A Review of No Free Lunch Theorems, and Their Implications for Metaheuristic Optimisation

Thomas Joyce and J. Michael Herrmann

Abstract The No Free Lunch Theorem states that, averaged over all optimisation problems, all non-resampling optimisation algorithms perform equally well. In order to explain the relevance of these theorems for metaheuristic optimisation, we present a detailed discussion on the No Free Lunch Theorem, and various extensions including some which have not appeared in the literature so far. We then show that understanding the No Free Lunch theorems brings us to a position where we can ask about the specific dynamics of an optimisation algorithm, and how those dynamics relate to the properties of optimisation problems.

Keywords No Free Lunch (NFL) · Optimisation · Search · Metaheuristics

1 Introduction

In science, computing, and engineering it is common to encounter situations in which a function can be evaluated on any inputs, and one wants to find the inputs that produce the highest (or equivalently, lowest) output. When the number of possible inputs is too large for it to be feasible to simply try them all, then one must instead rely on some strategy for finding a satisfactory solution. There are many such strategies, and here we look at a subset of these strategies called metaheuristic optimisers. Metaheuristic optimisers can be though of as simple, broadly applicable strategies for finding inputs to functions that result in desirable outputs.

There is a large literature on metaheuristic optimisers, and much work goes in to developing more effective optimisation algorithms and refining existing approaches. However, a fundamental result in optimisation, the No Free Lunch Theorem, shows that, all non-resampling optimisation algorithms perform equally, averaged over all problems. Understanding this result is of central importance for anyone working in

T. Joyce (✉) · J.M. Herrmann
The University of Edinburgh, Edinburgh, Scotland
e-mail: t.joyce@ed.ac.uk

J.M. Herrmann
e-mail: michael.herrmann@ed.ac.uk

X.-S. Yang (ed.), *Nature-Inspired Algorithms and Applied Optimization*,
Studies in Computational Intelligence 744, https://doi.org/10.1007/978-3-319-67669-2_2

optimisation, and in this chapter we give a detailed overview of the No Free Lunch (NFL) Theorems for optimisation, covering both the original theorems, and a number of extensions and refinements. We then examine how the results should influence research into meatheuristic optimisation.

2 Preliminaries

We start by introducing the central definitions and notation used throughout the chapter, most importantly functions and data. Functions are used as formal representations of the problems we want to solve, and data allows us to represent partial knowledge of these functions.

2.1 Functions

In this chapter we consider functions $f : X \to Y$, where X and Y are finite sets with $|X| = n$ and $|Y| = m$. We assume that neither X nor Y are empty. We denote the set of all such functions $\mathscr{F}$, with $|\mathscr{F}| = m^n$.

As we need to represent functions frequently throughout the chapter we introduce a concise representation scheme. Without loss of generality assume that $X = \{1, 2, \ldots, n\}$, then for any $f \in \mathscr{F}$ we can write an ordered list of y values $y_1, y_2, \ldots, y_n$ where $y_i = f(i)$. This ordered list of y values uniquely and fully describes f. When there is no ambiguity (e.g. in the case where $Y = \{0, 1\}$) we will sometimes omit the commas between elements.

Example 1 (Function) Let $X = \{1, 2, 3\}$ and $Y = \{0, 1\}$. We can then write out all $f \in \mathscr{F}$: 000, 001, 010, 011, 100, 101, 110, 111. For example, 110 corresponds to the case where $f(1) = 1, f(2) = 1$ and $f(3) = 0$.

In the optimisation literature there are many terms used to refer to the function being optimised. It is called variously the "cost function", "fitness function", "target function", "objective function", "test function", "problem" and even simply "the function". Here we generally use (problem) function. We now introduce some extensions to the basic function.

Definition 1 (*Bijection*) A function $f : X \to Y$ is a bijection iff $\forall\, x_1, x_2 \in X,\ x_1 \neq x_2 \implies f(x_1) \neq f(x_2)$ and $\forall\, y \in Y,\ \exists\, x \in X$ such that $f(x) = y$.

Definition 2 (*Permutation*) A permutation is a bijection from a set onto itself. That is, $\phi : X \to X$ is a permutation iff ϕ is a bijection.

Definition 3 (*Function Permutation*) Let ϕ be a permutation $\phi : X \to X$, and let $f : X \to Y$ be an arbitrary function. We call f_ϕ a function permutation where we

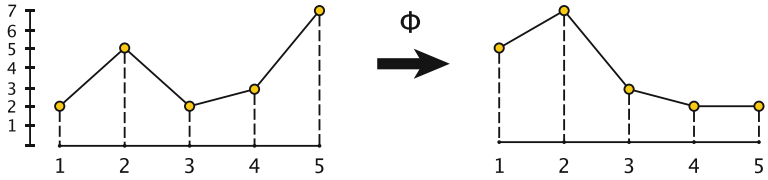

Fig. 1 Function permutation example, $f = 2, 5, 2, 3, 7$ (left) is permuted by $\phi = 2, 5, 4, 1, 3$, resulting in $f_\phi = 5, 7, 3, 2, 2$ (right)

define $f_\phi(x) = f(\phi(x))$. A function permutation can be thought of as re-ordering the output values. The resulting function has the same outputs, but they now correspond to different inputs.

Example 2 (Function Permutation) Let $f : \{1, 2, 3, 4, 5\} \rightarrow \{1, 2, 3, 4, 5, 6, 7\}$ with $f = 2, 5, 2, 3, 7$ and let ϕ be a permutation $\phi = 2, 5, 4, 1, 3$, then $f_\phi = 5, 7, 3, 2, 2$. See Fig. 1 for a graphical representation of this example.

2.2 Data

When optimising a function $f : X \rightarrow Y$ we assume we know X and Y, but only know some (or none) of the values $y = f(x)$. We represent such partial knowledge as a function $d : X \rightarrow Y \cup \{?\}$ which we call data. The question mark "?" represents an x value for which the $f(x)$ value is unknown. For all x values where we know $y = f(x)$ let $d(x) = f(x)$, for all other values of x we let $d(x) = ?$. In other words, if for some $x \in X$ we have $d(x) = ?$ then we currently do not know $f(x)$. Let $\mathscr{D}$ be the set of all possible data for functions in $\mathscr{F}$. It follows that $|\mathscr{D}| = (m + 1)^n$. We refer to the data where $\forall\, x \in X,\ d(x) = ?$ as "no data", as this represents the situation in which we know nothing about the problem function.

Example 3 (Data) Let $X = \{1, 2, 3, 4\}$ and $Y = \{0, 1\}$. Assume we have partial knowledge of some $f \in \mathscr{F}$. In particular, we know that $f(2) = 1$ and $f(4) = 1$. We can represent this data as a function d where $d(1) = ?$, $d(2) = 1$, $d(3) = ?$ and $d(4) = 1$. As discussed in Sect. 2.1, this can be represented more succinctly as: ?1?1. Note that, assuming the data is accurate, there are four possible functions consistent with the data: 0101, 0111, 1101 and 1111.

3 Optimisation Algorithms

We have introduced functions, and data, to represent the problems we want to solve, and partial knowledge of these problems, respectively. We now introduce a represen-

tation of optimisation algorithms themselves. We first introduce a *sampling policy*, which is the decision making component of the optimisation algorithm.

3.1 Sampling Policy

The situation discussed throughout this chapter is the following: there is a function $f : X \to Y$ which can be evaluate at any $x \in X$, and the current knowledge of the f is represented by data d. A sampling policy is a rule for deciding where to sample next, based on the current data. In other words, given the current data it specifies which of the potentially numerous x we should evaluate f at next.

Definition 4 (*Sampling Policy*) A sampling policy s is a function $s : \mathscr{D} \to X$, where $\mathscr{D}$ is the set of all possible data, as defined in Sect. 2.2.

Example 4 (Sampling Policy) Let $X = \{1, 2\}$ and $Y = \{0, 1\}$. An example sampling policy s is: $s(??) = 1$, $s(0?) = 2$, $s(1?) = 1$, $s(?0) = 1$, $s(?1) = 1$. We do not need to specify $s(00)$, $s(01)$, $s(10)$, or $s(11)$, as we are only concerned with the behaviour of the sampling policy when some points remain unsampled.

Definition 5 (*Non-Repeating Sampling Policy*) A non-repeating sampling policy s is a function $s : \mathscr{D} \to X$ s.t. if $s(d) = x$ then either $d(x) = ?$ or $|d| = m$.

Example 5 (Non-Repeating Sampling Policy) Let $X = \{1, 2\}$ and $Y = \{0, 1\}$. Then the set of possible data $\mathscr{D} = \{??, ?0, ?1, 0?, 1?, 10, 01, 00, 11\}$ and $s(??) = 1$, $s(?0) = 1$, $s(?1) = 1$, $s(0?) = 2$, $s(1?) = 2$, is a non-repeating sampling policy.

Here, we will consider deterministic non-repeating sampling policies, unless otherwise stated, and when there is no ambiguity we will just call them sampling policies.

3.2 Optimisation Algorithms

So far we have defined sampling policies, which decide where to sample given data. However, we can add to our data by evaluating the problem function f at some x and updating d by setting $d(x) = f(x)$. Essentially, an optimisation algorithm is the repeated use of a sampling policy, adding to our data each time we make a sample, until a termination condition is reached. For the time being we fix our termination condition to: terminate iff we have sampled f at every $x \in X$. Given this fixed termination condition an optimisation algorithm is fully specified by a choice of sampling policy.

Definition 6 (*Optimisation Algorithm*) An optimisation algorithm A based on sampling policy s is iterated use of that sampling policy and data updating:

1. Set d to be the initial data (generally "no data", but see Sect. 3.3).
2. If there are no unsampled $x \in X$, then terminate.
3. Sample at $x = s(d)$ and add the result to the data d, i.e. set $d(x) = f(x)$.
4. Go to step 2.

3.3 On-Policy and Off-Policy Behaviour

One useful observation on the behaviour of deterministic optimisation algorithms is the existence of what can be thought of as on-policy and off-policy behaviour. This distinction is important for later proofs, and so we make it clear here.

The on-policy behaviour of an algorithm is the the behaviour on inputs potentially seen when the algorithm is run until termination *starting with no information about* f. The on-policy behaviour is generally not the full behaviour, as there are data inputs that will never be seen in the normal execution of the algorithm, regardless off the function f being sampled. This is clarified in the following example:

Example 6 (On-Policy Behaviour) Consider the sampling policy described in Example 5. $s(??) = 1$, and so an optimisation algorithm using this sampling policy will initially sample $f(1)$, and thus the data ?0 and ?1 will never be seen, as those data are the possible results from evaluating $f(2)$ first. It is not that $f(2)$ will never be evaluated, rather that if we follow the policy, $f(2)$ will never be evaluated *first*.

Off-policy behaviour is the sampling policy restricted to exactly those data inputs that do not appear in the on-policy behaviour. Thus together the off-policy and on-policy behaviours describe the full behaviour. Broadly speaking, the on-policy behaviour is all the behaviour that is possible in the normal running of the optimiser, and off-policy behaviour is that that results from starting the optimisation algorithm with some initial (off-policy) knowledge of the problem function. In this chapter we restrict attention to on-policy behaviour unless explicitly stated.

3.4 Representing Optimisation Algorithm Behaviour

When discussing optimisation algorithms it is often helpful to consider their behaviour represented as a behaviour graph, which we now define:

Definition 7 (*Behaviour Graph*) An optimisation algorithm's behaviour can be represented as a directed graph, in which nodes represent data and the edges show all potential transitions between data resulting from sampling the problem function in accordance with the optimiser's sampling policy. We call such a graph the behaviour graph of the optimiser. See Fig. 2 for an example of a behaviour graph.

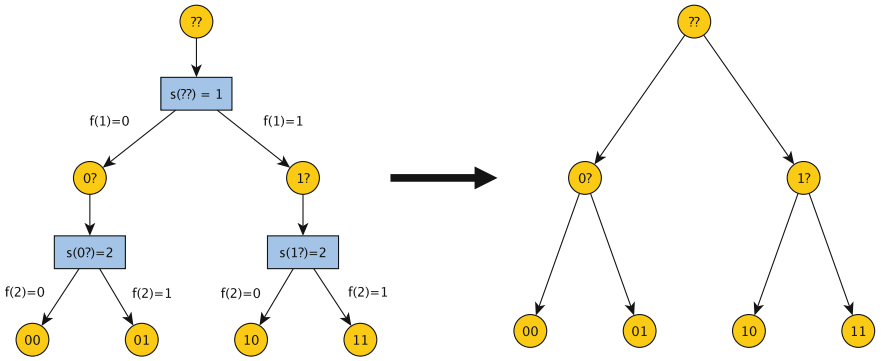

Fig. 2 Left: The on-policy behaviour of the optimisation algorithm using the sampling policy from Example 5. The yellow circular nodes represent data, the blue rectangular nodes show the sampling policy's decision based on data. Right: The same tree shown more compactly by removing the explanatory details. The tree still contains the same information and is the format we will generally use, for compactness

We now show that if we restrict attention to on-policy behaviour of deterministic, non-resampling optimisation algorithms, then the behaviour graph is in fact a tree. This is a key observation and is used variously throughout the rest of the chapter.

Theorem 1 (Optimisation Algorithm Tree Representation) *A deterministic non-resampling optimisation algorithm's on-policy behaviour can be uniquely represented as a directed graph. In fact this directed graph is a balanced tree.*

Proof From the definition of on-policy behaviour we start the optimisation with no data about the function, which we represent as the root node of the tree. The algorithm's sampling policy determines which x should be sampled given no data, say x_1. When the algorithm performs the sample at x_1 there are $|Y| = m$ possible results, and each of these results necessarily leads to different data (as the data is just a description of the result). Thus, we can potentially transition to any of m new data nodes by appending the result to the data, setting $d(x_1) = f(x_1)$.

At this point, either the node we are at represents data with no unknown values, in which case the algorithm halts and we are at a leaf, or the node contains at least one value for which $d(x) = ?$. In this second case the sampling policy will select one of these x to be sampled (as it is non-resampling), and again m possible results exist, we follow one of the possible edges (dependent on the results of the sample) and then repeat the process until we do eventually reach a leaf.

We now show that all paths do eventually lead to a leaf. Whenever an x is sampled an unknown $f(x)$ becomes known. As the algorithm terminates exactly when all $f(x)$ are known, and there are only $n = |X|$ unknown $f(x)$ at the start, and as, because the sampling policy is non-resampling, the algorithm only ever samples x for which $f(x)$ is unknown, then the algorithm necessarily terminates after exactly n samples, and we reach a leaf.

We now show that no two paths arrive at the same node. First we note that all paths through the tree start at the root node. Assume we have two paths that at some point separate. If they were to rejoin they must both eventually arrive at the same data set. However, the very fact that they separated means that for some x they produced different $f(x)$ values. Thus, their data can never be the same, and they will never rejoin at a node. It follows that the graph is a tree.

Finally, as every path necessarily terminates after exactly n steps, and every non-leaf node leads to exactly the same number m of immediate children, it follows that the tree is balanced.

Theorem 2 (Tree Representation Details) *The tree representing the on-policy behaviour of an optimiser for functions mapping $X \to Y$ where $|X| = n$ and $|Y| = m$ has $n + 1$ layers. If we label these layers $0, 1, \ldots, n$ starting from the root then layer i contains m^i nodes, and the tree contains $1 + m + m^2 + \cdots + m^n$ nodes in total. The final layer consists of m^n leaves with exactly one representing each of the m^n functions $f \in \mathscr{F} = Y^X$.*

Proof Each node represents particular data $d \in \mathscr{D}$. The root node is always d s.t. $\forall\, x \in X,\ d(x) = ?$. Call this root node layer 0. Each node on the i-th layer, for $i \in \{0, \ldots, n\}$, will represent data with exactly $n - i$ unsampled x values. Thus, the n-th layer will consist of leaves, and will be the final layer.

Every non-leaf node leads to exactly $m = |Y|$ immediate children. We have also seen in the proof of Theorem 1 that no node is the child of more that one node. Thus, as in layer zero there is 1 node in layer 1 there will be m, in layer 2 there will be m^2 nodes, and in layer i there will be m^i nodes for i ranging from 0 to n.

On the n-th layer there are m^n leaves, each containing data describing a different $f \in \mathscr{F}$. As $|\mathscr{F}| = m^n$ every $f \in \mathscr{F}$ must correspond to a leaf.

In Fig. 3 we show example on-policy trees for three different domains. It can be seen that even for $|X| = 4$ and $|Y| = 2$ the tree becomes fairly large. Although trees are possible for any finite X and Y, we will generally not be able to show them explicitly.

3.5 Paths down Trees

We have seen that the on-policy behaviour of an optimisation algorithm can be represented as a balanced tree (Theorem 1). We now show that running an optimiser on a particular function corresponds to taking a particular path down the optimiser's tree. Figure 4 provides a graphical example of these paths down trees.

Theorem 3 (Paths Down Trees) *When the on-policy behaviour of an optimisation algorithm is represented as a rooted tree, then paths down that tree biject with the functions $f \in \mathscr{F}$, and the path shows the choices that the algorithm will make when optimising the corresponding f.*

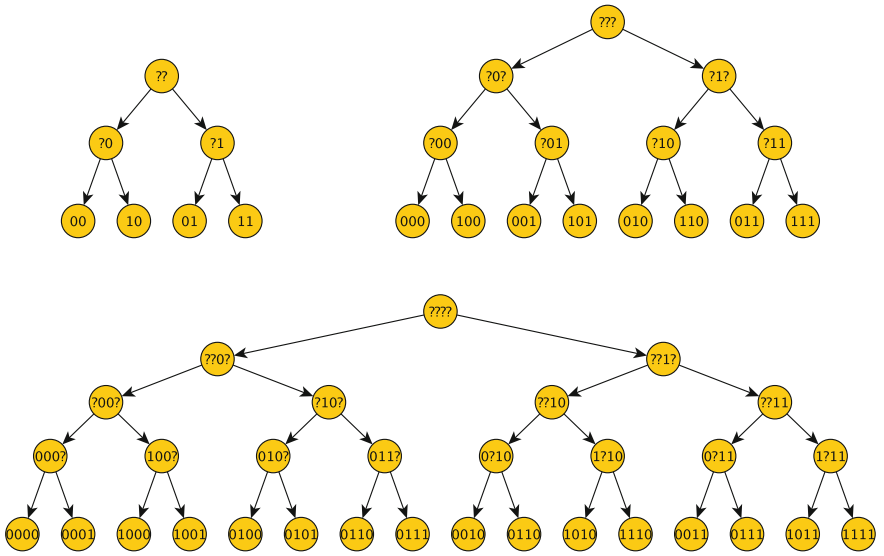

Fig. 3 The example policies, represented as trees. Top left: A policy for functions from $X = \{1, 2\}$ to $Y = \{0, 1\}$. Top right: A policy for functions from $X = \{1, 2, 3\}$ to $Y = \{0, 1\}$. Bottom: A policy for functions from $X = \{1, 2, 3, 4\}$ to $Y = \{0, 1\}$

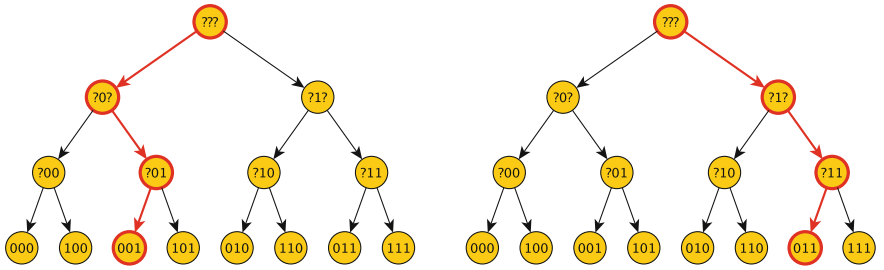

Fig. 4 Two example paths taken by the optimisation algorithm down its behaviour tree. The left path shows the route when the algorithm is run on $f = 001$ and the right path shows the behaviour on $f = 011$

Proof Recall that every leaf is the result of optimising one $f \in \mathscr{F}$. The existence of the bijection follows directly from the fact that the graph structure is a tree, and there is thus only a single path from the route to each leaf.

4 No Free Lunch

The original No Free Lunch (NFL) theorem for optimisation [1, 2] states that no optimisation algorithm can outperform any other under any metric over all problems. The theorem first appears in Wolpert and Macready's 1995 paper "No Free Lunch Theorems for Search" [1], but became more widely known through the same authors 1997 paper, "No Free Lunch Theorems for Optimization" [2]. There are many formulations of the result in the literature, with different emphasis. Here we present a representative selection, starting with Wolpert and Macready's own characterisation:

1. The average performance of any pair of algorithms across all possible problems is identical [2].
2. For all possible metrics, no search algorithm is better than another when its performance is averaged over all possible discrete functions [3].
3. On average, no algorithm is better than random enumeration in locating the global optimum [4].
4. The histogram of values seen, and thus any measure of performance based on it is independent of the algorithm if all functions are considered equally likely [5].
5. No algorithm performs better than any other when their performance is averaged over all possible problems of a particular type [6].
6. With no prior knowledge about the function $f : X \to Y$, in a situation where any functional form is uniformly admissible, the information provided by the value of the function in some points in the domain will not say anything about the value of the function in other regions of its domain [7].

As this selection shows, NFL allows a broad range of assertions. We will now formally state and prove the NFL theorem, after two preliminary definitions.

Definition 8 (*Traces*) The trace of an optimisation algorithm A running on a function f is the ordered list of (x, y) pairs sampled (where $y = f(x)$). We write $T_A(f)$ for the trace of algorithm A running on function f. We also define $T_A^{(k)}(f)$ to be the trace after k function evaluations. As we restrict attention to non-resampling optimiser, it follows that if n is the size of the domain of f then $T_A^{(n)}(f) = T_A(f)$. We call $T_A(f)$ the (full) trace and $T_A^{(k)}(f)$ a partial trace for any $k < n$. Let $\mathscr{T}_A$ be the set of all the traces that algorithm A produces on functions mapping X (where $|X| = n$) to Y, that is:

$$\mathscr{T}_A = \cup_{f \in \mathscr{F}} \cup_{k=0}^{n} T_A^{(m)}(f)$$

We also define a trace of just the inputs $T_A(f)_X$ and a trace of just the outputs $T_A(f)_Y$ as ordered lists of just the x and y values respectively from the full trace. We call these the input trace and the output trace. The output trace is sometimes called the performance vector or range trace in the literature. Similarly, we define $\mathscr{T}_{A,X}$ and $\mathscr{T}_{A,Y}$ all possible input traces and all possible output traces, respectively.

Example 7 (Traces) Let $X = \{1, 2, 3\}$ and $Y = \{0, 1\}$ and let A be an optimisation algorithm using sampling policy s, with $s(???) = 2$, $s(?0?) = 3$ and $s(?1?) = 3$

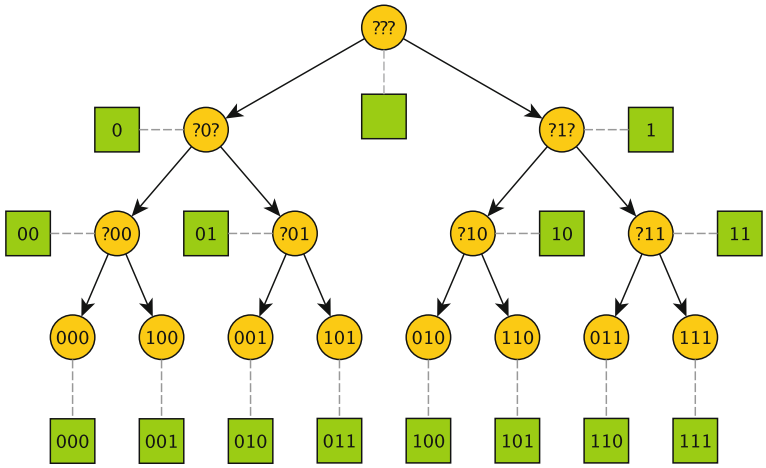

Fig. 5 The yellow nodes show the behaviour tree for an optimisation algorithm (using the sampling policy defined in Example 7) on functions between $\{1,2,3\}$ and $\{0,1\}$. The green squares attached to each yellow node show the corresponding output trace at that point. If we let A be the algorithm represented by the tree, then it can see that, for example, $T_{A(010),Y} = 100$ by reading the trace at the 010 leaf

(A is shown graphically in Fig. 5). Let $f : X \to Y$ with $f = 010$. Then $T_A(f) = \{(2,1),(3,0),(1,0)\}$, $T_A^{(2)}(f) = \{(2,1),(3,0)\}$, $T_A(f)_X = \{2,3,1\}$ and $T_A(f)_Y = \{1,0,0\}$, $\mathscr{T}_{A,Y} = \{\{0,0,0\},\{0,0,1\},\{0,1,0\},\{0,1,1\},\{1,0,0\},\{1,0,1\},\{1,1,0\},\{1,1,1\},\{0,0\},\{0,1\},\{1,0\},\{1,1\},\{0\},\{1\},\{\}\}$.

As noted in [8] optimisation algorithms have no "intrinsic purpose" and their behaviour needs to be qualified by some external metric. A metric provides a way to evaluate an algorithm's performance. We now make this idea of a metric precise.

Definition 9 (*Optimisation Metric*) An optimisation metric M is a function that maps output traces to $\mathbb{R}$. A metric can be thought of as assigning a value, or score, to an output trace, $M : \mathscr{T}_Y \to \mathbb{R}$.

An alternative but equivalent way to think of a metric is as a function M^* from an algorithm, a function, and a number of samples $t \in \mathbb{N}$, to a rating of how well the algorithm performs on that function after that many samples, that is, $M^* : \mathscr{A} \times \mathscr{F} \times \mathbb{N} \to \mathbb{R}$ where $M^*(A,f,t) = M(T^{(t)}_{A(f),Y})$.

Example 8 (Optimisation Metrics) A simple metric for measuring minimisation performance is a function that returns the minimum Y value in the trace (i.e. the smallest y value sampled so far). In fact, although simple, this metric is often used to compare optimisers, especially in cases where the true global minimum of the function is unknown.

We are now in a position to formally state and prove NFL. We follow the style of proof used by English [8–10], as this is in our opinion clearest, and naturally yields

various generalisations to the theorem. It involves explicit reference to the representation of optimisers as trees. Another particularly clear approach to the proof is using the "fundamental matrix" method from [11]. We have chosen a tree based method because, in our opinion, it makes the sequential decision aspect of the algorithm producing a trace (see Definition 8) more explicit. However, we strongly recommend [11] to any readers wanting an alternative approach.

Theorem 4 (NFL) *All optimisation algorithms produce the same set of traces when run over all possible functions between two finite sets. More formally, let* $Y^X = \{f \mid f : X \to Y\}$*, where X and Y are arbitrary finite sets. For all optimisation algorithms* A, B*,* $\{T_{A(f),Y} \mid f \in Y^X\} = \{T_{B(f),Y} \mid f \in Y^X\}$.

Proof As we have shown in Sect. 3.4, the behaviour of an optimisation algorithm is uniquely representable as a tree. This tree has $|Y|^{|X|}$ leaves and the trace at each leaf is unique. However, by a counting argument there are only $|Y|^{|X|}$ possible distinct traces of length $|X| = n$. Therefore, every optimisation algorithm produces the same set of traces, namely every possible trace exactly once.

It is worth noting that the proof above is very succinct, this is in part due to the reference to the proof that the behaviour of an optimisation algorithm is uniquely representable as a tree, but also partly a result of the decision tree proof style.

5 Basic No Free Lunch Extensions

We now present some additional results and observations that are of interest in themselves, and will also be used in refinements of NFL below. Theorem 4 showed that every algorithm is equivalent when full traces are considered. We first generalise to the case where we stop the optimisation after k steps, then we generalise further to the case of arbitrary stopping conditions. After this we give a proof that the NFL result still holds if we allow stochastic sampling policies.

5.1 k-Step No Free Lunch

In real applications, optimisation algorithms are not usually run until the entire domain has been sampled. In the original no free lunch papers by Wolpert et al. [1, 2] they show that the no free lunch theorem still applies if algorithms are run for some fixed number of steps. We restate this theorem here and prove it using tree representations. We first define a multi-set, which is used in the theorem.

Definition 10 (*Multi-set*) A multi set is a set in which elements can occur multiple times. Another way to think of a multi-set is as a set in which each element has an associated count.

Theorem 5 (k-step NFL) *All optimisation algorithms produce the same set of traces when run over all possible functions between two finite sets for k steps. More formally, let* $Y^X = \{f \mid f : X \to Y\}$, *where X and Y are arbitrary finite sets. For any optimisation algorithms A, B,* $\{T_A^{(k)}(f)_Y \mid f \in Y^X\} = \{T_B^{(k)}(f)_Y \mid f \in Y^X\}$. *In fact they produce the same multi-set, in that every possible trace appears the same number of times, with the exact value depending on k and* $|X|$.

Proof If we prune the full behaviour tree after k steps the resulting tree will have $|Y|^k$ leaves. Each leaf has corresponds to a unique output trace and these traces are of length k. Thus, as there are only $|Y|^k$ possible output partial traces of length k, each partial trace must be present exactly once in the leaves. This is the case for all optimisation algorithms.

Because we have pruned the tree after k steps, each leaf in the pruned tree (and thus each partial output trace) will result when optimising multiple problem functions. However, as the full behaviour tree was a balanced tree, it follows that each partial trace will be obtain the same number of times when all possible functions are considered.

We defined the behaviour tree for an optimiser in Sect. 3.4. We now define a similar but distinct tree representation, the trace tree. Essentially the behaviour tree show both the x and y values of the algorithms samples, the trace tree in contrast only shows the y values.

Definition 11 (*Trace Tree*) Let A be an optimisation algorithm for functions $f : X \to Y$. The trace tree for A is the behaviour tree with the nodes labelled with the trace, rather than the data.

An example trace tree is given in Fig. 6. It is the trace tree of the optimisation algorithm in Example 5, the behaviour tree of which is shown in Fig. 5. An example of a trace tree for a restricted number of steps is shown in Fig. 7.

We now show that trace tree only depends on the range and the size of the domain of the functions being optimised. The detail that we lose when we switch from behaviour trees to trace trees is exactly the detailed that differentiated the optimisers.

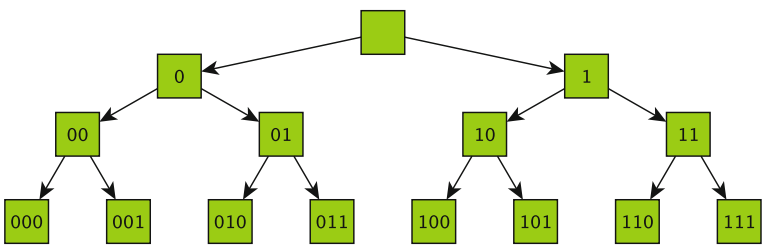

Fig. 6 The behaviour tree showing only the trace values. When considering just the traces, all algorithms have the same behaviour tree (see Theorem 6). What differentiates algorithms is that for different algorithms a given function will produce different *paths* in this tree

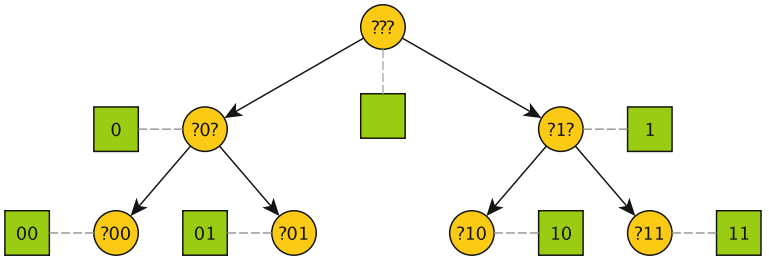

Fig. 7 The tree in Fig. 6 after $k = 2$ steps. As $|Y| = |\{0, 1\}| = 2$ there are $|Y|^k = 2^2 = 4$ leaves, each having one of the four possible two bit traces

Theorem 6 (Identical Trace Trees) *All optimisation algorithms for functions mapping X to Y produce the same trace tree. In fact, all optimisation algorithms for functions mapping Z to Y also produce the same trace tree, as long as* $|Z| = |X|$.

Proof Consider an optimisation algorithm A for functions $f : X \to Y$. We will show that A's trace tree does not depend on any of the specific details of A, and thus would be the same for any optimisation algorithm for functions $f : X \to Y$.

From the definition of a trace tree we know that the algorithm's trace tree has the same structure as the algorithm's behaviour tree. In particular, as we are only considering non-resampling optimisers we know that the trace tree will have $|X| + 1$ layers, and that each node in the tree will be either a leaf or the parent of exactly $|Y|$ other nodes.

Now we simply observe that if we consider a non-leaf node in the trace tree then we can detail the node's children without knowledge of A. Suppose the node we are considering has trace $y_1, y_2, \ldots, y_k$, then its $|Y|$ children will have the traces $y_1, y_2, \ldots, y_k, y$ for each $y \in Y$.

Next we note that the only influence the domain X has on the trace tree is in setting the number of layers to be $|X| + 1$, thus it is only the size of the domain that is important and the trace tree will be the same for any domain of that size.

Corollary 1 *The leaves of the trace tree are all possible problem functions.*

This was shown in Theorem 2 but is restated as a corollary above, as it is important in the proofs to follow.

Continuing our generalisations of NFL to early stopping scenarios, we now consider the more general situation in which the optimisation process can terminate based on the results so far, rather than simply after a fixed number of steps. As we will see, a no free lunch result still pertains.

5.2 Stopping Condition No Free Lunch

A stopping condition is a rule for when to stop the optimiser. We make the concept of stopping condition formal and then we show that the set of traces over all functions is algorithm independent for any stopping condition. In other words, an NFL result still holds.

Definition 12 (*Stopping condition*) A stopping condition is a function mapping output traces to either 0 or 1, $S : \mathscr{T}_Y \to \{0, 1\}$. The stopping condition can be thought of as looking at the results of the optimisation algorithm so far and deciding whether to continue searching. After each function evaluation an optimisation algorithm using a stopping condition S evaluates S on its current output trace T_Y and then stop iff $S(T_Y) = 1$.

Using the above definition we now state and prove a no free lunch result for arbitrary stopping conditions.

Theorem 7 (stopping-condition NFL) *For any stopping condition S, all optimisation algorithms produce the same set of traces when run over all possible functions between two finite sets. More formally, let* $Y^X = \{f \mid f : X \to Y\}$*, where X and Y are arbitrary finite sets. For any optimisation algorithms* A, B*,* $\{T_{A|S}(f)_Y \mid f \in Y^X\} = \{T_{B|S}(f)_Y \mid f \in Y^X\}$*, where* $T_{A|S}(f)_Y$ *is the output trace generated by algorithm A using stopping condition S running on function f.*

Proof From Theorem 1, we know that the behaviour of an algorithm can be represented by a tree. A stopping condition can be seen as a pruning of this tree. Whereas in the k-step NFL proof we cut each branch after the same number of steps, a stopping condition can potentially prune branches after differing numbers of steps.

As we have seen above in Theorem 6, all algorithms produce the same trace tree (see Fig. 6). The stopping condition leads to a pruning of this trace tree, specifically we prune all the children from any leaf with a trace T such that $S(T) = 1$.

A particular stopping condition, then, leads to a particular pruning. As the trace tree and the pruning are both algorithm independent the resulting pruned trace tree will be the same for all algorithms, and thus its leaves (the final traces produced) will be the same. It follows that, for any optimisation algorithms A, B, $\{T_{A|S}(f)_Y \mid f \in Y^X\} = \{T_{B|S}(f)_Y \mid f \in Y^X\}$.

5.3 Stochastic No Free Lunch

We now state and prove the basic no free lunch result for stochastic optimisation algorithms. The stochastic case was also considered in Wolpert and Macready's original papers [1, 2].

Definition 13 (*Stochastic Optimiser*) A stochastic optimisation algorithm is an optimiser that uses a stochastic sampling policy to choose where it samples. Previously a sampling policy was a function mapping $s : \mathscr{D} \to X$. In the case of a stochastic optimiser the sampling policy is instead a function $s : \mathscr{D} \to P_X$ where P_X is a probability distribution over X. To select the next x to sample given data d a sample is drawn from the distribution $s(x)$.

Theorem 8 (stochastic NFL) *Let $Y^X = \{f \mid f : X \to Y\}$, where X and Y are arbitrary finite sets. For any stochastic optimisation algorithms A, B, if we sample f uniformly at random from Y^X then $P(T_A(f) = t) = P(T_B(f) = t) = \frac{1}{|Y|^{|X|}}$ for all full length traces $t \in \mathscr{T}$.*

Proof Let A be a stochastic optimisation algorithm. This stochastic behaviour is equivalent to sampling a deterministic algorithm from some probability distribution over algorithms, then running that. However, we know from the original no free lunch result that regardless of the deterministic algorithm chosen, each trace is equally probable when each function is equally probable, thus each trace has probability $\frac{1}{|Y|^{|X|}}$ regardless of the stochastic optimiser used.

6 Refined and Generalised No Free Lunches

Since their original publication the NFL theorems have been augmented and specialised in various ways. In this section we survey these extensions, providing intuitive explanations of the results, as well as proofs and examples. We start with two definitions that are used in the extensions.

Definition 14 (*CUP*) Let G be a set of functions mapping X to Y. We say G is closed under permutation, or CUP, iff for any permutation $\phi : X \to X, f \in G \implies f_\phi \in G$.

Definition 15 (*Permutation Closure*) Let G be a set of function mapping X to Y. We define G_{cup} as the smallest set containing G that is closed under permutation.

6.1 Optimisation Algorithms Are Bijections

In this section we show that an optimisation algorithm can be seen as defining a function mapping $\mathscr{F}$ to itself, and that this function is a bijection, and thus a permutation. We also look at the behaviour of an optimiser when run on a particular function f, and we will see that the output trace produced can be interpreted as a function permutation of the input f. Results relating optimisation algorithms to permutations are worked through in detail in [12]. We start with a definition:

Definition 16 (*Shuffle Permutation*) $\pi : Y^X \to Y^X$ is a shuffle permutation if it is a bijection and, if $\pi(f) = g$ then there exists a permutation $\phi : X \to X$ such that $\forall x \in X$, $f(x) = g(\phi(x))$. That is, g is a function permutation (see Definition 3) of f. Intuitively, a shuffle permutation maps a function to new function with the same output values, but rearranged to correspond to different inputs.

Now we will show the sense in which an optimisation algorithm can be thought of as a map from $\mathscr{F}$ to itself. Recall that without loss of generality we assume that $X = \{1, 2, \ldots, n\}$. Recall also that $T_A(f)_Y$ is the full output trace of optimiser A running on function f. Thus $T_A(f)_Y$ is an ordered list of n output (i.e. $y \in Y$) values. Given this trace a new function, g, can be defined as $g(i) = T_A(f)_Y(i)$, where $T_A(f)_Y(i)$ is the i-th value of the output trace (i.e. the i-th y value encountered during the optimisation).

We now show that any optimisation algorithm naturally implies a map from Y^X to itself, and in fact this map is a shuffle permutation.

Theorem 9 (Optimisation Algorithms Imply Shuffle Permutations) *Let A be an arbitrary optimisation algorithm, then $T_A(.)_Y$ is a bijection $T_A(.)_Y : Y^X \to Y^X$. Further this bijections is a shuffle permutation.*

Proof That the implied map is a bijection follows from Theorem 1. Next we note that from their definition the optimisers are non-resampling and eventually sample every point. It follows that for any input function the output trace is just a reordering of the function's y values. Thus, the bijection is a shuffle permutation.

6.2 Representation Invariance

The representation of a problem is generally considered important for optimisation. However, an interesting corollary of NFL is that the representation doesn't matter when considering the ensemble of all possible problems. In other words, there is a representational no free lunch: No representation scheme is better than any other under any metric for any optimisation algorithm when average performance over all problems is considered. This representation invariance was made explicit in [13].

Theorem 10 (Representation Invariance *[13]*) *Given a function $h : X \to Y$ we can re-represent the problem by introducing a set C and a surjective map $g : C \to X$ and then considering a new function $f : C \to Y$ where $f(c) = h(g(c))$. As C is surjective then we know $|C| \geq |X|$. Then for any optimisers A, B,*

$$|C| = |X| \implies \{T_A(h)_Y | h \in Y^X\} = \{T_B(h)_Y | h \in Y^X\} = \{T_A(f)_Y | f \in Y^C\}$$

$$|C| > |X| \implies \{T_A(f)_Y | f \in Y^C\} = \{T_B(f)_Y | f \in Y^C\}$$

A full proof can be found in [13], however, it can be seen that the case when $|C| = |X|$ follows directly from Theorem 6.

6.3 Sharpened No Free Lunch

The Sharpened No Free Lunch Theorem (SNFL) was first presented in [14]. Whereas the original no free lunch theorem is a statement about algorithms on the set of all functions, SNFL shows that the result still holds even when we restrict consideration to certain subsets of function. In particular, SNFL states that all optimisation algorithms are equivalent over any subset of functions closed under permutation (see Definition 14).

Theorem 11 (SNFL [14]) *Let $G \subseteq Y^X$ be closed under permutation, then for any optimisation algorithms A, B, $\{T_A(f)_Y \mid f \in G\} = \{T_B(f)_Y \mid f \in G\}$.*

Proof We have seen in Theorem 9 that the output trace produced when an optimiser is run on a function f is always some permutation of f. In the same theorem we also saw that for any optimisation algorithm A, for any two functions $f, g \in \mathscr{F}$ $f \neq g \implies T_A(f)_Y \neq T_A(g)_Y$. From these two facts it follows that if the input set is permutation closed, then, regardless of the optimisation algorithm used, the set of output traces will be this same set of functions. In other words, any optimisation algorithm is a permutation on any permutation closed set of functions. $\{T_A(f)_Y \mid f \in G\} = G$ for any optimisation algorithm A. It follows that $\{T_A(f)_Y \mid f \in G\} = \{T_B(f)_Y \mid f \in G\}$.

Many researchers have examined the realism of the closed under permutation condition for real problems. In particular, Igel and Toussaint [15] show that the proportion of subsets of functions that are closed under permutation tends to zero double-exponentially as the size of the domain of the functions increases.

6.4 Focused No Free Lunch

The Focused No Free Lunch Theorem (FNFL) is an extension of SNFL (see Sect. 6.3). Essentially it shows that, when only considering a restricted set of optimisation algorithms, a (potentially very small) subset of the permutation closure of a test function is enough for NFL to hold. This result was first presented in [3]. Intuitively, because of the restriction to a subset of algorithms a more focused result is possible as there are fewer requirements to satisfy.

Theorem 12 (FNFL [3]) *Let β be a set of test functions, $\beta = \{f_1, f_2, \ldots, f_m\}$, and let $\mathscr{A}$ be a set of optimisation algorithms, $\mathscr{A} = \{A_1, A_2, \ldots, A_n\}$. Then there exists a "focused set" $F_{\mathscr{A}}(\beta)$, with $\beta \subseteq F_{\mathscr{A}}(\beta) \subseteq \beta_{CUP}$ such that all algorithms in $\mathscr{A}$ produce the same set of traces over $F_{\mathscr{A}}(\beta)$. Moreover, this focused set $F_{\mathscr{A}}$ can potentially be*

much smaller than β_{CUP} (note β_{CUP} is just the permutation closure of β, see Definition 15).

The proof can be found in [3], here we omit the proof and instead include some simple illustrative examples.

Example 9 (Simple FNFL Example) Consider optimising functions mapping between $\{1,2,3,4,5\}$ and $\{0,1\}$. Let $\beta = \{01010\}$, call this function f_1 (i.e. $f_1 = 01010$). Consider two optimisation algorithms: A, which inspects the function from left to right, and B, which inspects the function from right to left. $T_A(f_1)_Y = 01010 = T_B(f_1)_Y$, and thus $F_{\mathscr{A}}(\beta)$ for $\mathscr{A} = \{A, B\}$ is just $\{01010\}$.

Example 10 (Second FNFL Example) Again consider optimising functions mapping between $\{1,2,3,4,5\}$ and $\{0,1\}$. Let $\beta = \{00110\}$, let $f_1 = 00110$. Consider two optimisation algorithms, A, which inspects the function from left to right, and B, which inspects the function from right to left. $T_A(f_1)_Y = \{00110, 01100\} = T_B(f_1)_Y$, and thus $F_{\mathscr{A}}(\beta)$ for $\mathscr{A} = \{A, B\}$ is $\{00110, 01100\}$.

6.5 Almost No Free Lunch

Another important extension to the no free lunch theorem is the so called Almost No Free Lunch Theorem. The Almost No Free Lunch Theorem (ANFL) shows that if a stochastic optimiser (Definition 13) performs well on a given function then there is a function of similar complexity on which it performs badly [16].

Theorem 13 (ANFL [16]) *Let H be a randomised optimisation strategy, $X = \{1, \ldots, 2^k\}$, $Y = \{1, \ldots, m\}$ and $f : X \to Y$. Define $c = 2^{k/3}$. Then there exist at least m^{c-1} functions $g : X \to Y$ that differ from f on at most c inputs, such that the probability that H finds the optimum of g within c steps is less than or equal to c^{-1}.*

A proof is given in [16]. Functions of similar complexity here means any of evaluation time, circuit size representation, and Kolmogorov complexity.

Example 11 Let $k = 6$ and $m = 2$. Then $X = \{1, 2, 3, \ldots, 64\}$ and $Y = \{0, 1\}$. In this case ANFL asserts that there are at least $2^3 = 8$ functions $g : X \to Y$ that agree with f on all but at most 4 inputs such that H finds the optimum of g within 4 steps with probability bounded above by $\frac{1}{4}$.

6.6 Restricted Metric No Free Lunch

In this section we introduce a Restricted Metric No Free Lunch Theorem (RNFL) as an extension of the FNFL. Towards the end of [8] English notes the need for an NFL theory for the case of restricted metrics. There has been some work towards

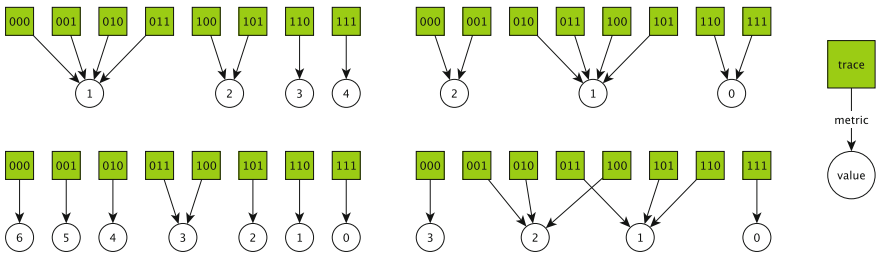

Fig. 8 Four example metrics showing mappings of traces to values. The values are determined as follows (clockwise from top left): (**1**) number of samples until a zero is found. (**2**) number of zeros in first two samples. (**3**) number of zeros in the whole trace. (**4**) value based on how many zeros are found, and how quickly they are found

this goal, for example in [17] they restrict attention to maximisation and show that the correspondence between NFL holding and the set of functions considered being closed under permutation breaks down when only considering optimisers running for k-steps. Specifically, they show that under a maximisation metric, the set of functions being closed under permutation is not necessary for NFL type results. This is an example of a restricted metric free lunch in an k-step optimisation setting.

The original NFL theorem and its successors consider arbitrary performance metrics, or equivalently all performance metrics. Here we show how a restriction on the set of metrics considered—a choice of a single metric for instance—yields NFL results on subsets of functions. This is similar to FNFL, except as well as considering restricted sets of algorithms and functions it also considers a restricted set of metrics. The key idea is that we compare the multi-sets of scores assigned to traces rather than the multi-sets of traces themselves. The sets of scores often has fewer unique elements (and can never have more) an d thus there are more situations in which no free lunch results hold. Figure 8 shows how metrics can reduce the set of traces to a smaller set of scores.

Theorem 14 (RNFL) *Let β be a set of test functions, $\beta = \{f_1, f_2, \ldots, f_m\}$, let $\mathscr{A}$ be a set of optimisation algorithms, $\mathscr{A} = \{A_1, A_2, \ldots, A_n\}$, and let $\mathscr{M}$ be a set of optimisation metrics, $\mathscr{M} = \{m_1, m_2, \ldots, m_k\}$. Then there exists a "restricted set" $R_{\mathscr{A},\mathscr{M}}(\beta)$, with $\beta \subseteq R_{\mathscr{A},\mathscr{M}}(\beta) \subseteq F_{\mathscr{A}}(\beta) \subseteq \beta_{CUP}$ such that all algorithms in $\mathscr{A}$ have the same average performance over $R_{\mathscr{A},\mathscr{M}}(\beta)$. A restricted set $R_{\mathscr{A},\mathscr{M}}(\beta)$ always exists, regardless of the choice of β, $\mathscr{A}$ and $\mathscr{M}$. In some cases it is identical to the focus set $F_{\mathscr{A}}(\beta)$ from the FNFL Theorem, but it can also be strictly smaller than the focus set.*

Proof We give a proof by providing an example in which the restricted set is smaller than the focused set. The fact that a restricted set always exists follows from the fact a focused set always exists, thus we need only to show that the restricted set is potentially a subset of the focused set. Let $X = \{1, 2, 3, 4, 5, 6\}$, $Y = \{0, 1\}$, $\beta = \{101001, 001011, 001111\}$, $\mathscr{A} = \{A_1, A_2\}$ where A_1 deterministically samples from left to right and A_2 deterministically samples $f(2), f(4), f(6), f(1), f(3), f(5)$ in that

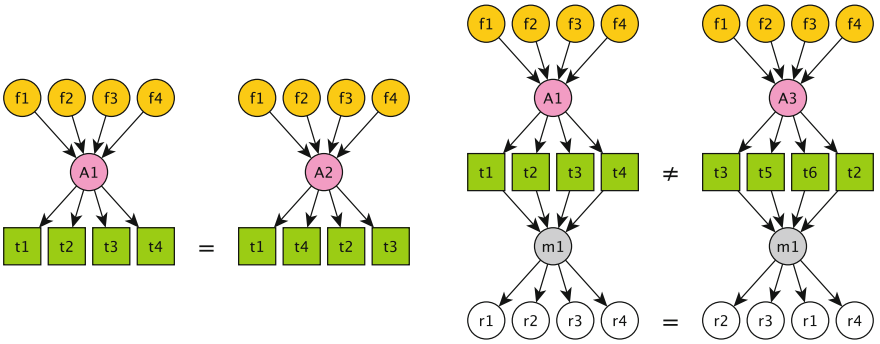

Fig. 9 On the left is a visualisation of a FNFL result for $\beta = \{f_1, f_2, f_3, f_4\}$ and $\mathscr{A} = \{A_1, A_2\}$. Over this set of four functions both optimisers produce the same set of traces, namely $\{t_1, t_2, t_3, t_4\}$. On the right we show that a RNFL can hold when a FNFL does not. For $\beta = \{f_1, f_2, f_3, f_4\}$ and $\mathscr{A} = \{A_1, A_3\}$ FNFL does not hold as A_1 produces the traces $\{t_1, t_2, t_3, t_4\}$ where as A_3 produces $\{t_2, t_3, t_5, t_6\}$. However, if we set $\mathscr{M} = \{m_1\}$ then a RNFL result holds, as both sets of traces lead to the same set of scores, $\{r_1, r_2, r_3, r_4\}$

order. Finally, let $\mathscr{M} = \{m_1\}$ where m_1 just returns the number of samples until the first 1 is found.

If follows from the definitions of the algorithms that, when run on the functions in β, A_1 produces the traces $\{101001, 001011, 001111\}$ and A_2 produces the traces $\{001110, 001011, 011011\}$. These sets of traces are not equal, and when we put these sets of traces through the metric m_1 they result in the multi-set of values $\{1, 3, 3\}$ and $\{3, 3, 2\}$, respectively, which are also not equal.

If we define $R_{\mathscr{A},\mathscr{M}}(\beta) = \{101001, 001011, 001111, 010010\}$ then running A_1 on the functions in $R_{\mathscr{A},\mathscr{M}}(\beta)$ produces the traces $\{101001, 001011, 001111, 010010\}$ and similarly running A_2 produces the traces $\{001110, 001011, 011011, 100001\}$. These sets of traces are still not equal, thus $R_{\mathscr{A},\mathscr{M}}(\beta)$ is not the "focus set" from the FNFL. However, when we put these sets of traces through the metric m_1 they result in the same multi-set of values $\{1, 2, 3, 3\}$. Thus, $R_{\mathscr{A},\mathscr{M}}(\beta)$ is a "restricted set" and we are done. See Fig. 9 for a visualisation of the proof.

6.7 *Multi-objective No Free Lunch*

So far we have considered metrics that map to a scalar. However, within the optimisation literature there is much work on so-called multi-objective optimisation problems, where the metric assigns a vector rather than a scalar. A natural question to ask is whether no free lunch results generalise to these situations. The answer is yes [18, 19]. The proof used in [18] works by defining a bijection between multi-objective problems and scalar problems. A further multi-objective result in [18] is that a no

free lunch holds over the set of all multi-objective functions with any particular shape of Pareto front.

However, as they show in [20], in the case of multi-objective optimisation real world constraints (such as the algorithm only having finite memory) can readily result in algorithms with differing performances. The key idea is that, in scalar optimisation keeping track of the current best solution is straight forward, whereas in multi-objective optimisation problems, where one is searching from the Pareto front, it becomes more practically difficult to store the current best (in this case a set of points making up the Pareto front) efficiently, and so even fairly weak restrictions on the algorithm can mean that practically it is necessary to instead store some sort of approximation of the Pareto front.

Thus, theoretically NFL holds for multi-objective functions, but when it comes to implementation multi-objective optimisers more often need to violate the NFL assumptions for reasons of pragmatism.

6.8 Block Uniform Distributions

In [10] English presents an intuitive necessary and sufficient condition for NFL. He defined *block uniform* distributions, and proved that NFL holds if and only if functions are sampled from a block uniform probability distribution. We state the theorem below, and a proof can be found in the paper.

Definition 17 (*Block-uniform distribution*) A probability distribution over the set of functions $\{f : X \to Y\}$ is block uniform iff $\forall f,\ \forall \phi,\ P(f) = P(f_\phi)$, where ϕ is a function permutation (see Definition 3) (Fig. 10).

Theorem 15 (NFL iff Block Uniform) *For any metric, all optimisation algorithms have the same expected performance if and only if there is a block uniform probability distribution over functions.*

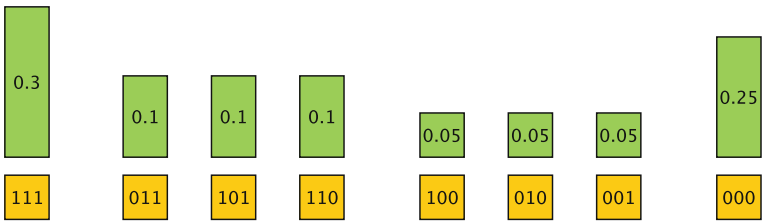

Fig. 10 An example of a block uniform probability distribution over functions $f : \{1, 2, 3\} \to \{0, 1\}$. There are four "blocks", corresponding to the four constituent permutation closed subsets of functions. The figure is not to scale

6.8.1 ϵ-Block Uniform

In [21] Everitt investigates what happens when a distribution is *almost* block uniform. He proves that the amount of free lunch available increases at most linearly with increasing ϵ, where ϵ characterises the distance of a distribution from block-uniform.

Thus, beyond the statement of Theorem 15, the amount of free lunch available is bounded by the distance of the probability distribution over functions from a block uniform distribution.

6.9 Infinite and Continuous Lunches

It should be noted that in this exploration of the NFL we have not considered infinite domains and continuous extensions. However, work to this end can be found in the literature [22–24]. This decision is in part due to the fact that the algorithms in which we are interested will be running on finite sets of possible inputs, and in part because optimisation algorithms in general use run on finite procession machines.

7 Comparing Optimisers After No Free Lunch

In the above sections we have covered in detail the original NFL results, and various extensions. It is evident that, as emphasised in [25], the existence and importance of free lunches is far from a straight forward question, and the opinions of researchers vary. This said, we now try to briefly summarise what the results, when considered as a whole, seem to really mean for algorithm comparison and evaluation:

1. The no free lunch results preclude meaningful comparison of optimisation algorithm's *exploration behaviour* without reference to specific problems.
2. However, almost all restrictions on the set of problem functions result in possible free lunches.
3. Similarly, but more generally, almost all probability distributions over problem functions result in possible free lunches.
4. More specifically, block-uniform distributions capture exactly the scenarios where no free lunch results hold for any metric.
5. However, when we are interested in no free lunch results with respect to particular metrics, and for limited numbers of samples, then free lunches are possible even under block-uniform distributions.
6. When free lunches are possible, their prominence tends to depend crucially on the optimisation metric used.
7. When free lunches are possible, the algorithms that achieve them are *aligned* with the probability distribution over problem functions.

8. When considering more than just the exploration behaviour of an algorithm, algorithms can be ranked. For example, some optimisers are simpler, some are faster and some tend to resample less than others.

Pragmatically the results mean that benchmarking alone cannot be used to evaluate an algorithm, but must be used in combination with clear underlying assumptions on the distribution of problem functions. The benchmark functions must be representative of the problems and there must be some smoothness, in the sense that being good at a problem means that the algorithm is likely to be good at similar problems.

8 Metaheuristic Optimisation After NFL

In this chapter we have explored in detail NFL results, and seen how they preclude superiority of any particular optimisation algorithm in many settings. However, there are many real problems that need solving through optimisation, and many popular, successful metaheuristic optimisers in common use, such as [26, 27]. The questions then, are firstly how can the NFL results be reconciled with the existence of effective optimisation methods, and secondly, how can we use a thorough understanding of NFL to improve research into and development of metaheursitc optimisers?

Optimisation algorithms are able to perform better than others in situations where free lunches exist, which can most generally be understood as situations in which block uniform distributions over functions, as discussed in Sect. 6.8, do not pertain. Just as a compression algorithm cannot universally compress, but must exploit expected input structure to compress well on average, so an optimisation algorithm cannot work well on all possible inputs, but must exploit expected input structure to optimise well on average over the inputs it receives. Luckily, in reality, problems tend to exhibit certain structure, such as local smoothness, or symmetries, and optimisers that exploit this structure *can* do better on average, for that class of problems, than those that do not. Successful metaheuristic optimisers then, are those that effectively exploit common problem structure.

However, despite a general awareness of the existence of the NFL theorems, it is still sometimes the case that new optimisers are presented as a panacea. Of course, the NFL results tell us specifically that this can never be true.

Instead, we must try to characterise the dynamics of optimisation algorithms, to understand their search behaviour, so that we can better understand which algorithms should be used for which problems, and how the algorithms hyper-parameters influence the search dynamics. This sort of investigation has been undertaken for PSO for example [28, 29], where the effects of the hyper-parameters on the search behaviour are considered in detail.

Pursuing a Bayesian understanding of metaheuristic optimisation is another potential approach to the problem of characterising optimisers and understanding which sorts of functions they work best for. Recently, Serafino has been emphasising the important close relationship between Bayesian optimisation and no free lunch

results [7, 30]. Roughly speaking, in a Bayesian framing of optimisation the standard NFL result becomes the claim that informative induction can not be done without prior assumptions. In fact, this is a widely known maxim, the necessity of an inductive bias voiced by various researchers: "A basic insight of machine learning is that prior knowledge is a necessary requirement for successful learning" [31], in other words, "[Y]ou can't do inference … without making assumptions" [32]. Making this relationship more precise, Streeter has shown in [33] that NFL applies only when a certain form of Bayesian learning is impossible.

Metaheursitic optimisation algorithms essentially make implicit assumptions about the kinds of problem function they will be used on, and we should aim to make these implicit assumptions as explicit as possible when either developing an optimiser, or investigating one's behaviour. However, uncovering the particular biases and affinities of an optimisation algorithm has proven to be very difficult, and it is not clear how one can represent these alignments in a general way. As metaheuristic optimisation continues to develop, it will be necessary for understanding of these problems to develop, too.

References

1. Wolpert, D.H., Macready, W.G.: No free lunch theorems for search. Technical report. SFI-TR-95-02-010, Santa Fe Institute (1995)
2. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. In: IEEE Trans. Evol. Comput. **1.1**, 67–82 (1997)
3. Whitley, D., Rowe, J.: Focused no free lunch theorems. In: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, pp. 811–818. ACM (2008)
4. Whitley, D.: Functions as permutations: regarding no free lunch, walsh analysis and summary statistics. In: Parallel Problem Solving from Nature PPSN VI, pp. 169–178. Springer (2000)
5. Culberson, J.C.: On the futility of blind search: an algorithmic view of 'no free lunch. Evol. Comput. **6**(2), 109–127 (1998)
6. Lattimore, T., Hutter, M.: No free lunch versus Occam's razor in supervised learning. In: Algorithmic Probability and Friends. Bayesian Prediction and Artificial Intelligence, pp. 223–235. Springer (2013)
7. Serafino, L.: No Free Lunch Theorem and Bayesian probability theory: two sides of the same coin. Some implications for black-box optimization and metaheuristics. In: (2013). arXiv:1311.6041
8. English, T.: No more lunch: analysis of sequential search. In: Proceedings of the 2004 Congress on Evolutionary Computation CEC2004, Vol. 1, pp. 227–234. IEEE (2004)
9. English, T.: Optimization is easy and learning is hard in the typical function. In: Proceedings of the 2000 Congress on Evolutionary Computation, vol. 2, pp. 924–931. IEEE (2000)
10. English, T.: On the structure of sequential search: beyond 'no free lunch'. In: Evolutionary Computation in Combinatorial Optimization, pp. 95–103. Springer (2004)
11. Ho, Yu-Chi, Pepyne, D.L.: Simple explanation of the no-free-lunch theorem and its implications. J. Optim. Theory Appl. **115**(3), 549–570 (2002)
12. Duéñez-Guzmán, E.A., Vose, M.D.: No free lunch and benchmarks. Evol. Comput. **21**(2), 293–312 (2013)

13. Radcliffe, N.J., Surry, P.D.: Fundamental limitations on search algorithms: evolutionary computing in perspective. In: Computer Science Today, pp. 275–291. Springer (1995)
14. Schumacher, C., Vose, M.D., Whitley, L.D.: The no free lunch and problem description length. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), pp. 565–570 (2001)
15. Igel, C., Toussaint, M.: A no-free-lunch theorem for non-uniform distributions of target functions. J. Math. Model. Algorithm. **3**(4), 313–322 (2005)
16. Droste, S., Thomas, J., Ingo, W.: Optimization with randomized search heuristics—the (A) NFL theorem, realistic scenarios, and difficult functions. Theor. Comput. Sci. **287**(1), 131–144 (2002)
17. Griffiths, E.J., Orponen, P.: Optimization, block designs and no free lunch theorems. Inf. Process. Lett. **94**(2), 55–61 (2005)
18. Corne, D.W., Knowles, J.D.: No free lunch and free leftovers theorems for multiobjective optimisation problems. In: Evolutionary Multi- Criterion Optimization. Springer (2003)
19. Service, T.C.: A no free Lunch theorem for multi-objective optimization. Inf. Process. Lett. **110**(21), 917–923 (2010)
20. Corne, D., Knowles, J.: Some multiobjective optimizers are better than others. In: The 2003 Congress on Evolutionary Computation, Vol. 4, pp. 2506–2512. IEEE (2003)
21. Everitt, T.: Universal indution and optimisation: no free lunch? In: Thesis (2013)
22. Auger, A., Teytaud, O.: Continuous lunches are free plus the design of optimal optimization algorithms. Algorithmica **57**(1), 121–146 (2010)
23. Rowe, J., Vose, M., Wright, A.: Reinterpreting no free lunch. Evol. Comput. **17**(1), 117–129 (2009)
24. Alabert, A. et al.: No-free-lunch theorems in the continuum. In: (2014). arXiv:1409.2175
25. Yang, X.-S.: Free lunch or no free lunch: that is not just a question? Int. J. Artif. Intell. Tools **21**(03), 1240010 (2012)
26. Yang, X.-S., Deb, S.: Cuckoo search via Lévy flights. In: World Congress on Nature & Biologically Inspired Computing, pp. 210–214. IEEE (2009)
27. Kennedy, J.: Particle swarm optimization. In: Encyclopedia of Machine Learning, pp. 760–766. Springer (2011)
28. Poli, R.: Mean and variance of the sampling distribution of particle swarm optimizers during stagnation. IEEE Trans. Evol. Comput. **13**(4), 712–721 (2009)
29. Erskine, A., Joyce, T., Herrmann, J.M.: Parameter selection in particle Swarm optimisation from stochastic stability analysis. In: International Conference on Swarm Intelligence, pp. 161–172. Springer (2016)
30. Serafino, L.: Optimizing without derivatives: what does the no free lunch theorem actually say? In: Notices of the AMS 61.7 (2014)
31. Ben-David, S., Srebro, N., Urner, R.: Universal learning versus no free lunch results. In: Philosophy and Machine Learning Workshop NIPS (2011)
32. David, J.C.M.: Information theory, inference and learning algorithms. Cambridge University Press (2003)
33. Streeter, M.J.: Two broad classes of functions for which a no free lunch result does not hold. In: Genetic and Evolutionary Computation-GECCO, pp. 1418–1430 Springer (2003)

http://www.springer.com/978-3-319-67668-5

Nature-Inspired Algorithms and Applied Optimization
Yang, X.-S. (Ed.)
2018, XI, 330 p. 42 illus., 28 illus. in color., Hardcover
ISBN: 978-3-319-67668-5