as search continues. Assume that A_1 decides to sample $f_2(1)$ and that A_2 decides to sample $f_2(5)$. Then we continue to define function f_2 so that $f_2(1) = 5$ and $f_2(5) = 7$. We can do this as long as they do not sample the same points. This has the following implications for the construction of function f_2:

$$f_2 = \langle 7,\ ?,\ 3,\ 2,\ 5,\ ?,\ ?,\ ? \rangle.$$

Thus, after sampling only two points in the search space we see that there must exist a function f_2 such that

$$\text{Apply}(A_1, f_1, 2) \equiv \text{Apply}(A_2, f_2, 2)$$

$$\text{Apply}(A_1, f_2, 2) \equiv \text{Apply}(A_2, f_1, 2)$$

and a set $C(F) = \{f_1, f_2\}$ can be defined which is smaller than the permutation closure. It does not matter than function f_2 is under-specified, and that in fact, f_2 actually represents a family of functions, all of which produce the desired behavior.

What happens if we are not so lucky, and algorithms A_1 and A_2 sample some of the same domain values? Whitley and Rowe (2009) present a constructive algorithm that creates a set of functions that yield a set of under-specified functions which in turn creates a closure $C(F)$. We can think of the set F as either allowing under-specified functions, or we can make every function in F specific by randomly filling in the unspecified co-domain values with unused co-domain values.

16.4.1 Partitioning the Permutation Closure under Focused NFL

There can be many ways to partition a set that is closed under permutation to obtain many additional sets that display Focused No Free Lunch results. Let $P(F)$ be the permutation closure of a set of functions denoted by F. Let $C(F)$ be a set of functions such that the specific algorithms A_k and A_j have identical performance on the set of function $C(F)$. In this case, we assume that $C(F)$ contains fully specified functions. Assume that $C(F)$ is a proper subset of $P(F)$. Then a Focused No Free Lunch result holds over $C(F)$ but a Focused No Free Lunch result must also hold over the set $P(F) - C(F)$ as well. Let $\mathcal{F}_1 = C(F)$. If we can extract a second proper subset $\mathcal{F}_2$ from the residual set $P(F) - \mathcal{F}_1$ such that algorithms A_k and A_j have identical performance over the functions in set $\mathcal{F}_2$, then the algorithms will also have identical performance over the residual set $P(F) - \mathcal{F}_1 - \mathcal{F}_2$. By recursively extending this idea, we can decompose the set $P(F)$ into subsets such that A_k and A_j will have identical performance over the functions in each subset $\mathcal{F}_i \subset P(F)$. This also means that $P(F)$ decomposes such that

$$P(F) = \bigcup \mathcal{F}_i.$$

Furthermore, recall that when we are constructing the set $C(F)$ under the conditions that we limit search to m steps, there can be exponentially many different functions that display the same behaviors for the first m steps. Thus, when constructing $C(F)$, there is not a unique way to partition $P(F)$. Assume that we pick a different set of functions, G such $G \in P(F)$ and we define $C(G)$ so that A_k and A_j have identical performance over the functions in $C(G)$. Furthermore, assume that $\forall i, C(G) \neq \mathcal{F}_i$. Let $\mathcal{G}_1$ denote the set $C(G)$. Then we can also define a different set of partitions where

$$P(F) = \bigcup \mathcal{G}_i = \bigcup \mathcal{F}_i.$$

However, the decomposition represented by the sets $\bigcup \mathcal{G}_i$ can be completely different from the decomposition represented by the sets $\bigcup \mathcal{F}_i$.

For example, one can construct cases where even the average size of the subsets that make up $\bigcup \mathcal{G}_i$ is different from the average size of the subsets in $\bigcup \mathcal{F}_i$. One can attempt to construct sets $\mathcal{F}_i$ such that every set is as small as possible. On the other hand, one can allow the search to "wander" through various random functions and allow the subsets that make up $\bigcup \mathcal{G}_i$ to grow larger before attempting to construct a function to create a closure.

Again, consider the following function:

$$f_1 = \langle 0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7 \rangle.$$

Assume that search algorithm A_1 samples $f_1(4) = 3$ and $f_1(8) = 7$ and halts. Assume that search algorithm A_2 samples $f_1(3) = 2$ and $f_1(6) = 5$ and halts. Now, instead of constructing a second function f_2 that produces a closure as quickly as possible, we will add an additional random function that causes the size of the subset to become larger.

We start to construct a different function f_2^* as follows. We still require that A_2 mimic the behavior of A_1 on f_1, thus we assume that A_2 samples $f_2^*(3) = 3$ and then $f_2^*(0) = 7$. This results in the partial construction of f_2^*:

$$f_2^* = \langle 7,\ ?,\ 3,\ ?,\ ?,\ ?,\ ?,\ ? \rangle.$$

This is sufficient to ensure that

$$\text{Apply}(A_1, f_1, 2) \equiv \text{Apply}(A_2, f_2, 2).$$

However, there can be up to $(N-m)!$ different ways of making an underspecified function specific. Suppose we pick the missing values in f_2^* randomly as follows:

$$f_2^* = \langle 7,\ 6,\ 3,\ 5,\ 1,\ 4,\ 0,\ 2 \rangle.$$

And now we execute algorithm A_1 for two steps; assume that it samples $f_2^*(4) = 5$ and $f_2^*(6) = 4$ and halts. We now are forced to construct a third function. But now, we will try to assign values to the underspecified function to create a closure. If possible, we want to create a function f_3 where

$$\text{Apply}(A_1, f_1, 2) \equiv \text{Apply}(A_2, f_2, 2)$$

$$\text{Apply}(A_1, f_2, 2) \equiv \text{Apply}(A_2, f_3, 2)$$

$$\text{Apply}(A_1, f_3, 2) \equiv \text{Apply}(A_2, f_1, 2).$$

With this goal in mind, we create construct f_3 as follows:

$$f_3 = \langle ?,\ 4,\ ?,\ 5,\ 2,\ ?,\ ?,\ ? \rangle.$$

Now assume that on f_3 algorithm A_1 samples $f_3(4) = 5$ and $f_3(2) = 4$ and halts and search algorithm A_2 samples $f_3(5) = 2$ and $f_3(4) = 5$ and halts. This yields the desired result. But it yields a different partitioning of the permutation space, because function f_1 is the same in both cases, but f_1 ends up in partitions of different size. Therefore, there is not a unique way to partition the permutation closure.

There can be many different ways of partitioning the permutation closure. Assume that a f_1 is given and that function f_2 is defined to satisfy the following condition:

$$\text{Apply}(A_1, f_1, m) \equiv \text{Apply}(A_2, f_2, m).$$

If after placing a function f_1 in a partition we can randomly pick any of the exponential many functions from the set of $(N-m)!$ possibilities as the second function, assume we construct them all in parallel, but we attempt to make each closure a different size. This is sufficient to create a large number of ways of partitioning the permutation closure.

16.4.2 Evaluating Search Algorithms

From a theoretical point of view, comparative evaluation of search algorithms is a dangerous, if not dubious, enterprise. But the alternative to testing is to just 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 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.

Each instance of an optimization problem has an associated objective function. Let β represent a particular set of benchmark functions. NFL implies that if algorithm **K** is better than algorithm **Z** on the benchmark set β, then algorithm **Z** must be better than **K** on the instances in $P(\beta) - \beta$. NFL theorems make it clear

that comparative evaluation is really a zero-sum game. But the $P(\beta) - \beta$ might be exponentially large and uncompressible. Focused No Free Lunch indicates that the zero-sum game can be played at a much smaller scale. Focused NFL implies that if algorithm **K** is better than algorithm **Z** on the benchmark set β, then algorithm **Z** must be better than **K** on the instances in $C(\beta) - \beta$. The set $C(\beta)$ where Focused No Free Lunch plays out can be very small indeed. Thus, there is even more reason to suggest extreme caution.

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.

16.5 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.

Unfortunately, it is not widely understood that there are significant differences in the Focused and Sharpened No Free Lunch results. And there are examples in the literature where the Sharpened No Free Lunch result has been overstated to imply that for any two (predetermined) algorithms, the behaviors of those algorithm will be identical if and only if the algorithms are compared over a set of functions closed under permutation. Focused No Free Lunch proves that this interpretation is incorrect as well as very misleading. Correcting this misunderstanding can only help to also clarify our understanding as to what it means to compare algorithms.

16.6 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 the solution needs to be.
2. Exploit problem-specific information when it is simple to do so. For example, most NP-complete problems 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. Don't 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 don't work well on real-world applications.

16.7 Current and Future Research Directions

One body of the literature asks the question "What representation is best?" Of course, the answer is that other No Free Lunch theorems show that in the general case there is no best representation. For discrete parameter optimization problems, one might use standard binary representations, or standard binary-reflect Gray codes. Or one might use real-valued floating point representations.

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 a 2000a paper by English 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 that 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.

Sources of Additional Information

The classic textbook *Introduction to Algorithms* by Cormen et al. 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,b). Igel and Toussaint have also contributed notable papers. Chris Schumacher's 2000 PhD dissertation, *Fundamental Limitations on Search Algorithms*, deals with various issues related to No Free Lunch.

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).

Corne and Knowles (2003) examine questions about No Free Lunch in the space of multi-objective optimization.

Auger and Teytaud (2008) look at the question of whether No Free Lunch applies to continuous functions and conclude that "continuous lunches are free". A paper by Rowe et al. (2009) entitled *Reinterpreting No Free Lunch* presents a general set-theoretic intrepretation of Sharpened No Free Lunch which examines symmetries over arbitrary domains and co-domain values. Whether No Free Lunch holds over continuous parameter optimization problems in practice may depend on what assumptions one makes about these spaces.

References

Auger A, Teytaud O (2008) Continuous lunches are free plus the design of optimal optimization algorithms. Algorithmica 57:121–146

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

Christensen S, Oppacher F (2001) What can we learn from no free lunch? In: GECCO 2001, San Francisco. Morgan Kaufmann, San Mateo, pp 1219–1226
Cook S (1971) The complexity of theorem proving procedures. In: 3rd annual ACM symposium on theory of computing, pp 151–158. ACM, New York
Cormen T, Leiserson C, Rivest R (1990) Introduction to algorithms. McGraw-Hill, New York
Corne D, Knowles J (2003) Real royal road functions for constant population size. In: Evolutionary multi-criterion optimization. LNCS 2632. Springer, Berlin
Culberson J (1998) On the futility of blind search. Evol Comput 6:109–127
Droste S, Jansen T, Wegener I (1999) Perhaps not a free lunch, but at least a free appetizer. In GECCO 1999, Orlando. Morgan Kaufmann, San Mateo, pp 833–839
Droste S, Jansen T, Wegener I (2002) Optimization with randomized search heuristics; the (A)NFL theorem, realistic scenarios and difficult functions. Theor Comput Sci 287:131–144
Droste S, Jansen T, Tinnefeld K, Wegener I (2003) A new framework for the valuation of algorithms for black-box optimization. Foundations of genetic algorithms. Morgan Kaufmann, San Mateo
English T (2000a) Practical implications of new results in conservation of optimizer performance. In: Proceedings of the PPSN 6, Paris. Springer, Berlin, pp 69–78
English T (2000b) Optimization is easy and learning is hard in the typical function. Proceedings of the CEC 2000, pp 924–931
Fischer S, Wegener I (2004) The Ising model on the ring: mutation versus recombination. In: Proceedings of the GECCO 2004, Seattle. Springer, Berlin, pp 1113–1124
Hooker JN (1995) Testing heuristics: we have it all wrong. J Heuristics 1:33–42
Horowitz E, Sahni S (1978) Fundamentals of computer algorithms. Computer Science Press, Washington, DC
Igel C, Toussaint M (2003) On classes of functions for which No Free Lunch results hold. Inf Process Lett 86:317–321
Igel C, Toussaint M (2004) A no-free-lunch theorem for non-uniform distributions of target functions. J Math Model Algorithms 3:313–322
Kauffman SA (1989) Adaptation on rugged fitness landscapes. In: Stein DL (ed) Lectures in the science of complexity, pp 527–618. Addison-Wesley, Reading
Radcliffe NJ, Surry PD (1995) Fundamental limitations on search algorithms: evolutionary computing in perspective. In: van Leeuwen J (ed) Computer science today. LNCS 1000. Springer, Berlin
Rana S, Whitley D (1997) Representations, search and local optima. In: Proceedings of the AAAI 1997, Providence. MIT, Providence, pp 497–502
Rana S, Whitley D (1998) Search, representation and counting optima. In: Davis L, De Jong K, Vose M et al (eds) Proceedings of the IMA workshop on evolutionary algorithms. Springer, Berlin
Rawlins G (ed) (1991) Foundations of genetic algorithms. Morgan Kaufmann, San Mateo
Rowe J, Vose M, Wright A (2009) Reinterpreting no free lunch. Evol Comput J 17:117–129

Schumacher C (2000) Fundamental limitations of search. PhD thesis, University of Tennessee

Schumacher C, Vose M, Whitley D (2001) The no free lunch and problem description length. In: Proceedings of the GECCO 2001, San Francisco. Morgan Kaufmann, San Mateo, pp 565–570

Storch T, Wegener I (2003) Real royal road functions for constant population size. In: Proceedign of the GECCO 2003, Chicago. Springer, Berlin, pp 1406–1417

Sudcamp T (1997) Languages and machines, 2nd edn. Addison-Wesley, Reading

Tovey CA (1985) Hill climbing and multiple local optima. SIAM J Algebr Discret Methods 6:384–393

Watson JP, Barbulescu L, Whitley D, Howe A (1999) Algorithm performance and problem structure for flow-shop scheduling. In: Proceeding of the AAAI 1999, Orlando, pp 688–695

Whitley D (1999) A free lunch proof for Gray versus binary encodings. In: Proceedings of the GECCO 1999, Orlando. Morgan Kaufmann, San Mateo, pp 726–733

Whitley D (2000) Functions as permutations: regarding no free lunch, walsh analysis and summary statistics. In: Schoenauer M et al (eds) Proceedings of the PPSN 6, Paris. LNCS 1917. Springer, Berlin, pp 169–178

Whitley D, Rowe J (2008) Focused no free lunch theorems. In: Proceedings of the GECCO 2008, Atlanta. ACM, New York

Whitley D, Rana S, Heckendorn R (1997) Representation issues in neighborhood search and evolutionary algorithms. In: Poloni C et al (eds) Genetic algorithms and evolution strategies in engineering and computer science. Wiley, New York, pp 39–57

Whitley D, Rowe J, Bush K (2004) Subthreshold seeking behavior and robust local search. In: Proceedings of the GECCO 2004, Seattle. Springer, Berlin, pp 282–293

Wolpert DH, Macready WG (1995) No free lunch theorems for search. Technical report SFI-TR-95-02-010, Santa Fe Institute

Wolpert DH, Macready WG (1997) No free lunch theorems for optimization. IEEE Trans Evol Comput 4:67–82