No Free Lunch, Program Induction and Combinatorial Problems

John R. Woodward and James R. Neil

School of Computer Science, University of Birmingham, B15 2TT, UK
Email:J.R.Woodward@cs.bham.ac.uk J.R.Neil@cs.bham.ac.uk

Abstract. This paper has three aims. Firstly, to clarify the poorly understood No Free Lunch Theorem (NFL) which states all search algorithms perform equally. Secondly, search algorithms are often applied to program induction and it is suggested that NFL does not hold due to the universal nature of the mapping between program space and functionality space. Finally, NFL and combinatorial problems are examined. When evaluating a candidate solution, it can be discarded without being fully examined. A stronger version of NFL is established for this class of problems where the goal is to minimize a quantity.

1 Introduction

For every problem that an optimizer does well on, there is a corresponding problem on which it does poorly. Over all possible problems the performance of all optimizers are equivalent. There are many ways of stating NFL and a number of papers have been published [1–5]. NFL may seem counter intuitive, but this is because the conditions under which NFL hold are not usually stated. Firstly, no point in the search space is revisited. Secondly, all functions are considered. Finally, the overhead of determining the next point to visit is ignored. These points are discussed and illustrated in section 3.

Program induction involves generating a program and testing it against a serise of test cases [6]. This process is repeated until terminating conditions are met. The method of generation of the program is not of concern in this paper. An error score is assigned to a program depending on its performance on the set of test cases. A number of test cases are needed to reflect the behaviour of the desired program beyond 'reasonable doubt'. The practitioner is faced with the job of providing a set of test cases in the hope that if a program that passes the test cases, it will generalize well. In general, the optimum set of test cases will be highly problem dependent. In this paper, program induction will be taken to mean over a space of programs defined by a Turing Complete instruction set. Often, program induction uses a function set that is specific to the problem and in most cases is not Turing Complete.

Imagine the following typical situation in program induction. A search algorithm produces a series of programs as candidate solutions. As each program is tested on the series of test cases, if at any point the error score of program

exceeds that of the best program so far testing can be halted. Search algorithms that halt testing in this case are called *terminating* search algorithms.

The original motivation of this work was to establish a terminating version of NFL for program evolution which states all terminating search algorithms perform equally. However, due to the nature of the mapping between programs and their functionality, it is found NFL does not hold.

The original motivation can still be applied to combinatorial problems where the goal is to minimize a quantity. A combinatorial problem is one which involves selecting a subset from a set in order to minimize or maximize a certain quantity subject to certain constraints (the ordering may be important for some problems).

Possibly the best know example of a combinatorial problem is the travelling salesman problem, where the goal is to minimize the total distance travelled between a set of cities. If a route is being examined, and after only a few cities it has exceeded the total distance of the best route so far, it can be discarded and hence does not need to be fully evaluated.

Section 2 repeats a number of definitions from [3] which are biult upon on in section 6 in order to prove a terminating version of NFL for combinatorial problems. In section 3 the three reasons why NFL is misunderstood are discussed. NFL and program induction are considered together in section 4 and the claim is made that there is a potential free lunch. In section 5 , combinatorial problems are considered in the context of search algorithms that can terminate. In section 6 a terminating version of NFL is proved.

2 Search Algorithm Framework

This section repeats a framework presented in Schumacher [3] that allows a simple analysis of search algorithms. This framework is perhaps simpler that the one presented by Wolpert and Macready[4, 5].

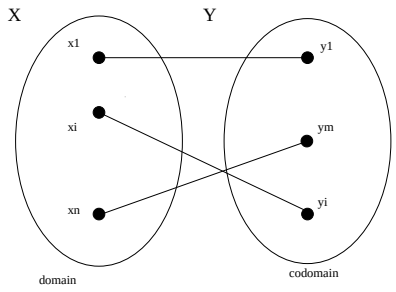

Fig. 1. A function is a mapping from X to Y. Points in the set X map to points in the set Y. In general point x_i maps to point y_i and $|X| \geq |Y|$.

Functions Let X and Y be finite sets and $f : X \rightarrow Y$ be a function where $y_i \equiv f(x_i)$. The size of X is $|X|$ and the size of Y is $|Y|$. Each value in X maps to a single value in Y. For a given X and Y there are $|Y|^{|X|}$ possible functions.

Search Operators and Search Vectors A search operator and a search algorithm are taken to be the same and represent any algorithm that produces a search vector. A search vector V is an ordered sequence of points in X (this is equivalent to the concept of a walk in English [2])$V \equiv \langle x_1, x_2, \ldots, x_m \rangle$. It is assumed that no point is revisited. The ith element in this vector corresponds to the ith point visited. A complete search vector is any vector that lists all points in X once and only once and has length $|X|$. There are $|X|!$ distinct complete search vectors. We are not concerned with how the search vectors are generated.

A search vector corresponds to a path in X. Given search vector and function will produce a corresponding path in Y. Let us call this sequence of points in Y a performance vector. A search vector of length l corresponds to a performance vector of length l, given a specific function.

Define an overall performance measure to be a function that maps the set of performance vectors generated by a given search vector A and a set of functions F to a real number. A NFL result over a set of functions F is defined to exist when two algorithms have the same overall performance measure. In [3] four equivalent statements of NFL are discussed, the third is stated here;

Theorem 1. *NFL: Every search algorithm generates precisely the same collection of performance vectors when all functions are considered.*

3 Discussion of NFL

Revisiting Points NFL only considers unique points visited in X, if a point is visited again it is not counted. Let us look at the issue of revisiting points. Consider two search algorithms, random search and exhaustive search, and a search space of n points. Random search visits points and also has a chance of revisiting points. Exhaustive search moves systematically from point to point visiting each point only once. Exhaustive search is guaranteed to find the target point within n evaluations, while we can only make probabilistic statements about random search. Exhaustive search will do better than random search over all problems due to the fact that random search will revisit points. NFL only considers novel evaluations, so in the above case both algorithms are considered to behave the same (random search can only make n novel evaluations). One might argue that we could supplement the random search algorithm with a memory so it avoids revisiting points, however this will require some computational overhead (see subsection Overheads). Over all functions, exhaustive search is better than random search if revisiting points *is counted.*

All Functions A value in the domain maps to a value in the codomain under one function, but under another function could map to a different value. If

we consider all functions, one value in the domain maps to *every* value in the codomain. If we only consider one function then the best algorithm visits the optimal point first. This means other points will be visited later. Therefore there exists a function where the target point is visited last in the search. For every function a search algorithm does well on, there is a corresponding function on which it does badly.

Overheads The overhead of calculating the next point to visit is not taken into account. If two search algorithms produce the same search vectors, they appear to behave the same in terms of the points they visit, but one may involved a more expensive computation than the other. In terms of wall clock time they will appear to perform differently. Real search algorithms do revisit points but this could be avoided by keeping a record of points visited. However, for any reasonable sized problem space this overhead becomes considerable and cannot be ignored.

When evolutionary algorithms are compared, they are run for a certain number of generations, and so the overheads are ignored. As in the case above comparing exhaustive search with random search, the overhead of maintaining a list of points visited is much more computationally expensive than the overhead associated with a typical exhaustive search algorithm.

4 Program Induction and NFL

Program induction involves searching the space of computer programs to find a target function represented by a set of test cases [6]. In practice, limits are placed on the length of programs, the amount of memory the program can access and the length of time a program can run. Often these limits are imposed by the user, but could be self adapted.

The mapping between the program space and the space of functionality is very specific and is related to the universal distribution and Kolmogorov complexity[7]. Kolmogorov complexity is the length of the shortest program that produces a given functionality. In essence, the universal distribution says there are lots of programs with simple functionality and few with complex functionality and this is independent of the computer language used to express the programs. There is a many to one mapping between the space of programs and the space of functionality. In the terminology of Evolutionary Computation there are many genotypes for a given phenotype [6]. In the more tradional tree based genetic programming

It is interesting to note that a similar universal distribution exists in the more tradional tree based genetic programming provided modularity is permitted in the representation [8].

Langdon [9] has investigated the distribution of functionality of programs with varying length. He suggests that above some certain minimum size threshold, the distribution is largely independent of length. The frequency of output behaviour is plotted for varying program sizes for three different problem domains

(the Boolean problems, symbolic regression, evolving agent) and demonstrates that they each tend towards a limiting distribution and that the distribution is not uniform. While these problem domains do not use Turing Complete instruction sets he states that is seems reasonable that a similar result will also apply to big trees including iteration or recursion and memory (i.e. Turing Complete systems).

So where does the universal distribution leave NFL? Figure 2 shows a space of enumerated programs. There are 7 programs in this space. Imagine algorithm $A \equiv \langle 1,2,4,6,3,5,7 \rangle$ and $B \equiv \langle 1,3,2,5,7,4,6 \rangle$. The programs A visits have functionality $\langle a,b,c,d,a,b,b \rangle$. The programs B visits have functionality $\langle a,a,b,b,b,c,d \rangle$. Within 4 evaluations A has sampled all of the functionality avaliable, while B requires 7 evaluations. Whatever the target functionality is A cannot do worse than B. This diagram demonstrates that there is a free lunch in the situation where there is a non-uniform many to one mapping between spaces as there is with program induction.

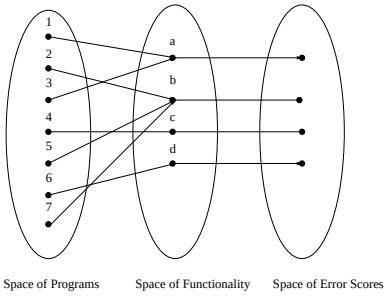

Fig. 2. A space of programs maps to a space of functionality. This mapping is many to one and is related to the universal distribution. A free lunch exists between these two spaces. The space of functionality maps to a space of error scores, between these two spaces NFL still holds.

There are three spaces involved in program induction (see figure 2). A space of programs maps to a space of functionality and this space maps to an error space. If one could directly search the space of functionality we would face a NFL distribution. One can see this by imagining an error function that returns true only if all test cases are passed. This will give a needle in a haystack function and over all functionality it has been show that this obeys NFL [3]. Alternatively the error function could return the number of test cases failed and this would give a binomial distribution. Over all functionality NFL holds for this method of measuring error. We are confined to search the space of programs as we cannot directly access the space of functionality.

One might try to rescue NFL by adding a clause to the theorem excluding programs that have functionality which have been seen before (i.e. by only counting programs with novel functionality). This would be like adding memory

to search algorithms to avoid revisiting points. In the case of program evolution, weeding out such programs is impossible due to Rice's Theorem, which states that non trivial properties of a program cannot be decided. The only way we can determine if two programs have the same functionality over the set of test cases is to execute them. Hence do not have the option of patching up the NFL theorem.

5 Combinatorial Problems and Terminating Search Algorithms

The idea of terminating the testing of a solution can be applied to combinatorial problems were the goal is to minimize a certain quantity. For example in TSP the goal is to minimize the total round trip distance. Another well know combinatorial problem is the knapsack problem, but here the idea is to maximize a quantity. The idea of termination can be applied to combinatorial problems where the goal is minimize a quantity.

To illustrate the idea of termination TSP is used as it is probably the best documented. The goal is to visit all the cities stated by the shortest route. Presented with a number of candidate routes, the round trip distance of each one could be calculated in turn and the best selected. A route is evaluated by accumulating a running total as each city is visited. If at any time the running total of a potential route is greater than the shortest route seen so far, further examination of that route is unnecessary and can be discarded immediately. Hence a route could be ruled out before the total distance has been calculated.

What does the space of these problems look like? Some functions are not available, for example a needle in a haystack function does not exist. A single TSP problem can be represented as a table of cities and distances. From a given scenario other problems can be generated simply by re-labelling the cities. If there are n cities, there are $n!$ ways of re-labelling these cities (i.e. all permutations). Whitley [10] has pointed out that NFL holds for permutations of functions. Strictly speaking each permutation must be distinct. Hence when evaluating a complete route in one evaluation NFL holds.

Assuming we have a search algorithm that can terminate examination of a route when it is possible to do so - do we still have a NFL situation? The aim of the next section is to establish a NFL for terminating search algorithms.

6 Extending The Framework For Terminating Search Algorithms

The current framework needs to be extended to allow us to include the case of terminating search algorithms. The framework described above is suitable for scenarios where we can make statements about a point after a single evaluation. We can think of function optimization, a single evaluation tells us the value of a function at that point and we can say if it is a better point than the best so far. We cannot partially evaluate a point, a point is either evaluated or not.

In the case of combinatorial problems it is possible to make a statement about a potential solution after only partially evaluating it. To evaluate a route in the TSP, the distances between each city in the route are summed up to form the total distance. The complete route is the sum of all the distances that compose the route. If at any point during the journey the running total exceeds that of the best route so far there is no point continuing to evaluate that route.

Let us consider a given configuration of cites under the previous framework. There are n cites, each labelled c_i. There are $n!$ routes to try out, which can be labelled R_i and each route maps to a total distance D_i travelled for that route. A search vector is a list of routes $\langle R_1, R_2, \ldots, R_{n!} \rangle$ and this corresponds to a performance vector $\langle D_1, D_2, \ldots, D_{n!} \rangle$.

Extending this framework involves representing a route as list of each of the cities visited in order so we can talk about partially evaluating a route. Thus, for example $R_1 \equiv \langle c_1, c_2, \ldots, c_n \rangle$, represents a route. A search vector $\langle R_1, R_2, \ldots, R_n \rangle$ may be written out in full, explicitly listing each city e.g.

$\langle \langle c_1, c_2, \ldots, c_n \rangle, \langle c_2, c_1, \ldots, c_n \rangle, \ldots \langle c_n, c_{n-1}, \ldots, c_1 \rangle \rangle$.

This can be called a potential search vector as we may not need to evaluate all points. It represents an ordered list of the cities that may be visited. The potential search vector has a corresponding potential performance vector

$\langle \langle d_1, d_2, \ldots, d_n \rangle, \langle d_2, d_1, \ldots, d_n \rangle, \ldots \langle d_n, d_{n-1}, \ldots, d_1 \rangle \rangle$,

where d_i is the running total of the journey so far (i.e. the distance travelled in visiting the first i cities listed in the route).

Given a potential search vector, the whole of the first route must be examined as there is nothing to compare it against for the purpose of termination. The second element in the potential search vector lists the second candidate route. If at any point during the journey dictated by this route, its accumulated distance exceeds that of the first, then no more cities are visited and the remaining part of the route need not be examined. Similarly, on subsequent routes, cities towards the start of the route will be visited, but cities towards the end of the route may not be. A list of the cites that are actually visited can be stored in a vector called an actual search vector. For example

$\langle \langle c_1, c_2, \ldots, c_n \rangle, \langle c_2, \ldots \rangle, \ldots \langle c_i, \ldots \rangle, \ldots \langle c_n, \ldots \rangle \rangle$.

The dots at the end of each route (except the first) represent the fact that some routes may be partially evaluated. This will have a corresponding actual search vector. For example

$\langle \langle d_1, d_2, \ldots, d_n \rangle, \langle d_2, d_1, \ldots \rangle, \ldots \langle d_i, \ldots \rangle, \ldots \langle d_n, d_{n-1}, \ldots \rangle \rangle$

To summarize these 4 vectors, a potential search vector is an ordered list of all the points that may potentially be evaluated. A potential performance vector is an ordered list of all the values returned by an evaluation, if all the points are evaluated. An actual search vector is an ordered list of all the points that are actually evaluated (remaining points in a potential search vector are not evaluated due to the ability of the algorithm to terminate). An actual performance vector is an ordered list of the actual returned values of points that are evaluated.

The terminating NFL theorem (TNFL) for combinatorial problems can be stated as

Theorem 2. *TNFL: Every* terminating *search algorithm generates precisely the same collection of* actual *performance vectors when all distinct permutations of a function are considered.*

Proof. Let us consider NFL3 stated in[3] which is: Every search algorithm produces the same collection of performance vectors (when all permutations of a function are considered). Let us restate this in the new terminology for terminating search algorithms. Every search algorithm produces the same collection of *potential* performance vectors. Now let us consider the ability of the algorithm to terminate. If one search algorithm terminates early on some routes on one function then as all search algorithms produce the same collection of potential performance vectors there will be an equivalent skipping of cities towards the end of other routes. But these collections are identical therefore the skips made are identical. Hence two terminating algorithms produce the same set of *actual* performance vector.

This proof is stated in terms of TSP but is of course applicable to any combinatorial problem where the goal is to minimize a quantity.

7 Comments and Discussion

It is the author's hope that by stating the three assumptions under which NFL is stated have cleared up any misunderstanding the reader may have had about NFL before reading this paper. It is perhaps interesting to note that in [5] the following is stated

"We cannot emphasize enough that no claims whatsoever are being made in this paper concerning how well various search algorithms work in practice."

Real algorithms do revisit points but this is undesirable. There are two ways to avoid this problem. The first would be to maintain a list of points visits but this is impractical. A second way would be to construct an exhaustive search algorithm, this is commented on later.

In the NFL framework all functions are considered. What is the distribution of problems in the real world? This is a difficult but fundamental question, so a slightly easier and more formal question may be to ask what problems would we reasonably expect to be able to solve on a computer. One of the aims of program induction is to produce a program with the ability to generalize well from a small set of training data. If the target functionality is incompressible it is impossible to generalize as there is no underlying rule so we can say random functions are unlearnable. It does not make sense to talk about program induction on problems that are incompressible. Generally we are interested in problems with some sort of structure that can be exploited and used to make predictions about input data that was not in the training set. Schumacher [3] has show that NFL results are independent whether or not the set of functions is compressible.

Real search algorithms do revisit points. A list of points visited could be maintained to avoid revisits but this would be impractical for most problems. Exhaustive search algorithms will perform a systematic search without revisiting

points. If there are n points in the search space then there are $n!$ ways to visit these points and therefore $n!$ distinct exhaustive search algorithms. In general exhaustive search algorithms will have different lengths and different memory and time requirements. There are many exhaustive search algorithms that cannot be compressed and are essentially just an explicit list of the programs to visit. Other exhaustive algorithms can be written as very short programs, but there are very few of these. Ideally it would be advantageous to visit programs with different functionality but knowing what the functionality of a program is without testing it is undecidable. It appears that some exhaustive search algorithms are better than others.

It is perhaps not surprising that NFL and TNFL hold for combinatorial problems. Combinatorial problems have an intrinsic symmetry and without any knowledge about this beforehand, nothing can be gained.

NFL says all search algorithms perform equally. Now we can say all terminating search algorithms perform equally. This is a restatement of NFL, just at a slightly deeper level. In the standard NFL set up, an evaluation is counted as the evaluation of a candidate solution. In this slightly more refined framework an evaluation counts as evaluation of part of a candidate solution. This different counting system allows us to consider potential savings made by terminating testing early.

8 Summary

NFL is a central but often misunderstood result. NFL holds if revisiting of points is not counted, all functions are considered (or more strictly permutations of a function [10]) and the effort in calculating the next point to visit is ignored. Not revisiting points is unrealistic for most search algorithms, however exhaustive search algorithms are a class of algorithm that only visit points once.

The universal distribution describes the nature of the mapping between the space of programs and the space of functionality. This essentially says there are lots of programs with simply functionality and fewer programs with complex functionality. Due to the universal distribution NFL does not hold for program induction. It seems reasonable that the argument can be extended to other representations (e.g. classifier systems) where the mapping between the representation of a potential solution and its behaviour is a non-uniform many to one mapping. This is backed up the results in Langdon [9]. Attempts may be made to patch up NFL by only counting programs with novel functionality but due to Rice's Theorem this is not possible. Hence, there is a free lunch to be had in program induction, but just how to get at it is not clear.

The idea of terminating search algorithms can be applied to combinatorial problems where the goal is to minimize a quantity. It is shown that there is a terminating version of NFL. This theoretical result may be of little practical importance as the overhead associated with checking the condition for termination may be larger than any potential saving.

Acknowledgements

Michael Vose, Darrell Whitley, Jon Rowe, Xin Yao, Stefano Cattani

References

1. Droste, S., Jansen, T., Wegener, I.: Perhaps not a free lunch but at least a free appetizer. In Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E., eds.: Proceedings of the Genetic and Evolutionary Computation Conference. Volume 1., Orlando, Florida, USA, Morgan Kaufmann (1999) 833–839
2. English, T.M.: Evaluation of evolutionary and genetic optimizers: No free lunch. In Fogel, L.J., Angeline, P.J., B"ack, T., eds.: Evolutionary Programming V: Proc. of the Fifth Annual Conf. on Evolutionary Programming, Cambridge, MA, MIT Press (1996) 163–169
3. Schumacher, C., Vose, M.D., Whitley, L.D.: The no free lunch and problem description length. In Spector, L., Goodman, E.D., Wu, A., Langdon, W., Voigt, H.M., Gen, M., Sen, S., Dorigo, M., Pezeshk, S., Garzon, M.H., Burke, E., eds.: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), San Francisco, California, USA, Morgan Kaufmann (2001) 565–570
4. Wolpert, D.H., Macready, W.G.: No free lunch theorems for search. Technical Report SFI-TR-95-02-010 (1995)
5. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation **1** (1997) 67–82
6. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, dpunkt.verlag (1998)
7. Kirchherr, Li, Vitanyi: The miraculous universal distribution. MATHINT: The Mathematical Intelligencer **19** (1997)
8. Woodward, J.R.: Modularity in genetic programming. In: Genetic Programming, Proceedings of EuroGP 2003, Essex, UK, Springer-Verlag (2003)
9. Langdon, W.B.: Scaling of program fitness spaces. Evolutionary Computation **7** (1999) 399–428
10. Whitley, D.: Functions as permutations: Implications for no free lunch, walsh analysis and summary statistics. In Schoenauer, M., Deb, K., Rudolph, G., Yao, X., Lutton, E., Merelo, J.J., Schwefel, H.P., eds.: Parallel Problem Solving from Nature – PPSN VI, Berlin, Springer (2000) 169–178