,


# SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET


## Universal Induction and Optimisation: No Free Lunch?


av

**Tom Everitt**



2013 - No 3



MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET, 106 91 STOCKHOLM

,

,

# Universal Induction and Optimisation: No Free Lunch?

Tom Everitt

---

Självständigt arbete i matematik 30 högskolepoäng, Avancerad nivå

Handledare: Tor Lattimore, Peter Sunehag och Marcus Hutter

2013

,

# Universal Induction and Optimisation: No Free Lunch?

Tom Everitt

March 18, 2013

**Abstract**

Inductive reasoning is the process of making uncertain but justified inferences; often the goal is to infer a general theory from particular observations. Despite being a central problem in both science and philosophy, a formal understanding of induction was long missing. In 1964, substantial progress was made with Solomonoff's *universal induction*. Solomonoff formalized Occam's razor by means of algorithmic information theory, and used this to construct a universal Bayesian prior for sequence prediction. The first part of this thesis gives a comprehensive overview of Solomonoff's theory of induction.

The optimisation problem of finding the arg max of an unknown function can be approached as an induction problem. However, optimisation differs in important respects from sequence prediction. We adapt universal induction to optimisation, and investigate its performance by putting it against the so-called No Free Lunch (NFL) theorems. The NFL theorems show that under certain conditions, effective optimisation is impossible. We conclude that while universal induction avoids the classical NFL theorems, it does not work nearly as well in optimisation as in sequence prediction.

**Acknowledgements**

Thanks to my supervisors Tor Lattimore, Peter Sunehag and Marcus Hutter at ANU. An extra thanks to Tor Lattimore for many enlightening discussions, and to Marcus Hutter and ANU for hosting me for this Master's thesis.

# Contents

# List of figures



# List of tables

# 1 Introduction

The goal of optimisation is to find an input (to some system) that yields a high output. When an input is tried, you become aware of the associated output, but trying inputs (probing) is costly. The goal is therefore to only try a small number of inputs before finding one that yields a high output. In other words, the goal is efficient optimisation in the number of probes.

What information is required for efficient optimisation to be possible? This will be the central question of this thesis. Formally it is natural to represent an optimisation problem with an unknown function; the goal is then to quickly find the arg max of the function. We also need a representation of the information/uncertainty we have about the function. Adopting a Bayesian perspective, the information about the function can be represented as a *prior* probability distribution over the class of all functions. For example, if one function $f$ has probability 1 and all other functions probability 0 in the prior, then this represents a complete certainty in $f$ being the true function. In this case the maximum should easily be found in only one probe (disregarding computational aspects).

More realistically, the problem might be to find an ideal input to a system that is only partially known. For example, the task might be to find the ideal amount of gas to input into the ignition of a car engine, and the only thing known about the system might be that it is described by a low-degree polynomial. This situation can be represented by a prior with relatively high weight on low-degree polynomials, and low or zero weight on other functions.

As different inputs are supplied to the system and the outputs of those inputs are measured, the knowledge of the system grows. For example, if an input $x$ is found out to map to some $y$, then all functions not consistent with this behaviour may be discarded. The prior may be updated to a *posterior* distribution where inconsistent functions receive probability 0 (it is now certain they are not the true function), and consistent functions get a corresponding upweighting (cf. Bayes' rule). If the prior with high weight on low-degree polynomials was correct, then only a few probes should be required to detect which function describes the input-output relation, so a good input should be found quickly.

But what if nothing is known about the system the function describes, and the only information one has about the function is its input-output behaviour in some probed points? Does that make it impossible optimise the function efficiently? or is there a universal principle for how to optimise a function when nothing else is known? Most humans seem to have the intuition that from seeing, say, 100 points of a function, they can often discover the pattern of the function and (somewhat) accurately extrapolate its behaviour in unseen points. Is there any formal justification for such a claim?

Let us begin with a negative observation. In answer to overly bold claims about the universal performance of some search algorithms, Wolpert and Macready [WM97] showed a number of results called No Free Lunch (NFL) theorems. Essentially they proved that if a search algorithm does well on some set of functions, then that must be mirrored by bad performance on another set of functions. More formally, they showed that all search algorithms will perform the same in uniform expectation over all functions (see Part II for more details).

The NFL theorems were generally interpreted to say that a problem-specific bias is required for efficient optimisation. The argument roughly goes like this: The uniform prior is *the* most unbiased prior, as it gives equal weight to all functions. The NFL theorems show that it is impossible to optimise well under the uniform prior. Hence a problem specific bias is necessary for efficient optimisation.

This argument is *not* accepted in this thesis. It can be shown that if a function is sampled from the uniform distribution, then with high probability the function exhibits no particular structure (is algorithmically random). Obviously, if no pattern exists, no intelligent optimiser (humans included) can find a pattern for extrapolating the function behaviour. Efficient optimisation thus becomes hopeless. In contrast, functions that do behave according to some pattern should—at least in principle—be possible to optimise efficiently. Remarkably, the notion of structure and randomness can be given formal definitions based on Kolmogorov complexity described in Section 3.

Along these lines it may be argued that the uniform prior is not bias-free but biased towards randomness, and that this explains the difficulty in optimising under a uniform prior. A principled alternative called the *universal distribution* exists. The universal distribution is based on Kolmogorov complexity, and is designed to give high weight to structured problems and low weight to random ones. It is often advocated as a formalisation of Occam's razor [RH11].

Since the universal distribution is only biased towards structure *per se*, it does not favour any particular problem over another. It does give lower probability to random functions than the uniform distribution, but since random functions are next to hopeless to optimise efficiently, this should not be seen as a major deficit. Rather, this is precisely what gives an intelligent optimiser a fair chance to find a pattern.

In sequence prediction, which is a rather general induction setting, a predictor based on the universal distribution has been shown to do exceptionally well on sequences generated by computable distributions. This induction principle is called *universal induction* or *Solomonoff induction*.

The main goal of this thesis is an adaption of universal induction to optimisation. A similar venture for Supervised Learning was made in [LH11]. Our most important results include a proof that the NFL theorems do not

apply to the universal distribution, as well as some upper bounds on the "amount of free lunch" under the universal distribution (Section 11 and 12). Part II also contains a number of minor contributions, indicated in its introduction.

Before the new contributions, we will provide background on two areas. Part I explains Kolmogorov complexity and Solomonoff induction. The first sections of Part II recounts the most important NFL results, including a literature review on NFL and optimisation in Section 8. Throughout I use "we" rather than "I", so that the thesis can be consistent with a recent paper submission [EL13].

# Part I
# Kolmogorov Complexity and Universal Induction

In this part we give an account of Kolmogorov complexity and universal (Solomonoff) induction. The two main sources are [LV08, Hut05]. The first offers an expansive exposition on Kolmogorov complexity, including a wide range of applications. The second is more concise and primarily directed towards Artificial Intelligence. The aim here is to give a comprehensive overview of the core results of Kolmogorov complexity and to lay a foundation for applications to optimisation in Part II. Unless otherwise mentioned, results and definitions found in this part (Part I) are from the previously mentioned sources; further discussion and motivation can be found in them.

## 2 Information

### 2.1 Strings

Binary strings are natural objects for representing information. Formally, let $\mathbb{B} = \{0, 1\}$ and define a binary string $s$ as a sequence $s_1 \dots s_n$ with $s_i \in \mathbb{B}$ for $1 \leq i \leq n$. We say that the *length* of a string $s = s_1 \dots s_n$ is $n$, and denote it by $\ell(s) = n$. We use the notation $s_n$ and $s_{m:n}$ to extract the $n$th and the $m$-to-$n$th bits of $s$ respectively.

The letters $s$, $t$ and $q$ will be used for arbitrary strings, and $\epsilon$ for *the empty string* (of length 0). The set of all strings of length $n$ is denoted by $\mathbb{B}^n$; the set of all finite strings is denoted by $\mathbb{B}^* = \bigcup_{n\in\mathbb{N}} \mathbb{B}^n$; and the set of all finite, *non-empty* strings is denoted by $\mathbb{B}^+ = \mathbb{B}^* - \{\epsilon\}$. The letter $z$ will be used for one-way infinite sequences $z_1 z_2, \dots$ with $z_i \in \mathbb{B}$. The set of all one-way infinite sequences will be denoted by $\mathbb{B}^\infty$.

A number of operations on strings and one-way infinite sequences can be defined. Let $s = s_1 \dots s_n$, $t = t_1 \dots t_m$ and $z = z_1 z_2, \dots$. The *concatenation* of $s$ and $t$ is written $st = s_1 \dots s_n t_1 \dots t_m$, and has length $\ell(st) = \ell(s) + \ell(t)$. The concatenation of $s$ with an infinite sequence $z$ is the infinite sequence $sz = s_1 \dots s_n z_1 z_2 \dots$. The empty string $\epsilon$ is the *identity element* for concatenation; that is, $\epsilon s = s\epsilon = s$ for all strings $s$, and $\epsilon z = z$ for all one-way infinite sequences $z$.

*Exponentiation* is defined thus: Let $s \in \mathbb{B}$ and $n \in \mathbb{N}$. Then

$$s^n = \underbrace{s \dots s}_{n \text{ times}}$$

For example, $0^3 = 000$.

Let $s = tq$ be a string. Then $t$ is said to be a *prefix* of $s$, and $s$ is said to be an *extension* of $t$. If $q \neq \epsilon$, then $t$ and $s$ are said to be *proper* prefixes and extensions respectively. A prefix $s = z_{1:n}$ of $z$ is called an *initial segment* of $z$.

## 2.2 Integers and strings

It will be convenient to identify natural numbers with strings in a somewhat non-standard manner. Using the lexicographical enumeration of

$$\mathbb{B}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$$

identify any natural number $i$ with the $i$th element of this enumeration. So, for example, 0 is identified with $\epsilon$ and 5 with 01. This differs from the more standard assignment where for instance 2 corresponds to 10, 010, 0010, etc. The primary benefit of our identification is that the correspondence is bijective.

Using this identification, the length of a number $i$ may be defined as the length of its corresponding string. It is easily verified that the length grow logarithmically with the number; more precisely, $\log_2(i) \leq \ell(i) \leq \log_2(i+1)$ for all $i \in \mathbb{N}$ [LV08, p. 14].

## 2.3 Prefix codes

What information does a string contain? This completely depends on the *coding*. To have a string represent something, a *code* has to be defined that assigns a meaning to each string. Strings with assigned meaning are called *code words* or just *words*. For example, the (Extended) ASCII-code used on many computers, assigns symbols to binary strings of length 8. In ASCII, the letter A is encoded by 01000001 and B by 01000010.[1] When typing on a computer, the letters are *encoded* as binary strings according to the ASCII-code, and typically stored in some file. The letters may later be retrieved, *decoded*, using the ASCII-code in the "opposite direction".

**Definition 2.1** (Codes)**.** A *code* is a partial function $C : \mathbb{B}^* \rightharpoonup X$, where $X$ is known as the set of *objects*. The strings $w \in \mathbb{B}^*$ on which $C$ is defined are known as the *code words* of $C$. If $C(w) = s$ we say that $w$ is a *$C$-code word for $s$*, and say that $s$ is the *meaning* or *object* of $w$.

Several ASCII-code words can be appended to each other with maintained decodability. This is not a property of all codes. Imagine, for instance, that we had chosen the code 0 for A, the code 1 for B, the code 01 for C and so on. Then it would not be clear whether 01 should be decoded as AB or as C.

[1] See for instance `http://www.ascii-code.com/` (accessed February 12, 2013) for a full list of the ASCII code words.

For the ASCII-code, the property that guarantees unique decodability is that all codes have the same length. This is an undesirable restriction in some cases. Sometimes one object is much more frequent than another, in which case it may be advantageous to assign a shorter code word to the frequent object. In other cases, we may want to construct a code for infinitely many objects (such as for the natural numbers). In both these cases, code words of varying length are required.

The general theory of *prefix codes* can be used to construct codes with code words of different lengths, while retaining the unique decodability of appended code words.

**Definition 2.2** (Prefix codes)**.** A code $C$ is a *prefix (free) code* if no code word of $C$ is a proper prefix of another code word of $C$.

The prefix property makes it possible to tell where one code word stops and the next starts in a sequence of appended code words. Intuitively, the sequence can be read bit by bit until a code word is found. Since this code word is not a prefix of any other code word, it must be correct to decode the code word immediately.

The ASCII-code is prefix, since no code word of ASCII is a proper prefix of another (this is obvious since all code words have the same length). And accordingly, it is always possible to tell where one code word stops and the next starts in an ASCII-sequence (given the starting point of the first code word). A more interesting example is the $1^n0$-code for the natural numbers (Table 1). The idea is very simple: encode every number $n$ with $n$ 1's

| Number | code word |
|---|---|
| 0 | 0 |
| 1 | 10 |
| 2 | 110 |
| 3 | 1110 |
| ⋮ | ⋮ |
| $n$ | $1^n0$ |
| ⋮ | ⋮ |

Table 1: The $1^n0$-code for numbers.

followed by a 0. This way no code word can be a prefix of another. As an example, the sequence 11011100 can only be decoded as 2,3,0:

$$\underbrace{110}_{2}\,\underbrace{1110}_{3}\,\underbrace{0}_{0}$$

An important point is that the starting point has to be fixed. In many prefix codes, it is not possible to start in the middle of a sequence and decode it correctly (the $1^n0$-code is an exception).

## 2.4 Standard codes

It is now time to introduce a number of standard encodings, which we will frequently rely on in the subsequent developments.

### 2.4.1 Strings

A standard prefix code for strings which will be of much use is the following. Encode every string $s$ as $\overline{s} = 1^{\ell(s)}0s$. That is, prefix $s$ with the $1^n0$-code word for $\ell(s)$, so that the encoder knows how long $s$ is before starting to read it.

For example, the string 101 will be encode as $\overline{101} = 1110101$, because

$$\overline{101} = \underbrace{111}_{1^{\ell(101)}}\underbrace{0}_{0}\underbrace{101}_{101}$$

The empty string $\epsilon$ has code word $\overline{\epsilon} = 1^{\ell(\epsilon)}0\epsilon = 0$ (so $\ell(\overline{\epsilon}) = 1$).

### 2.4.2 Pairs and tuples

The standard prefix code for strings can be used for a standard code of pairs of strings. Pairs of strings will be encoded as $(s,t) = \overline{s}t$. This makes it clear where $s$ stops and $t$ starts. The length of a pair $(s,t)$ is $\ell(s,t) = \ell(\overline{s}t) = 2\ell(s) + 1 + \ell(t)$. By intention, this does not entail a prefix code for both $s$ and $t$. The reason for this choice of definition will become clear in the definition of Kolmogorov complexity in Section 3. To obtain a prefix code for both $s$ and $t$, the construction $\overline{(s,t)} = \overline{\overline{s}t}$ may be used.

Triples $(s_1, s_2, s_3)$ are encoded as $\overline{s_1}\,\overline{s_2}s_3$ and so on. In situations where the cardinality of the tuple is not clear from the context, such as when an arbitrary tuple can be supplied to a program, $n$-tuples $(s_1, \ldots, s_n)$ can be encoded as $1^n0(s_1, \ldots, s_n) = 1^n0\overline{s_1}\ldots\overline{s_{n-1}}s_n$.

### 2.4.3 Rational numbers

The set of rational numbers $\mathbb{Q}$ can be identified with pairs of natural numbers, together with a sign. For $m, n \in \mathbb{N}$ encode the rational number $m/n$ as $1(m,n) = 1\overline{m}n$, and the rational number $-m/n$ as $0(m,n) = 0\overline{m}n$ (where we have tacitly used the identification between natural numbers and strings described in Section 2.2).

## 2.5 Kraft's inequality

A useful intuition for prefix codes is that by including a short code word—say 10—in our code, we rule out all longer strings starting with 10 as code words (because 10 is a proper prefix of those). In this respect, short code

words take up "more space" in the set of of potential code words than long code words. This intuition has a geometric interpretation.

The set of all one-way infinite binary sequences naturally represent the interval $[0, 1]$ by interpreting sequences as binary expansions of real numbers. In this view, let a (finite) string $s$ correspond to the half-open interval $\Gamma_s = [s000\ldots, s111\ldots)$. Observe that the length of this interval is precisely $2^{-\ell(s)}$. Observe also that by using $s$ as a codeword, no string $s'$ with overlapping interval can be used as a code word.

This is the intuition behind Kraft's inequality (see e.g. [LV08, Section 1.11.2] for a full proof).

**Theorem 2.3** (Kraft's inequality)**.** *There is a prefix code with code words of length $l_1, l_2, \ldots$ if and only if $\sum_i 2^{-l_i} \leq 1$.*

## 2.6 Optimality

The Kraft inequality makes precise the "maximum shortness" of the code words of a prefix code. Consider for example the $1^n0$-code for the natural numbers discussed above. This code is optimal in the sense that the lengths of its code words are maximally short with respect to Kraft's inequality. The code for 0 has length 1, the code for 1 has length 2 and so on, so the total length is $\sum_{i\in\mathbb{N}} 2^{-i} = 1$.

However, in another sense it is rather inefficient. Consider the number $99^{999}$. This number is shortly describable in the standard (informal) mathematical language. To construct a formal code in which $99^{999}$ is short, the ASCII-code discussed above can be used: encode $99^{999}$ with the ASCII-code for 10e+100 as is done on most calculators. This coding only requires $7 \cdot 8 = 56$ bits for $99^{999}$. On the other hand, in the $1^n0$-code the code word for $99^{999}$ requires approximately $10^{1989}$ pages! (assuming we could fit ten thousand 1's on every page). In this sense, the $1^n0$-code is very inefficient.

The ASCII-coding pays the price for the shorter code word of $99^{999}$ by having longer code words for the small natural numbers. While the number 2 requires only 3 bits in the $1^n0$ code, it requires 8 bits in ASCII.

Which code is better depends on the situation. The $1^n0$ code is better if one mainly wants to encode small numbers, and the ASCII-code is better if one (sometimes) wants to encode larger numbers. There is, however, an objective sense in which the ASCII-based code is preferable: No number has a substantially longer ASCII-code word than $1^n0$-code word, but some numbers (such as $99^{999}$) have much shorter ASCII-code words than $1^n0$-code words. This is the idea of additively optimal codes, developed further in the next section.

# 3 Kolmogorov complexity and additively optimal prefix codes

In this section the aim will be to quantify the information content of a string $s$, called the *Kolmogorov complexity* of $s$. The intuition is that while some strings have no shorter description than the string itself, other strings are significantly compressible. For example, the string $t$ of a thousand consecutive 0's is highly compressible since $t$ has a short description. In this sense, $t$ has low information content. Conversely, to describe a random string, one generally needs to describe each single bit by itself. Such incompressible strings are said to have high information content.

That there is an (essentially) objective measure of information content is somewhat surprising. The measure is obtained by the construction of an *additively optimal code* for strings (in this section, all codes describe strings).

**Definition 3.1** (Additive optimality)**.** A code $C$ is *additively optimal* for a class $\mathcal{C}$ of codes, if for all $C' \in \mathcal{C}$ exists a constant $c$ such that for all strings $s$, the existence of a $C'$-code word $w'$ for $s$ implies the existence of a $C$-code word $w$ for $s$ with $\ell(w) \leq \ell(w') + c$.

As an example, recall the comparison between the $1^n0$-code for numbers and the ASCII-code for numbers. In the class of these two codes, only the ASCII-code was additively optimal (it is possible to construct numbers that the the ASCII-code has arbitrarily much shorter code words for but not the other way around). Other code-classes may have several additively optimal elements.

Unfortunately, there is no additively optimal prefix code for the class of all prefix codes. But in the restriction to effective prefix codes (defined in a moment) an additively optimal element exists. The first aim of this section will be to develop such an additively optimal effective code. The existence will then be used to define Kolmogorov complexity, which can be interpreted as an objective measure of the information content of a string.

## 3.1 Prefix-machines

Turing-machines correspond to partial recursive functions (see, for instance, [Cut80]). It is natural to say that a code is *effective* if it is a partial recursive function. Some, but not all, partial recursive functions define prefix codes. Define a *prefix-function* $f : \mathbb{B}^* \rightharpoonup \mathbb{B}^*$ as a partial function with the property that if $f(w)$ is defined, then $f(wt)$ is undefined for all $t \neq \epsilon$; that is, $f$ is a function defining a prefix code. Consequently, effective prefix codes are partial recursive prefix-functions.

Just as Turing-machines correspond to partial recursive functions, there is a type of machine corresponding to the partial recursive prefix-functions.

It is called a *prefix-machine*, and is essentially a Turing-machine with slightly limited input-output behavior.

**Definition 3.2** (Prefix-machines)**.** A *prefix-machine* $V$ is a Turing-machine [Cut80] modified in the following way. $V$ has a one-way infinite input tape and a one-way infinite output tape, allowing only 0 and 1 as symbols. $V$ also has a one-way infinite work-tape that allows blank symbols # in addition to 0 and 1. $V$ has a reading head for the input tape that can only be moved to the right; and $V$ has a writing head for the output tape that always moves one step to the right after it has written a symbol. The writing head cannot be moved in any other manner.

If $V$ halts after having read an initial segment $w$ of the input tape (the input head is on the rightmost symbol of $w$), and the string to the left of the writing head is $s$, then we say that $V$ *outputs* $s$ *on input* $w$, and write $V(w) = s$.

Note that by the restriction in input-output behaviour, if a prefix-machine $V$ halts on $w$, then $V$ does not halt on any proper prefix of $w$, nor does it halt on any proper extension of $w$. This shows that prefix-machines define partial recursive prefix-functions.

Conversely, the following proposition shows that every partial recursive prefix-function is computed by a prefix-machine (using the fact that every partial recursive function is computed by a Turing-machine).

**Proposition 3.3** (Prefix-machines and partial recursive prefix-functions)**.** *For every partial recursive prefix-function $f$, there is a prefix-machine $V$ such that $V(w) = s$ whenever $f(w) = s$.*

*Proof.* For a given partial recursive prefix-function $f$, there is a Turing-machine $V'$ computing it. Using $V'$, we can construct a prefix-machine $V$ that also computes $f$.

Let $V$ have a variable $w$ initialized as the empty string $\epsilon$, and let $V$ execute the following loop. For growing $k \geq 0$, $V$ simulates $V'$ with inputs $wt$ with $t$ of size at most $k$ for $k$ time steps, until a halting input $wt$ is found (if not, $V$ runs forever). If $t = \epsilon$, then $V$ outputs the output of $V'(w)$. If not, then $V$ reads one more symbol $\chi$ from the input tape, sets $w = w\chi$ and restarts the loop.

Since $V'$ computes a prefix-function, if $V'$ halts on an input $w$ then $V'$ will not halt on any proper prefix or extension of $w$. So whenever $V$ gets an input sequence starting with $w$, then $V$ will halt exactly when it has read $w$, and output $V'(w)$. And if the input sequence given to $V$ does not contain any initial segment on which $V'$ halts, then $V$ will not halt either. So $V$ computes the same partial function $f$ as $V'$. □

We have thus established that there is a natural correspondence between prefix-machines and partial recursive prefix-functions/effective prefix codes.

## 3.2 Universal prefix-machines

A pivotal property of the Turing-machines is that they can be described by strings, and that there is a universal Turing-machine $U$ that simulates any Turing-machine $V$ given the string-description of $V$. By a similar argument as is used for Turing-machines, the prefix-machines can be described by strings. Henceforth the string-descriptions of prefix-machines will be treated as numbers (according to the string-number identification described in Section 2.2), which yields an effective enumeration $V_1, V_2, \ldots$ of all prefix-machines (where $V_i$ is the machine described by $i$).[2]

To enable prefix-machines to take more than one argument, we will make use of the standard encoding of pairs (tuples) described in Section 2.4.2. For example, $V_i(q, w)$ means $V_i(\overline{q}w)$.

A prefix-machine $V_0$ is a *universal prefix-machine* if there is an exhaustive enumeration of all prefix-machines $V_1, V_2, \ldots$ such that for all $i \in \mathbb{N}$ and all $q, w, s \in \mathbb{B}^*$ it holds that $V_0(q, i, w) = s$ whenever $V_i(q, w) = s$.[3] The enumeration $V_1, V_2, \ldots$ is called the enumeration associated with $V_0$. The reason for $q$ will be apparent shortly.

The following shows the existence of a universal prefix-machine.

**Proposition 3.4** (Universal prefix-machines)**.** *There is a universal prefix-machine $V_0$.*

*Proof.* $V_0$ can roughly be designed as follows. First $V_0$ reads $q$ and $i$ and stores them on the work-tape. Then it continues to simulate $V_i$ as described by $i$, except that $V_0$ simulates $V_i$ reading symbols from $q$ until $V_i$ has read past $q$. When (if) $V_i$ reads past $q$, then $V_i$ is simulated to read symbols directly from $V_0$'s input tape. □

## 3.3 Existence proofs for prefix-machines

The standard way to show that a certain function is a partial recursive prefix-function is to outline the construction of a prefix-machine computing it. This normally involves devising a prefix code for which it is "clearly decidable" (for a prefix-machine) where a code-word stops in an input-sequence. Thereafter the argument normally proceeds much like the standard argument for the existence of certain Turing-machines: The rest of the computation-procedure is described in a way that makes it clear that it could, in principle, be implemented on a Turing-/prefix-machine.

[2]The enumeration is effective in the sense that given an index $i$, it is clear which prefix-machine is denoted by $V_i$. This is immediate, since the index is a description of how to simulate $V_i$ on $V_0$.

[3]Technically, the universal prefix-machine we have defined here may be called an *additively optimal* prefix-machine. There are (degenerate) universal prefix-machines that for example require the input to appear twice on the input tape (so $V(qq, ii, ww) = V_i(q, w)$) [LV08, Example 2.1.2]. This degenerate kind of universal prefix-machine cannot be used to define Kolmogorov complexity.

### 3.4 Description length

Let $V$ be a prefix-machine. Define for any string $s$ the *(minimum) description length* of $s$ with respect to $V$ as:

$$K_V(s) = \min_w \{\ell(w) : V(w) = s\} \tag{1}$$

The minimum description length is the length of the shortest description—or maximum compression—of $s$ in the code $V$.

In many situations it is natural to ask what the description length of an object is *relative to a description of another object*. For example, the complexity of an image might be high, but if we have a sequence of images (such as in a movie) it can be natural to ask what the complexity of one image is *given the preceding image*. In a movie, the latter quantity is often much smaller. This motivates the more general notion of conditional description length.

Define the *conditional (minimum) description length* of a string $s$ with respect to a prefix-machine $V$ and given information $q$ as

$$K_V(s \,|\, q) = \min_w \{\ell(w) : V(q, w) = s\} \tag{2}$$

That is, the length of the shortest addition $w$ to $q$ such that $V(q, w) = s$. In the case of a movie, $w$ could be a description of how the current image $s$ differs from the preceding image $q$.

Just as when no $q$ was supplied, all prefix-machines $V$ define a prefix code $V(q, \cdot)$ for any fixed $q$.

### 3.5 Additive optimality

Recall Definition 3.1 of additive optimality. Universal prefix-machines define additively optimal codes for the class of effective prefix codes, by the following theorem.

**Theorem 3.5** (Additive optimality)**.** *Let $U$ be a universal prefix-machine and let $q$ be any string. Then $U(q, \cdot)$ describes an additively optimal prefix code.*

*Proof.* Let $C$ be an effective prefix code computed by a prefix-machine $V$. Then there is a prefix-machine $V'$ such that $V'(q, \cdot)$ computes $C$ ($V'$ works like $V$, except that it first reads past $q$). In the enumeration $V_1, V_2, \ldots$ associated with $U$, $V' = V_i$ for some $i$. This means that whenever $w$ is a $C$-code word for a string $s$ (that is, when $V'(q, w) = V(w) = s$), then $U(q, i, w) = s$. Therefore the minimal $U(q, \cdot)$-code word for any string $s$ is at most $\ell(\bar{i}) = 2\ell(i) + 1$ longer than the minimal description length of $s$ with respect to $V$.

The constant $c_V = \ell(\bar{i})$ is sometimes known as the *compiler constant* for $V$. □

## 3.6 Kolmogorov complexity

Having obtained an effective additively optimal prefix code, it is fairly straightforward to define the *Kolmogorov complexity* of a string $s$. The definition uses the length of the shortest code word for $s$ in an additively optimal code.

**Definition 3.6** (Conditional Kolmogorov complexity)**.** Let $U$ to be a particular conditional universal prefix-machine, from now on known as *the reference machine*. When enumerating prefix-machines subsequently, the enumeration will be with respect to $U$. Let the *conditional Kolmogorov complexity* be defined as

$$K(s\,|\,q) = K_U(s\,|\,q) \tag{3}$$

Finally, define $K(s) = K(s\,|\,\epsilon)$ for the *unconditioned Kolmogorov complexity*.

The invariance theorem below show that the choice of conditional universal prefix-machine only has limited impact, and thus that Kolmogorov complexity is an essentially objective notion (see Section 3.9 for further discussion).

**Theorem 3.7** (Invariance theorem)**.** *For any prefix-machine $V$ there is a constant $c_V$ such that for all strings $s$ and $q$*

$$K(s\,|\,q) \leq K_V(s\,|\,q) + c_V$$

*Proof.* By Theorem 3.5, the code $U(q,\cdot)$ is additively optimal. Thus, for any prefix-machine $V$, there exists a constant $c_V$ such that for all strings $s$ and $q$, the shortest $U(q,\cdot)$-code word for $s$ is at most $c_V$ longer than any $V$-code word for $s$. □

*Example* 1 (Kolmogorov complexity). Consider the following two strings. Let $s$ be the string of one million 0's, and let $t$ be a string of one million random 0's and 1's. Then the complexity of $s$ is low, since there is a simple prefix-machine $V$ that on input $\overline{n}$ outputs $10^n$ (ten to the $n$) number of 0's. $V$ has a simple index $i$, and outputs $s$ on input $\overline{6} = 11010$. Therefore the complexity of $s$ is $K(s) \leq 2\ell(i) + 1 + \ell(\overline{6}) = 2\ell(i) + 6$.

For $t$ the situation is the opposite. With high probability there is no simple prefix-machine that outputs $t$ on a short code word, intuitively because there is no structure in $t$ to exploit. Kolmogorov complexity can be seen as a formal measure of structure, with lower complexity corresponding to more structure. ♢

Figure 1 gives a more humorous illustration of compression and Kolmogorov complexity.


Figure 1: *Kolmogorov directions.* An XKCD web-comic (by Randall Munroe, January 2013), depicting a highly (maximally?) compressed set of directions. Fetched from `http://xkcd.com/1155/` on January 28, 2013.

To help intuition, it is useful to consider the *two-part nature* of the code words of $U$. Indeed, the Kolmogorov complexity may equivalently be defined as

$$K(s\,|\,q) = \min_{i,w}\{\ell(i,w) : V_i(q,w) = s\} \tag{3$'$}$$

There is often a tradeoff between on the one hand describing a complex prefix-machine (with complex index $i$) for which a short program suffices to produce $s$, and on the other describing a simple prefix-machine for which a longer program is required to produce $s$. In the following example, which also illustrates conditional Kolmogorov complexity, it is natural to put all information in the index $i$, and no information in $w$.

*Example* 2 (Prefix-complexity). The complexity of a string $s$ given the same string $s$ is

$$K(s\,|\,s) \leq c \tag{4}$$

for some constant $c$ independent of $s$. The intuitive reason is that there is a program $i$ for the machine $U$ that copies any input to the output tape. In more detail, there is a prefix-machine $V_i$ that on any input string $s$ (in the standard prefix-encoding of $s$) outputs $s$. This means that $U(s,i,\epsilon) = V_i(s,\epsilon) = V_i(\overline{s}) = s$, which implies $K(s\,|\,s) \leq 2\ell(i) + 1$ where $i$ is independent of $s$. $\diamondsuit$

### 3.7 Complexity bounds

It is possible to establish an upper bound on the complexity of any string.

**Proposition 3.8** (Maximum complexity)**.** *There exist constants $c_1, c_2 > 0$ such that*

$$K(s\,|\,q) \leq \ell(s) + K(\ell(s)) + c_1 \leq \ell(s) + 2\ell(\ell(s)) + c_2 \tag{5}$$

*for all $s$ and $q$.*

*Proof.* Any string $s$ can be (prefix-)coded by a prefix code for $\ell(s)$ immediately followed by $s$ (this was for example used in the $\overline{s} = 1^{\ell(s)}0s$ code). By using the additively optimal code associated with the reference machine $U$ to code the length of $s$, we can code $\ell(s)$ in $K(\ell(s))$ bits. This motivates the first inequality.

One particular choice of coding for lengths is to encode $\ell(s)$ by $\overline{\ell(s)} = 1^{\ell(\ell(s))}0\ell(s)$. This yields a code where every string $s$ is encoded as $1^{\ell(\ell(s))}0\ell(s)s$. For example, the string $t = 10010$ is encoded as $t' = 1100110010$, since

$$t' = \underbrace{11}_{1^{\ell(\ell(t))}}\ 0\ \underbrace{01}_{\ell(t)=5}\ \underbrace{10010}_{t} \tag{6}$$

In this coding, every string has a code word of length $2\ell(\ell(s)) + \ell(s) + 1$. This code is somewhat longer than the $K(\ell(s))$, but much more efficient than $1^{\ell(s)}0s$. □

By using Kraft's inequality, it is possible to also derive a type of lower bound for Kolmogorov complexity. Although little can be said about the complexity of an arbitrary (single) string, it is possible to say something about the minimum complexity of some collections of strings.

First a definition: We say that a string $s$ is *compressible* if $K(s) < \ell(s)$ and that $s$ is *incompressible* if $K(s) \geq \ell(s)$. Analogously, the notions *compressible-* and *incompressible with respect to $q$* are defined by $K(s\,|\,q) < \ell(s)$ and $K(s\,|\,q) \geq \ell(s)$ respectively.

**Proposition 3.9** (Minimum complexity)**.** *For any given length $n$, at most half of the strings of length $n$ are compressible.*

*Proof.* For any $n$, there are $2^n$ strings $s$ of length $n$. For $s$ to compressible, there must be a code word $w$ of length less than $n$. By Kraft's inequality, in any prefix code there can be at most $2^{n-1}$ code words of length strictly less than $n$. Thus, at most half of all strings of length $n$ can be compressible. □

Note that the bound is not tight for most $n$, as there are some code words that are much shorter than $n - 1$ for most $n$. Also, the proposition is only true for prefix codes. For codes that are not prefix, there are can be up to $2^n - 1$ code words shorter than $n$.

It is straightforward to generalize compressibility to *k-compressibility*. A string $s$ is *k-compressible* if $K(s) < \ell(s) - k$ and *k-incompressible* if it is not $k$-compressible. The corresponding generalisation of Proposition 3.9 then reads: At most $2^{n-k-1}$ strings of length $n$ are $k$-compressible.

The bounds of Proposition 3.8 and 3.9 are often essential tools in establishing properties of Kolmogorov complexity.

## 3.8 Structure and randomness

Incompressibility may also be used to define *randomness*. Essentially, a string is *(Martin-Löf) random* if it is incompressible. As most strings are $k$-incompressible for some small $k$, this shows that most strings are "essentially" random. This corresponds to our intuition that flipping a coin $n$ times yields a random sequence with high probability. Martin-Löf randomness is sometimes called *algorithmic randomness*.

The opposite of randomness is *structure*. The more compressible a string is, the more structured.

## 3.9 Objectiveness

Kolmogorov complexity is often interpreted to quantify the *maximum compression* or *information content* of individual strings.

For example, assume that a reference machine $U$ has been fixed which gives complexity $K(s)$ to some string $s$. Then $K(s)$ can be interpreted as the maximum compressibility of $s$, even though there is always some prefix-machine $V_s$ that assigns an arbitrarily short code word to $s$. From the perspective of $U$ it can be argued that $V_s$ then contains the information $s$, and thus that $K(s)$ is a better measure of complexity than the measure $K_{V_s}(s)$, which is "tailored" to $s$.

But what if $V_s$ is also a universal prefix-machine? Given any string $s$, there exists a *universal* prefix-machine $U_s$ such that $K_{U_s}(s) = 1$. Of course, from the perspective of $U$, the reason is still that $U_s$ "contains" the information $s$, and that $U_s$ is tailored to give $s$ low complexity. But since $U_s$ is a universal prefix-machine, there is no formal reason for why $U_s$ should not have been chosen as reference machine instead of $U$. In which case $U$ would have seemed to give inexplicably high complexity to $s$.

One possible solution is to deem some universal prefix-machines "natural" and to require the reference machine to be chosen among those. For example, it might be argued that a natural reference machine should assign lower complexity to 00000000000000000 than to seemingly random strings such as 100100101101110101. In this view, naturalness must be inherited through *simple simulation*; that is, if $U'$ is natural and there is a short $U'$-description of $U''$, then $U''$ should also be natural. Although imprecise, the

concept of naturalness provides some means for deciding the complexity of particular strings.

Mueller [Mue06] tried to use the idea of simple simulation to find an objective reference machine. Unfortunately it turned out that simple simulation did not yield an objective reference machine, and the attempt failed. So an informal appeal to naturalness remains the only solution for determining the complexity of single strings.

In this thesis, the objectiveness provided by the invariance theorem (Theorem 3.7) will suffice for all results. That is, any universal prefix-machine $U'$ must agree with the chosen reference machine $U$ on the complexity of most strings, in the sense that

$$\exists\, c_{U,U'} : \forall s, q : |K(s\,|\,q) - K_{U'}(s\,|\,q)| \leq c_{U,U'} \tag{7}$$

Much effort has gone into the study of complexity of growing initial segments of infinite sequences, as it pertains to sequence prediction (Section 5.6 below). Asymptotically, any two reference machines agree on the complexity of such initial segments.

# 4 Computability

An important question is whether $K$ is computable. In this section, a hierarchy of computability concepts is presented and the position of $K$ in the hierarchy is determined.

## 4.1 Degrees of computability

If the domain and range of a function $f$ have standard string-encodings (that is, if the domain and are subsets $\mathbb{B}^*$, $\mathbb{N}$ or $\mathbb{Q}$) then $f$ is *recursive* if there is an algorithm computing it.

Some functions $f$ are not recursive, but are still computable in some sense. For example, a function $f$ with the real numbers $\mathbb{R}$ as range can be defined as *computable* by means of a recursive *approximation-function*.

**Definition 4.1** (Computable functions)**.** A function $f : \mathbb{B}^* \to \mathbb{R}$ is *computable* if there is a recursive approximation-function $g : \mathbb{B}^* \times \mathbb{N} \to \mathbb{Q}$ such that for all $s \in \mathbb{B}^*$ and all $k \in \mathbb{N}$, $|g(s,k) - f(s)| \leq 1/k$.

The intuition is that $f$ may be approximated arbitrary well by $g$. Note that computability and recursiveness are equivalent if the range of $f$ is $\mathbb{N}$.

In general, a function $g$ is a recursive *approximation-function* for $f$ if for all $s$, the function $g(s,k)$ approaches $f(s)$ when $k$ goes to infinity. Approximation-functions with different additional requirements are the main tool for defining computability-types weaker than recursiveness.

One such weaker type of computability of interest is *semi-computability*. A semi-computable function also have a recursive approximation-function. The difference is that there is no guarantee for how close the approximation is for any given $k$. Instead, the approximation-function must be monotonically increasing or decreasing.

**Definition 4.2** (Semi-computable functions)**.** A partial function $f : \mathbb{B}^* \rightharpoonup \mathbb{R}$ is *upper semi-computable* if there is a decreasing, recursive approximation-function $g : \mathbb{B}^* \times \mathbb{N} \to \mathbb{Q}$ for $f$. That is, $g$ should be recursive and satisfy: For all $s \in \mathbb{B}^*$, $\lim_{k\to\infty} g(s,k) = f(s)$ whenever $f(s)$ is defined, and $g(s,k) \geq g(s,k+1)$. Further, $f$ is *lower semi-computable* if $-f$ is upper semi-computable, and $f$ is *semi-computable* if at least one of $f$ and $-f$ are upper semi-computable.

Semi-computable functions with range $\mathbb{N}$ are not necessarily recursive, but neither are they entirely incomputable. If a function $f$ is upper semi-computable it is possible to approximate it with a recursive function $g(s,k)$ that approaches $f(s)$ from above, and is identical to $f(s)$ in the limit. So $g(s,k)$ forms a lower bound for $f(s)$ for all $k$, and the bound becomes better with increasing $k$. The problem is that there is generally no guarantee for how tight the bound is. If $g(s,k)$ has been evaluated to 87 for all $k$ smaller than, say, 100'000'000, then $f(s)$ may in fact be 87, but can also be arbitrarily much smaller.

Further weaker notions of computability are also available. For example, if we remove the restriction of the approximation-function $g$ being always decreasing, we get the *approximable* functions. The approximable functions include the semi-computable functions (both the upper and the lower variant), and also some non-semi-computable functions.

## 4.2 Semi-computability of $K$

The complexity function $K(s\,|\,q)$—here treated as a function of $s$ for any fixed $q$—only takes on non-negative integer values. The following two theorems show that $K$ is upper semi-computable, but not computable.

**Theorem 4.3** (Semi-computability of $K$)**.** *$K$ is upper semi-computable.*

*Proof.* The proof constructs a recursive, decreasing approximation-function $g(s,k)$ for $K(s\,|\,q)$.

Let $g(s,k)$ simulate all possible inputs of length at most $2\ell(s)+c$ to $U$ for $k$ steps (the constant $c$ as in Proposition 3.8). When done, $g$ outputs the length of the shortest input that made $U$ produce $s$ in at most $k$ steps. If no input produced $s$ in $k$ steps (a common situation for small $k$), $g$ outputs $2\ell(s)+c$ This is an upper bound on the complexity by Proposition 3.8.

The function $g(s,k)$ is recursive, since it simulates a prefix-machine on a finite number of inputs, each for a finite number of steps. And $g(s,k)$

is clearly decreasing in $k$, since any output that produces $s$ in at most $k$ steps will also produce $s$ in at most $k+1$ steps. Finally, $g(s,k) \to K(s\,|\,q)$ when $k \to \infty$. To see why, assume that $w$ is the shortest input on which $U(q,w) = s$. Then $U(q,w)$ halts in a finite number $m$ of time steps, so for all $k \geq m$ it holds that $g(s,k) = K(s\,|\,q)$. (Unfortunately, there is no general procedure to determine the number $m$.) □

**Theorem 4.4** (Incomputability of $K$)**.** *$K$ is not computable.*

*Proof.* Fix some $q \in \mathbb{B}^*$. Throughout this proof, let $s(n)$ denote the first string of length $n$ (in the lexicographic order) that is incompressible with respect to $q$. Recall that $s$ incompressible with respect to $q$ if $K(s\,|\,q) \geq \ell(s)$, and that there are incompressible strings of all lengths by Proposition 3.9.

Assume that $K(s\,|\,q)$ were computable; that is, that there were a program computing $K(s\,|\,q)$ on any input $s$. Building on this program, it would be easy to construct a prefix-machine $V_i$ such that $V_i(q,\overline{n}) = s(n)$ for all $n$.

This leads to a contradiction. For any $n$, the reference machine $U(q,i,\overline{n})$ would return $s(n)$, so all $s(n)$ would have complexity at most $2\ell(i)+2\ell(n)+2$. This would imply

$$n \leq K(s(n)\,|\,q) \leq 2\ell(i) + 2\ell(n) + 2 \tag{8}$$
$$\leq 2\log_2(i+1) + 2\log_2(n+1) + 2 \tag{9}$$

for all $n$, which is a contradiction for sufficiently large $n$ ($i$ remains fixed). In other words, an incompressible string would be compressible. □

In conclusion, although the Kolmogorov complexity is not computable, it can still be approximated in the semi-computability sense.

# 5 Measures and induction

Inductive reasoning is the process of making uncertain but justified inferences; often the goal is to infer a general theory from particular observations. For example, according to the famous anecdote, Newton discovered gravity when seeing an apple fall from a tree. (Presumably, he also recalled a large number of other (particular) examples of things falling or "attracting" each other in space).

Inductive inference is a central tool in science. One of the most important induction principles is Occam's razor, which may be interpreted as "given several possible explanations for an observed phenomenon, the *simplest* explanation should be preferred". The problem is that it is (i) often unclear which explanation is simpler, and (ii) unclear to what extent a simpler theory should be preferred to a more complicated theory if the more complicated theory gives a more exact explanation.

Given the right setup, Kolmogorov complexity can be used as a formalization of the vague term *simple*. Kolmogorov complexity thus offers a formal solution to (i). Further, Kolmogorov complexity can be used to construct a prior, which together with Bayes' rule offers a convincing solution to (ii). Kolmogorov complexity can thus be used as a basis of a formal theory of scientific induction [RH11].

First we will review some general measure theory and construct measure spaces for $\mathbb{B}^*$ and $\mathbb{B}^\infty$. For these spaces, two measures (priors) $\mathbf{m}$ and $\mathbf{M}$ are constructed in accordance with Occam's razor. We then give an account of how $\mathbf{M}$ can be used for induction (sequence prediction) and recount a strong result by Solomonoff [Sol78] that shows the strong inductive performance of $\mathbf{M}$.

## 5.1 Definition of measure

*Measure theory* formalizes probability theory. Here we will only briefly recount the most important definitions, for a more complete overview we refer to any standard textbook on formal probability theory (for instance [Res98]).

**Definition 5.1** ($\sigma$-algebra)**.** A *$\sigma$-algebra* on a sample space $\Omega$ is a collection $\Sigma$ of subsets of of $\Omega$ satisfying:

- $\Sigma$ contains $\Omega$.
- $\Sigma$ is closed under countable union and complementation. That is, if $A \in \Sigma$, then $A^c = \Omega - A \in \Sigma$; and if $\{A_i\}_{i\in I}$ is a countable collection of elements of $\Sigma$, then $\bigcup_{i\in I} A_i \in \Sigma$.

The elements of $\Sigma$ are called *measurable sets* or *events*.

Note that since $\Sigma$ contains $\Omega$ and is closed under complementation, it also includes the empty set $\emptyset = \Omega^c$. $\Sigma$ must also be closed under countable intersection, since $\bigcap_{i\in I} A_i = \left(\bigcup_{i\in I} A_i^c\right)^c$.

A *measure space* on a space $\Omega$ is a pair $(\Sigma, \Omega)$ where $\Sigma$ is a $\sigma$-algebra on $\Omega$.

**Definition 5.2** (Measure)**.** Given a measure space $(\Sigma, \Omega)$, a function $\lambda : \Sigma \to [0, 1]$ is a *measure*[4] on $(\Sigma, \Omega)$ if it satisfies:

- $\lambda(\Omega) = 1$,
- $\lambda(\bigcup_{i\in I} A_i) = \sum_{i\in I} \lambda(A_i)$ for any countable collection $\{A_i\}_{i\in I}$ of pairwise disjoint elements of $\Sigma$.

[4]In the measure-theory literature, a more general version of measure that can take on any non-negative real number and $+\infty$ is often considered. In such contexts, our version of measure with $\lambda(\Omega) = 1$ is often called a *probability measure*.

An important consequence is that $\lambda(\emptyset) = 0$. This follows, since $\lambda(\Omega) = \lambda(\Omega \bigcup \emptyset) = \lambda(\Omega) + \lambda(\emptyset)$. By subtracting $\lambda(\Omega)$ from both sides, $\lambda(\emptyset) = 0$ is established.

When $\Omega$ is countable, the standard choice of $\sigma$-algebra is the *power-set* $2^\Omega = \{A : A \subseteq \Omega\}$ of $\Omega$. However, when $\Omega$ is infinite, it is often hard (or even impossible) to obtain a measure on all subsets. Some sets are *immeasurable* in the sense that no "natural" measure can assign a value to them.[5]

We will often use a slightly weaker version of measure, called *semi-measure*.

**Definition 5.3** (Semi-measure)**.** A *semi-measure* on a measure space $(\Sigma, \Omega)$ is a function $\lambda : \Sigma \to [0, 1]$ satisfying

- $\lambda(\Omega) \leq 1$,
- $\lambda(\bigcup_{i \in I} A_i) \geq \sum_{i \in I} \lambda(A_i)$ for any collection $\{A_i\}_{i \in I}$ of pairwise disjoint elements of $\Sigma$.

The difference between measures and semi-measures is that the *full event* $\Omega$ only needs to have measure *at most* 1, and that the union of disjoint events may have a larger measure than the sum of the parts. Note that the inequalities are set in a way so that semi-measures must assign the empty set measure 0.

## 5.2 Measure spaces on $\mathbb{B}^*$ and $\mathbb{B}^\infty$

We will now construct measure spaces on the set of strings $\mathbb{B}^*$ and the set of one-way infinite sequences $\mathbb{B}^\infty$. These measure spaces will be the only measure spaces we will use this section.

For the measure space $\mathbb{B}^*$ we will simply use the "maximal" $\sigma$-algebra $2^{\mathbb{B}^*}$. So the measure space on $\mathbb{B}^*$ becomes $(2^{\mathbb{B}^*}, \mathbb{B}^*)$.

For the measure space on $\mathbb{B}^\infty$, some further notation needs to be developed. Define for any string $s$ the *cylinder* $\Gamma_s = \{sz : z \in \mathbb{B}^\infty\}$. The cylinders are subsets of $\mathbb{B}^\infty$, but do not form a $\sigma$-algebra. To obtain a $\sigma$-algebra on $\mathbb{B}^\infty$, let $\Psi$ be the *$\sigma$-closure* of the set of all cylinders. That is, let $\Psi$ be the set of all $A \subseteq \mathbb{B}^\infty$ that can be obtained from any collection of cylinders by means of (repeated application of) countable union and complementation. The $\sigma$-algebra $\Psi$ is sometimes called the *Borel* $\sigma$-algebra. The measure space we will use for $\mathbb{B}^\infty$ is $(\Psi, \mathbb{B}^\infty)$.

For brevity, we will sometimes write $\mathbb{B}^*$ for $(2^{\mathbb{B}^*}, \mathbb{B}^*)$ and $\mathbb{B}^\infty$ for $(\Psi, \mathbb{B}^\infty)$, keeping in mind which measure spaces are actually intended.

[5] Such immeasurable sets include the so-called *Vitali sets*, see for instance [Fri70].

## 5.3 Measure conventions

First adopt the abbreviations $\lambda(s) = \lambda(\{s\})$ for the singleton events of $(P(\mathbb{B}^*), \mathbb{B}^*)$ and $\nu(s) = \nu(\Gamma_s)$ for the cylinder sets of $(\Psi, \mathbb{B}^\infty)$. Further, all semi-measures are extended with provided information $q \in \mathbb{B}^*$, in semblance to conditional Kolmogorov complexity. Every semi-measure $\lambda$ is thus extended to a class of measures $\lambda_q$. The provided information $q$ is sometimes useful when studying induction.

**Definition 5.4** (Computable measure)**.** We say that a (class of) measure(s) $\lambda$ on $(2^{\mathbb{B}^*}, \mathbb{B}^*)$ is *computable* if there is a computable function $g_\lambda$ that satisfies $g_\lambda(s, q) = \lambda_q(s)$, and that $\lambda$ is *lower semi-computable* if $g_\lambda$ is lower semi-computable. Similarly, a measure $\nu$ on $(\Psi, \mathbb{B}^\infty)$ is *(lower semi-)computable* if there is a (lower semi-)computable function $g_\nu$ such that $g_\nu(s, q) = \nu_q(\Gamma_s)$ for all strings $s$ and $q$.

## 5.4 Measures on $\mathbb{B}^*$

The uniform measure on $(2^{\mathbb{B}^*}, \mathbb{B}^*)$ is the *discrete Lebesgue-measure* $\mu$, defined by $\mu(s) = 2^{-2\ell(s)-1}$ on the singleton events $\{s\}$ for all $s \in \mathbb{B}^*$. (The measure $\mu$ simply ignores provided information $q$, so $\mu = \mu_q$ for all $q$.) Defining a measure on the singleton events uniquely determines it on all other subsets of $\mathbb{B}^*$, by the axioms of a probability measure.

A *universal* semi-measure $\mathbf{m}$ for $\mathbb{B}^*$ can be defined as follows.

**Definition 5.5** (The discrete universal distribution)**.** Let for every $s, q \in \mathbb{B}^*$

$$\mathbf{m}_q(s) = 2^{-K(s\,|\,q)} \tag{10}$$

The semi-measure $\mathbf{m}$ is called the *discrete universal distribution.* It agrees with Occam's razor in assigning higher probability to strings with low complexity.

That $\mathbf{m}$ is a semi-measure (sums to at most 1) follows from Kraft's inequality (Theorem 2.3 on page 12). Kraft's inequality gives that $\sum_{w \in C} 2^{-\ell(w)} \leq 1$ for any set $C$ of code words in a prefix code. As the reference machine defines a prefix code, it follows that $\sum_{s \in \mathbb{B}^*} \mathbf{m}_q(s) \leq 1$ for all strings $q$. As not all programs are a shortest code words for some string, the summation will in fact be strictly less than 1. Therefore, $\mathbf{m}$ will only be a semi-measure and not a measure.

### 5.4.1 Dominance of m

An important property of $\mathbf{m}$ is that it *dominates* all semi-computable semi-measures on $(2^{\mathbb{B}^*}, \mathbb{B}^*)$.

**Definition 5.6** (Dominance of measures)**.** Let $\rho$ and $\nu$ be two semi-measures on some measure space $(\Sigma, \Omega)$. If there is a constant $c > 0$ such that $\rho(A) \geq c \cdot \nu(A)$ for all events $A \in \Sigma$, then we say that $\rho$ *dominates* $\nu$ with the constant $c$. Similarly, $\rho$ *dominates* a class $\mathcal{M}$ of measures on $(\Sigma, \Omega)$ if $\rho$ dominates each element of $\mathcal{M}$.

The following discussion explains why $\mathbf{m}$ dominates all semi-measures.

There is an effective enumeration $\lambda^1, \lambda^2, \ldots$ of all semi-computable semi-measures on $(2^{\mathbb{B}^*}, \mathbb{B}^*)$ [LV08, p. 267]. Essentially, the index $i$ in $\lambda^i$ represents a code for a program (semi-)computing $\lambda^i_q(s)$. Fixing one such enumeration/reference machine, it is natural to extend the definition of Kolmogorov complexity to the lower semi-computable semi-measures on $(2^{\mathbb{B}^*}, \mathbb{B}^*)$ by $K(\lambda) = \min_i\{K(i) : \lambda = \lambda^i\}$.

Semi-measures $\lambda^i$ that are "simple" (have short descriptions) will receive simple indexes $i$ and therefore high weight, whereas complicated, arbitrary semi-measures will receive complex indexes. Examples of fairly simple measures include $\mu$ and $\mathbf{m}$ since they have comparatively simple descriptions.

The dominance of $\mathbf{m}$ over all semi-computable measure follows from that

$$\mathbf{m}_q(s) = \sum_{i \in \mathbb{N}} 2^{-K(i)} \lambda^i_q(s) \tag{11}$$

holds up to a multiplicative constant [LV08, Theorem 4.3.3]. This immediately gives that $\mathbf{m}_q(s) \geq 2^{-K(i)}\lambda^i_q(s)$ for any semi-computable semi-measure $\lambda^i$ on $(2^{\mathbb{B}^*}, \mathbb{B}^*)$. We state this as a proposition for future reference.

**Proposition 5.7** (Dominance of $\mathbf{m}$)**.** *The discrete universal measure* $\mathbf{m}$ *dominates every lower semi-computable semi-measure* $\lambda$ *with a constant* $2^{-K(\lambda)}$.

Dominance is one reason for using semi-measures rather than measures. It can be shown that no computable measure dominates all other computable measures [LV08, Lemma 4.3.1]. Meanwhile, $\mathbf{m}$ is a lower semi-computable semi-measure (since $K$ is upper semi-computable) and $\mathbf{m}$ dominates all lower semi-computable semi-measures.

## 5.5 Measures on $\mathbb{B}^\infty$

The uniform distribution on $\mathbb{B}^\infty$ is the *continuous Lebesgue-measure* $L(s) = 2^{-\ell(s)}$. The important difference between the discrete case and the continuous case is that in the continuous case, the event of a short string $s$ contains all extensions of $s$. In the discrete case, any extension of $s$ is a separate event.

There is a continuous universal distribution $\mathbf{M}$ for $(\Psi, \mathbb{B}^\infty)$. In analogy to the prefix-machines, there is a type of machine called *monotone machines* which may be used to define $\mathbf{M}$. Rather than defining monotone machines,

however, we take a shortcut and define $\mathbf{M}$ via an enumeration $\nu^1, \nu^2, \ldots$ of all semi-computable semi-measures [LV08, p. 295]. Let $K(\nu) = \min_i\{K(i) : \nu = \nu^i\}$.

**Definition 5.8** (The continuous universal distribution)**.** Let the *continuous universal distribution* $\mathbf{M}$ be defined as[6]

$$\mathbf{M}(s) = \sum_{i\in\mathbb{N}} 2^{-K(i)}\nu^i(s) \tag{12}$$

The properties of $\mathbf{M}$ mirrors to a large extent the properties of $\mathbf{m}$. The semi-measure $\mathbf{M}$ trivially dominates all lower semi-computable semi-measures $\nu$ on $(\Psi, \mathbb{B}^\infty)$ with a constant $2^{-K(\nu)}$.

Further, $\mathbf{M}$ is lower semi-computable: The value $2^{-K(i)}\nu^i(s)$ is lower semi-computable for all lower semi-computable semi-measures $\nu$. Therefore, a sum $\mathbf{M}(s)$ can be lower semi-computed by lower semi-computing an increasing number of terms to increasing accuracy.

It can be shown that $\mathbf{M}$ assigns higher weight to simple initial segments $s$; in fact, $\mathbf{M}(s) \approx \mathbf{m}(s)$ so $\mathbf{M}(s) \approx 2^{-K(s)}$ for all strings $s$ (both approximations are up to logarithmic factors in the length of $s$). Thus $\mathbf{M}$ agrees with Occam's razor in assigning higher probability to "simple" events.

## 5.6 M and sequence prediction: Solomonoff induction

Sequence prediction is a rather general induction setting. For example, it can model a scientist making repeated experiments, or the development of the weather. Formally, in the setting of sequence prediction an infinite sequence $z$ has been generated according to some distribution $\nu$ on $(\Psi, \mathbb{B}^\infty)$. The task is to (repeatedly) guess the next bit $z_{n+1}$ for growing initial segments $z_{1:n}$.

Assume, for instance, that we are trying to predict the weather based on previous meteorological observations. Let rain be encoded as 0 and sunshine as 1, and let the task be to predict the weather (sun or rain) the next day given a string $z_{1:n}$ representing the weather of previous days.

The weather is presumably described by some computable distribution $\nu$. This means that the best prediction $s$ of $z_{n+1}$ (the weather tomorrow) would be the prediction given by (the Bayesian $\nu$-posterior) $\nu(z_{n+1} = s|z_{1:n1}) = \nu(z_{1:n}s)/\nu(z_{1:n})$. Unfortunately, the true distribution $\nu$ of how the weather develops is unknown. Solomonoff's idea was that $\nu$ could be replaced with an *inductive prior* $\rho$ that would converge to the true distribution $\nu$, given that $\nu$ came from some (preferably large) class of distributions.

To be able to quantify how well a certain prior $\rho$ performs on sequence prediction, we need a formal benchmark for how well $\rho$ manages to predict sequences generated from a true measure $\nu$. One interesting benchmark is based on the *$\nu$-expected prediction-distance.*

[6]For technical reasons it is standard to include provided information to $\mathbf{m}$ but not to $\mathbf{M}$.

**Definition 5.9** (Prediction-distance)**.** Define the *$\nu$-expected prediction-distance* of prediction $n$ between a semi-measure $\rho$ and a measure $\nu$ as

$$D_n(\rho,\nu) = \sum_{s\in\mathbb{B}} \sum_{z_{1:n}\in\mathbb{B}^n} \nu(z_{1:n}) \left(\sqrt{\rho(z_{n+1}=s\,|\,z_{1:n})} - \sqrt{\nu(z_{n+1}=s\,|\,z_{1:n})}\right)^2$$

That is, the $\nu$-expected value of the distance $\left(\sqrt{\rho(z_{n+1}=s\,|\,z_{1:n})} - \sqrt{\nu(z_{n+1}=s\,|\,z_{1:n})}\right)^2$, summed over all next possible bits $s$.

A few remarks can be made about this benchmark. First, the expectation is taken with respect to the true measure $\nu$. The effect is that it is more valuable to do well on likely sequences than unlikely sequences. Second, one might question why the "complicated" prediction-distance is used rather than a simple comparison of whether both predicted the same bit (1 or 0) as the most likely next bit. Such a comparison could indeed have been interesting. However, we are often interested in the *probability* of an event rather than only the most likely outcome. Say, for instance, that there is a 45% chance of rain tomorrow. Then we are interested in knowing that (so we can take our umbrella in case), rather than only knowing that the chance of sun is higher than the chance of rain. Such "decision-theoretic" aspects are better captured by the prediction-distance.

Solomonoff discovered that dominance was an important feature for an inductive prior. The following can be proven about prediction-distance for a prior semi-measure $\rho$ dominating another measure $\nu$ (see [LV08, Section 5.2] for a proof[7]).

**Theorem 5.10** (Solomonoff induction)**.** *Let $\rho$ be a semi-measure and $\nu$ be a measure, and let $\rho$ dominate $\nu$ with a constant $c$. Then the* total *$\nu$-expected prediction-distance between $\rho$ and $\nu$ is finite and bounded by* $\ln(1/c)$*:*

$$\sum_{n\in\mathbb{N}} D_n(\rho,\nu) \leq \ln(1/c) \tag{13}$$

Since $\mathbf{M}$ dominates all computable measures $\nu$ with a constant $2^{-K(\nu)}$ by definition, we immediately obtain the following corollary.

**Theorem 5.11** (Solomonoff induction)**.** *For all computable measures $\nu$, the total $\nu$-expected prediction-distance between $\mathbf{M}$ and $\nu$ is finite and bounded by $K(\nu)$:*

$$\sum_{n\in\mathbb{N}} D_n(\mathbf{M},\nu) \leq K(\nu)\ln(2) \tag{14}$$

[7] More precisely, the theorem is a special case of what is proven in Example 5.2.5 on page 356 in [LV08].

This shows that $\mathbf{M}$ is an excellent prior, given that the true distribution is computable (which, arguably, is a quite reasonable assumption in most practical cases). For an in-depth investigation of the philosophical aspects of universal induction, see [RH11].

Since induction with $\mathbf{M}$ works for any (lower semi-)computable (semi-)measure it is often called *universal induction*. Other names include *Solomonoff induction*, recognizing its inventor Ray Solomonoff [Sol64a, Sol64b, Sol78].

### 5.6.1 Induction with a uniform prior

It is instructive to compare the inductive performance of $\mathbf{M}$ and the uniform measure $L$.

Under a uniform prior $L$, the next bit is 0 or 1 with equal probability, given any initial sequence $z_{1:n}$. To illustrate why this is undesirable for induction we will consider two examples, one where the sequence is random (uniformly distributed) and one where the sequence is structured.

Consider first the case where $z$ is sampled from a deterministic environment $\nu$ which give probability 1 to $101010\ldots$ and probability 0 to all other sequences. Then $\mathbf{M}$ will quickly converge to $\nu$, and correctly predict the next bit with high probability. This follows from Theorem 5.11 since $\nu$ is computable and has low complexity. The uniform prior $L$, on the other hand, will after every initial segment predict that 0 is as likely as 1. $\mathbf{M}$ thus clearly outperforms $L$ in this case, as was expected.

But what if $z$ was sampled from the uniform distribution $L$? Then the prior $L$ would correctly predict that both 0 and 1 were equally likely next bits. But since $L$ is a computable distribution (with low complexity), $\mathbf{M}$ would quickly converge to predicting that 0 and 1 were equally likely as well.

These two examples give a sense of how a uniform prior is biased towards randomness. No matter how long initial segment is highly structured, the uniform prior insists that 0 and 1 are equally likely next bits. The universal prior $\mathbf{M}$, on the other hand, adapts. If the initial segment is structured, it predicts that the structure is likely to remain. If no structure is present in the initial segment, than the next bit is probably randomly sampled. In this sense, a universal prior is *less biased* than a uniform prior.

## 5.7 Other induction settings

Not all induction problems can be (naturally) cast as sequence-prediction tasks. For example, in supervised machine learning (see for instance [Mar09]), a finite number of predictions should be made based on a finite number of learning examples. A related problem is the optimisation of a function over a finite domain, which will be the focus in Part II. In these finite cases, it is sometimes more natural to use the discrete universal mea-

sure **m**. (Philosophically it does not make much difference, as both measures dominate their respective measure classes, and **m** and **M** differ from each other by at most a logarithmic factor in the string-argument).

Fewer results have been obtained in the finite settings. One of the major aims of Part II will be to develop an understanding of the usefulness of **m** in function optimisation over a finite domain.

### 5.8 AIXI and universal intelligence

Another setting is a generalization of sequence prediction to the *active case*. In this setting, the goal is not only to predict the next symbol, but to interact with an environment. That is, at every time step, the *agent* makes an *action* (0 or 1), and the *environment* (the perceived input sequence) reacts to it. Perhaps surprisingly, this makes the problem significantly harder. A well-developed theory called AIXI exists for this setting, see [Hut05].

AIXI arguably constitutes an *optimal, universal intelligent agent* (disregarding computational aspects) [Hut05]. Thanks to the semi-computability of Solomonoff induction, AIXI can be computably approximated. In a certain sense, the approximation performs as well as any other computable agent with the same computation power, save for rather big multiplicative constants [Hut05, Section 7.2.7].

## 6 Summary of Part I

In this part we have given an introduction to a formal, essentially objective complexity measure $K$ for strings, called *Kolmogorov complexity*. We used $K$ to construct two "universal" measures **m** and **M** with a bias towards simplicity (in the Kolmogorov-complexity sense). A result on **M**'s *total prediction-distance* to any computable measure $\nu$ was formulated. A natural interpretation of this result is that **M** is an excellent prior for sequence prediction.

# Part II
# No Free Lunch and Optimisation

The problem of optimisation arises in many situations. For example, the development of a medicine can naturally be understood as an optimisation problem where the optimal combination of chemicals is sought. The Travelling Salesman Problem can be seen as an optimisation problem where the goal is to find a maximally short route. In nature, one can understand evolution as an optimisation process, striving to find as good gene combinations as possible.

Optimisation can be formalized as: An unknown function $f : X \to Y$ is probed at particular points $x_i \in X$ and the value $f(x_i)$ is perceived; the aim being to quickly find high/maximum values of $f$. As points of $f$ are probed, the knowledge of $f$ grows, and better guesses can be made. Most optimisation problems can (more or less naturally) be formalized this way. In the medicine example, the function $f$ could be the difference between the fraction of patients cured minus the number of side effects. For evolution, $f$ could be defined as the amount of off-spring a certain gene-combination yields (in a certain environment).

The aspect we will be interested in is what prior information one needs about $f$ in order to optimise $f$ efficiently. Typically, informed guesses must be made about $f$'s global behaviour from a set of particular examples (the probed points). From this "information-theoretic" perspective, optimisation includes an induction problem.

The No Free Lunch (NFL) theorems show that under some priors, efficient optimisation is not possible. Intuitively, if all functions are equally likely, then no meaningful extrapolation from any acquired knowledge is possible; therefore all algorithms will perform equally badly [WM97]. When a prior P makes all algorithms perform equally well, we say that *NFL holds* for P. One of the main results so far has been a precise characterization of priors for which NFL holds, known as the Non-uniform NFL theorem [IT04] (see Section 8 and 9.3 for further discussion).

Note however that not all optimisation problems are interesting to study from an information-theoretic perspective. In the Travelling Salesman Problem, for instance, the challenge is not lack of information—all distances are known—but the computational complexity of the problem.

The universal distribution provides a universal prior, which in the case of sequence prediction can be used whenever the true environment is computable (see Section 5.6 above). This thesis will culminate in a characterization of whether the universal distribution can be used also for optimisation (Section 11 and 12), improving on the current knowledge in several directions.

Other contributions include a precise derivation of the probabilistic

framework underlying most NFL-studies (Section 7.5); a direct proof of a generalization of the Non-uniform NFL theorem (Theorem 9.9); a continuity theorem for NFL (Theorem 9.11); and a discussion on particular performance measures (Section 10).

We start by making the problem description precise (Section 7). We then give an overview of the NFL research done so far, chiefly during the last 15 years (Section 8). The most important results will be given detailed accounts in Section 9. Our main results then appear in sections 9.4–12, together with an outlook and some concluding remarks in Section 13.

# 7 Preliminaries

In this section we make the setting formally precise, and introduce most of the required notation.

## 7.1 Search problems

Let $X$ and $Y$ be two finite ordered spaces containing at least two elements each. $X$ will be known as the *search space* or the *domain*, and $Y$ as the *range* or the *co-domain*. Together they will be known as the *problem context*. Lower case $x$ and $y$, sometimes with sub- or superscripts, will denote elements of $X$ and $Y$ respectively.

A *(black-box optimisation) problem*[8] is a probability measure P over the finite set $Y^X = \{f : X \to Y\}$ of functions from $X$ to $Y$. $\mathrm{P}(f) = \mathrm{P}(\{f\})$ should be interpreted as the prior belief in $f$ being the *true function*. Formally, P is a measure on the measure space $(\Sigma, \Omega)$ with $\Omega = \{f : X \to Y\}$ and $\Sigma = 2^{\{f:X\to Y\}}$ the power-set of $\Omega$. The measure P will sometimes also be referred to as the *prior*.

As the function is probed at some point $x_i$, its behaviour $y_i$ in the probed point will become known. Let $\{(x_i, y_i) : x_i \text{ has been probed}\}$ be the knowledge acquired of $f$. The associated event $A = \{f : f(x_i) = y_i)\}$ can be used to update the prior belief in $f$, $\mathrm{P}(f)$, to a posterior belief $\mathrm{P}(f \,|\, A) = \mathrm{P}(f, A)/\mathrm{P}(A)$.

From P we can also derive the probability of events such as "$f(x) = y$" for a certain point $x$. Let $B = \{f : f(x) = y\}$. Then $\mathrm{P}(f(x){=}y) = \mathrm{P}(B)$. It is also straightforward to ask for the probability of "$f(x) = y$" given some knowledge $A$. The letters $A$ and $B$ will denote function events. Often (but not always), function events have an associated set of $(x, y)$-pairs as was the case with the acquired knowledge above.

[8] Sometimes known as a Black-box Search problem, or Optimisation with Randomized Heuristics [DJW02].

## 7.2 Algorithms

Only knowing the problem context $X$,$Y$ and the prior P, a search algorithm $a$ should try different values $x \in X$, and, depending on the $y$-values returned, try new points in $X$. Often the goal is to find a certain type of value (such as the maximum) as soon as possible, but formal investigations often allow a wider range of performance measures (see Definition 9.1 below, and Section 10 for a discussion). Since the algorithm only has access to a distribution P (and not to the true function), the goal for the algorithm will be to do well in P-expectation.

### 7.2.1 Search traces

To define search algorithms formally, some further notation need to be developed. A search trace $T_n$ is an ordered $n$-tuple $\langle (x_1, y_1), \ldots, (x_n, y_n) \rangle \in (X \times Y)^n$, representing a search history. $T$ and $S$ will be used as names for traces. The empty trace will be denoted by $\langle\rangle$.

Let $\mathcal{T}_n$ be the set of all search traces of length $n$, and let $\mathcal{T} = \bigcup_{i=0}^{|X|} \mathcal{T}_i$ be the set of traces of any length. If $T_n = \langle (x_1, y_1), \ldots, (x_n, y_n) \rangle$, let $T_n^x = \langle x_1, \ldots, x_n \rangle$ be the corresponding *probing sequence* and $T_n^y = \langle y_1, \ldots, y_n \rangle$ the corresponding *result vector*. Analogously, let $\mathcal{T}_n^x$ be the set of all probing sequences of length $n$, and $\mathcal{T}_n^y$ be the set of all result vectors of length $n$. Finally, let $\mathcal{T}^x$ and $\mathcal{T}^y$ denote the set of probing sequences and result vectors of any length.

Elements of $\mathcal{T}_{|X|}$, $\mathcal{T}_{|X|}^x$ and $\mathcal{T}_{|X|}^y$ will be called *full-length traces*, *full-length probing sequences* and *full-length result vectors* respectively.

### 7.2.2 Deterministic search algorithms

A deterministic search algorithm $a$ can now be modelled as a function

$$a : S \in \mathcal{T} \mapsto x \in X - S^x \tag{15}$$

from search traces to new points to probe. The *no-revisiting* condition $a(S) \notin S^x$ ensures that no search algorithm searches the same point twice, a restriction commonly used in the literature (see [WM97] for a discussion). Let $\mathcal{A}$ be the set of all deterministic algorithms.

A deterministic search procedure $a$ encountering a function $f$ thus generates traces

$$T_n(a, f) = \langle (x_1, y_1), \ldots, (x_n, y_n) \rangle \in (X \times Y)^n \tag{16}$$

where each $(x_{i+1}, y_{i+1}) = a\langle (x_1, y_1), \ldots, (x_i, y_i) \rangle$. Let $T.(x, y)$ be the trace $T$ appended with $(x, y)$.

*Example* 3 (Trace-generation). Assume that the true function $f$ is always 4 on the search space $X = \{0, 1, 2\}$, and assume that $a$ always searches $X$ from left to right. Then $a$ will start searching at 0; that is, $a(\langle\rangle) = 0$. The perceived value will be 4, so the search trace becomes $T_1(a, f) = \langle(0, 4)\rangle$. Continuing, $a(\langle(0, 4)\rangle) = 1$. That is, $a$ searches at 1 in the second step, and will again perceive value 4. This yields the search trace $T_2(a, f) = \langle(0, 4), (1, 4)\rangle$. And so on. ♢

### 7.2.3 Probabilistic algorithms

Many general search algorithms are not deterministic. Instead the next search point is (sometimes) chosen stochastically. It is, for example, common to let hill-climbing algorithms restart at a random point once they reach a local optima. A probabilistic algorithm $b$ can be modeled by conditional probability distributions $\Pr_b(\cdot\,|\,T)$ over next search points for all $T \in \mathcal{T}$. The expression $\Pr_b(x\,|\,T)$ denotes the probability that $b$ continues the search at $x$ after the trace $T$. The no-revisiting condition for probabilistic algorithms states that for any $T$ and any $x \in T^x$, $\Pr_b(x\,|\,T) = 0$; in other words, $\Pr_b(\cdot\,|\,T)$ should have no support outside $X - T^x$.

Note that deterministic algorithms are special cases probabilistic algorithms. A deterministic algorithm $a$ is a probabilistic algorithm for which the conditional distributions $\Pr_a(\cdot\,|\,T)$ have range $\{0, 1\}$ for all traces $T$.

In fact, probabilistic algorithms may equally well be modeled by algorithms that are only stochastic prior to the search phase. Such algorithms that probabilistically chooses a deterministic algorithm $b$ before the search phase and then deterministically searches like $b$ will be called *pre-sampling* algorithms.

**Lemma 7.1** (Pre-sampling algorithms)**.** *For every probabilistic algorithm $b$ there is an equivalent pre-sampling algorithm $b'$ in the sense that for every $T \in \mathcal{T}$ and $x \in X$, it holds that $\Pr_b(x\,|\,T) = \Pr_{b'}(x\,|\,T)$.*

*Proof.* Let $b$ be a probabilistic algorithm. For each $T \in \mathcal{T}$, independently sample a search point $x \in X$ according to the conditional probability $\Pr_b(x\,|\,T)$. Denote the full sample with a deterministic function (algorithm) $a : \mathcal{T} \to X$.

The probability for a given sample $a$ is then $\Pr_b(a) = \prod_{T\in\mathcal{T}} \Pr_b(x\,|\,T)$. Further, for any $T \in \mathcal{T}$ and any $x \in X$, it holds that $\Pr_b(x\,|\,T) = \sum_{a:a(T)=x} \Pr_b(a)$.

Let $b'$ sample deterministic algorithms according to $\Pr_b(a)$. Then $\Pr_{b'}(x\,|\,T) = \sum_{a:a(T)=x} \Pr_b(a) = \Pr_b(x\,|\,T)$, which completes the proof. □

Let $[[r_1 = r_2]]$ be the *truth function* that is 1 if $r_1 = r_2$ and 0 otherwise. The following is an immediate corollary of Lemma 7.1.

**Proposition 7.2.** *The probability that a probabilistic algorithm b generates a trace $T \in \mathcal{T}_n$ on a function $f$ is*

$$\Pr_b(T \mid f) = \sum_{a \in \mathcal{A}} \Pr_b(a)[[T_n(a, f) = T]] \tag{17}$$

Proposition 7.2 will be useful in constructing a measure on traces (Section 7.5) and in generalizing results on deterministic algorithms to probabilistic ones.

## 7.3 An example

We now give a longer example illustrating our setup. Readers who already feel confident with the setup may safely skip this example.

*Example* 4. Assume that $X = \{-2.0, -1.9, \ldots, 0, \ldots, 1.9, 2\}$, and that the class $F$ of functions is the polynomials $f(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_5 x^5$ of degree at most five with bounded integer coefficients $c_i \in \{0, \pm 1, \ldots, \pm 10\}$. There are then 21 constant polynomials (degree 0), $21^2$ polynomials of degree 1 and so on. Denote the number of polynomials with the same degree as a polynomial $f$ with $\#\deg(f)$. Let the distribution be $\mathrm{P}(f) = 1/(6\#\deg(f))$. Then the summed probability of all polynomials of a certain degree is $1/6$, and since there are six different degrees (0–5) the probability sums to 1. The probability of a single "simple" (i.e., low-degree) polynomial is higher than a "complex", high-degree polynomial, however, since there are fewer polynomials of low degrees. Let the range be determined implicitly by $Y = \{y : \exists x \in X, f \in F : f(x) = y\}$.

The algorithm should now try values in $X$ in order to find a maximum of the sampled function as quickly as possible. Assume that the true function is $f(x) = -x^4 + 4x^2 - 2x - 1$ (the algorithm does not receive this information, of course; the algorithm only has access to the distribution P).

Assume the algorithm starts with trying $0 \in X$. Then it will receive $f(0) = -1$ in return; the search trace becomes $T_1 = \langle (0, -1) \rangle$. This provides a limited amount of information, so the algorithm may (arbitrarily) try 1 at the next probe, which gives $f(1) = 0$. The trace becomes $T_2 = \langle (0, -1), (1, 0) \rangle$. The polynomial with the highest sampling-probability that is consistent with $T_2$ is $p_1(x) = x - 1$. This indicates that 2 should be the maximum. Trying that, however, it receives $f(2) = -5$ (which is inconsistent with the conjectured polynomial $x - 1$). The algorithm thus needs to revise its model, and make a new attempt. And so on.

Figure 2 illustrates the situation after the second probe.

Substantial effort has been put into the problem of how to best optimise a function in different settings. In the setting of this example, *Lagrange interpolation* offers a systematic approach. In other settings, Newton's method, Hill-climbing and Simulated Annealing are popular alternatives. The problem distribution P is not always known or explicitly specified, but it can be

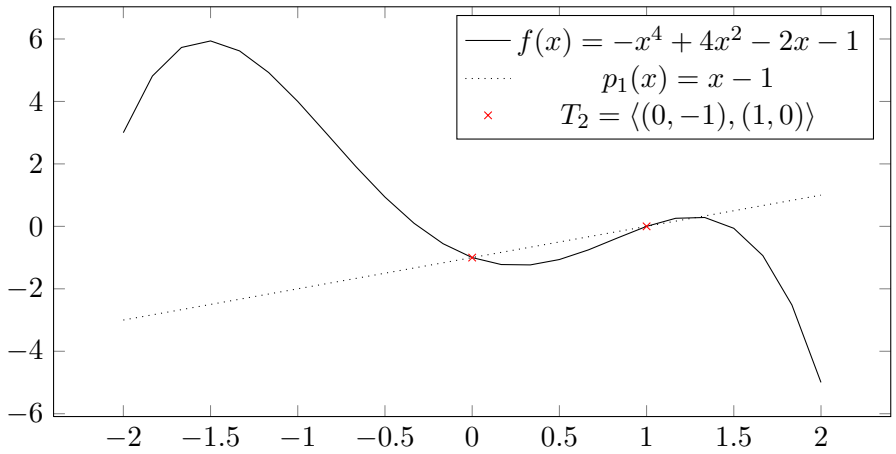


Figure 2: The situation for the search algorithm in Example 4 after two probes. The algorithm has searched at 0 and 1, which has generated the search trace $T_2 = \langle (0,-1),(1,0) \rangle$. The algorithm (erroneously) conjectures that $p_1(x) = x - 1$ might be the true function.

argued that even when the problem distribution is not specified, an implicit distribution is used. For example, if one uses a hill-climbing algorithm it may be argued that one is implicitly assuming that the function is somewhat continuous. This may be expressed as a prior problem-distribution with (much) higher weight on continuous functions than on "scattered" ones.

As the true function is not known, the goal is typically to do well in P-expectation (a precise definition of how to measure performance will be given in Definition 9.1 below). ♢

## 7.4 Permutations, functions and enumerating algorithms

Many of the subsequent results will depend on permutations of different types of objects.

**Definition 7.3** (Permutations)**.** Define a *permutation of size* $n$ as a bijective function $\sigma : \{1,\ldots,n\} \to \{1,\ldots,n\}$. Extend the domain of permutations $\sigma$ of size $n$ to arbitrary ordered sets of size $n$, according to $\sigma(X) = \{\sigma(x_1),\ldots,\sigma(x_n)\} = \{x_{\sigma(1)},\ldots,x_{\sigma(n)}\}$ and to functions $f : X \to Y$ by $(\sigma f)(x) = f(\sigma^{-1}(x))$ where $X$ is some ordered set of size $n$. Subsequently permutations will always be assumed to be of the size of the search space $|X|$, and the set of all such permutations will be denoted by $\Pi$.

To illustrate some applications of Definition 7.3, define an *enumerating algorithm* as an algorithm that searches $X$ in a pre-defined order without heeding the returned $y$-values. Let $e$ be the enumerating algorithm that

searches $X$ in order, and for any permutation $\sigma \in \Pi$ let $e_\sigma$ be the enumerating algorithm that searches $X$ in the order $\sigma(X)$. (Recall that $X$ is an ordered set by assumption.)

It is then easy to see that $e$ generates the full-length result vector $\langle f(x_1), \ldots, f(x_{|X|})\rangle$ exactly when $f$ is the true function. Analogously, $e_\sigma$ generates the full-length result vector $\langle f(\sigma(x_1)), \ldots, f(\sigma(x_{|X|}))\rangle = \langle (\sigma^{-1}f)(x_1), \ldots, (\sigma^{-1}f)(x_{|X|})\rangle$ exactly when $f$ is the true function. It follows that the full-length result vector of $e_\sigma$ is $\langle f(x_1), \ldots, f(x_{|X|})\rangle$ exactly when $\sigma f$ is the true function.[9]

Functions are naturally identified with sets of ordered pairs, by $f = \{(x_1, f(x_1), \ldots, (x_{|X|}, f(x_{|X|})))\}$. Using the order on $X$, we will sometimes also identify functions with tuples $f = (f(x_1), \ldots, f(x_{|X|}))$.

The latter representation coincides with the representation of full -length result vectors (result vectors of length $|X|$). Incidentally, this allows us to express our observation about enumerating algorithms rather concisely. The observations can be expressed as $T^y_{|X|}(e, f) = f$ and $T^y_{|X|}(e_\sigma, \sigma f) = f$. We state this as a lemma for future reference.

**Lemma 7.4** (Result vectors of enumerating algorithms)**.** *The result trace for the enumerating algorithm $e$ on any function $f$ is the associated tuple for the function itself, that is $T^y_{|X|}(e, f) = f$. Further, if $\sigma \in \Pi$ then $T^y_{|X|}(e_\sigma, \sigma f) = f$.*

On a related note, we also introduce the concept of an *informed choice*.

**Definition 7.5** (Informed choice)**.** A choice of a next search point is *informed* if it depends on the perceived $y$-values, and it is *blind* if it only depends on the probing sequence. An algorithm that only makes blind choices is called a *blind* algorithm. Formally, $a$ is blind if for all traces $S$ and $T$ with $T^x = S^x$, and all $x \notin T^x$ it holds that $\Pr_a(x \,|\, T) = \Pr_a(x \,|\, S)$.

Enumerating algorithms are blind deterministic algorithms. For this reason, they are sometimes also known as *oblivious algorithms* [GO05].

An example of a stochastic blind algorithm is the random algorithm. It can be defined as the probabilistic algorithm *rand*, with $\Pr_{rand}(x \,|\, T) = 1/(|X| - |T^x|)$ for any trace $T$ and any $x \notin T^x$ (and 0 for $x \in T^x$).

## 7.5 A measure on traces

In this section the goal will be to derive a measure $\mathrm{P}_a$ over the set of full-length traces $\mathcal{T}_{|X|}$, where $\mathrm{P}_a(T)$ will be the probability that $a$ generates $T$. Formally, $\mathrm{P}_a$ will be a measure on the measure space $(2^{\mathcal{T}_{|X|}}, \mathcal{T}_{|X|})$. We will sometimes refer to $\mathrm{P}_a$ as the *trace-measure* induced by P and $a$.

[9] The concept of "permuting" the enumerating algorithm $e$ can be generalized to arbitrary algorithms. This leads to a generalized result on how permutations interact when applied to functions and algorithms respectively [SVW01].

Given a function $f$, a trace $T = \sigma\langle(x_1, y_1), \ldots, (x_{|X|}, y_{|X|})\rangle$ is possible if and only if $f(x_i) = y_i$ for all $1 \leq i \leq |X|$. Therefore the measure $\mathrm{P}_a$ must be a refinement of P in the sense that $\sum_{\sigma\in\Pi} \mathrm{P}_a(\sigma\langle(x_1, y_1), \ldots, (x_{|X|}, y_{|X|})\rangle) = \mathrm{P}(f)$. From the perspective of $\mathrm{P}_a$, the event of a function $f$ will be identified with the set of all traces consistent with $f$.

In a similar manner, the event of a trace $T_k$ of length $k < |X|$ will be identified with the set of all full-length extensions of $T_k$. Introducing the notation $T[i{:}j] = \langle t_i, \ldots, t_j\rangle$ and $T[i] = t_i$ for extracting components of a vector $T = \langle t_1, \ldots, t_n\rangle$, the event of $T_k$ is formally identified with the set $\{T \in \mathcal{T}_{|X|} : T[1{:}k] = T_k\}$.

By combing the measure $\mathrm{Pr}_a(T \,|\, f)$ from Proposition 7.2 with P we arrive at a measure $\mathrm{P}_a$ over $\mathcal{T}_{|X|}$,

$$\mathrm{P}_a(T) = \sum_{f\in F} \mathrm{P}(f)\mathrm{Pr}_a(T \,|\, f) \tag{18}$$

$$= \sum_{f\in F} \mathrm{P}(f) \sum_{a\in\mathcal{A}} \mathrm{Pr}_b(a)[[T_n(a, f) = T]] \tag{19}$$

This defines the measure $\mathrm{P}_a$ for all points (singleton events) $T \in \mathcal{T}_{|X|}$, which uniquely determines $\mathrm{P}_a$ for all subsets (events) of $\mathcal{T}_{|X|}$ by Carathéodory's extension theorem.

$\mathrm{P}_a$ is obviously non-negative and additivity follows from the additivity of $\mathrm{Pr}_a$. The following verifies that $\mathrm{P}_a$ sums to 1.

$$\mathrm{P}_a(\mathcal{T}_{|X|}) = \sum_{S\in\mathcal{T}_{|X|}} \mathrm{P}_a(S) \tag{20}$$

$$= \sum_{S\in\mathcal{T}_{|X|}} \sum_{f\in F} \mathrm{P}(f)\mathrm{Pr}_a(S \,|\, f) \tag{21}$$

$$= \sum_{f\in F} \mathrm{P}(f) \sum_{S\in\mathcal{T}_{|X|}} \mathrm{Pr}_a(S \,|\, f) \tag{22}$$

$$= \sum_{f\in F} \mathrm{P}(f) = 1 \tag{23}$$

So $\mathrm{P}_a$ is a proper measure (Definition 5.2 on page 24).

### 7.5.1 Types of events

To be able to speak about events such as "$a$ searches at $x$ in its $k$th probe", we define random variables $x^k$ and $y^k$ for the $k$th search point and the $k$th perceived function value. Formally, for every $1 \leq k \leq |X|$, they can be defined as

$$x^k : T \in \mathcal{T}_{|X|} \mapsto T^x[k] \tag{24}$$

$$y^k : T \in \mathcal{T}_{|X|} \mapsto T^y[k] \tag{25}$$

For example, $\mathrm{P}_a(y^1=2\,|\,x^1=10)$ denotes the probability that $a$ finds the value 2 at its first probe, given that $a$'s first probe is at 10. Note that $\mathrm{P}_a(x^{n+1}=x\,|\,\langle x^1y^1\ldots x^ny^n\rangle=T) = \mathrm{Pr}_a(x\,|\,T)$.

The random variables $x^k$ and $y^k$ enable us to describe any property of the search trace. Although possible, they will not be used to describe all types of events. In particular, a trace $T_k$ will still denote the event of all full-length extensions of $T_k$; a result vector $R$ (of any length) will denote the event of all traces consistent with $R$; and, finally, the event of a function $f$ is the event of all traces consistent with $f$. As all these objects have different formal representations (traces are ordered lists of $X$,$Y$-pairs, result vectors ordered lists of $y$-values, and functions sets of pairs), there should be no risk of confusion. For example, $\mathrm{P}_a(R\,|\,f) = \mathrm{Pr}_a(R\,|\,f)$ is the probability that $a$ generates the result vector $R$ on $f$.

# 8 Literature review

We now give an overview of the research done on NFL and optimisation, before we go into a more detailed treatment in subsequent sections.

## 8.1 NFL Theorems

Wolpert and Macready [WM95, WM97] are generally credited for the first formal proof of the No Free Lunch (NFL) theorem for black-box optimisation, proving that when uniformly averaged over all functions, no search algorithm will be any better or worse than any other.[10] Schumacher et al. [SVW01] made an important generalization to more general classes of functions. This also sharpened the theorem. They observed that NFL holds for a uniform prior over a class $F$ of functions if and only if $F$ is closed under permutation (c.u.p.). This theorem is often known as the *sharpened* NFL theorem. A more detailed account of Wolpert and Macready's and Schumacher et al.'s theorems are given in Section 9.2.

Igel and Toussaint [IT04] generalized the sharpened NFL theorem to non-uniform averages, showing that the precise condition for NFL is that every permutation $\sigma f$ of a function $f$ have the same probability/weight as $f$—a so-called block-distribution is required (see Section 9.3). Streeter [Str03] reformulated the result in a Bayesian setting, showing that the conditions for NFL can equivalently be phrased in terms of (zero) information gain for a Bayesian learner.

## 8.2 Classes of functions

There are many classes of functions for which NFL does not hold, an immediate consequence of the fact that most function classes that are not c.u.p.

[10] See also [Sch94, RS95, WB69].

(the fraction of c.u.p. function classes vanishes quickly with growing $X$ and $Y$ [IT04]). In particular, there are free lunches for the subsets of functions defined by the conditions:

1. A global minimum is not adjacent to a global maximum [IT04],
2. The number of local minima is bounded [IT01, IT04]
3. The maximum steepness is bounded [IT01, IT03b],
4. The Kolmogorov complexity is lower than $k$, [Str03, McG06].

In all cases, the reason is that the sets are not c.u.p. However, at least in the case of subsets formed by a bound on the Kolmogorov complexity, there may still be further subsets for which NFL holds, as demonstrated by [SVW01].

Streeter [Str03] also found that if the average is taken with respect to the universal distribution, free lunch exists. In Section 11.3 we prove a related theorem. In the related area of supervised learning, Lattimore and Hutter [LH11] showed that there is free lunch under the universal distribution.

Griffiths and Orponen [GO05] studied the more specific case of binary functions with a particular maximization performance measure. Unsurprisingly, using a particular performance measure yields less free lunch. In their setting, NFL sometimes hold for function classes not c.u.p.

## 8.3 Single functions, searchability

It is also possible to ask on which functions an algorithm will be able to outperform a random search. The pioneering work on this interesting field was done by Christensen and Oppacher [CO01], who conjectured that a function would be *searchable* if it was "self-similar", i.e., points close in the search space should have close target values (compare Item 3 in the list above). They developed a simple measure of self-similarity by counting the number of jumps over the median, and showed that a "submedian-seeker" algorithm outperformed random search on functions with few median-jumps.

This work was later extended in [WR06], and, in turn, by Jiang and Chen [JC11]. The latter study developed a general model of Lipschitz-continuity for any structured search space. They also gave an algorithm outperforming random search on many Lipschitz-continuous functions, and being equally good on some very hard Lipschitz-functions. Hard Lipschitz-functions include the so-called needle-in-a-haystack (NIAH) functions, defined and discussed in Section 12.2.

Lipschitz-continuity and self-similarity may be criticized as searchability criteria, however, since on the one hand they include provably hard of functions such as NIAH functions (proven hard in [DJW04]), and on the other

they exclude intuitively searchable functions[11] such as

$$f(x) = \begin{cases} x & \text{if } x \text{ even} \\ 0 & \text{otherwise.} \end{cases}$$

So Lipschitz-continuity is neither a sufficient nor necessary condition for searchability. (It may still be the correct criteria for many practical optimisers such as Genetic Algorithms (GA) and Simulated Annealing (SA), however, as argued by Christensen and Oppacher in [CO01].)

As predictability (of the maximum, at least) seems to be a key aspect, it is natural to turn to Kolmogorov complexity $K$ and the universal distribution. Borenstein and Poli [BP06] informally investigate the possibility to use the universal distribution in optimisation. They argue that since it is possible to "hide" a maximum as a hard-coded point in the program with small ($\mathcal{O}(\log_2|\mathcal{X}|)$) penalty in complexity, there are many functions with low KC that still are hard to search. As examples, they mention NIAH-functions and trap-functions (a trap-function is a compressible function with one hard-coded exception). However, their argument is informal (and they seem to neglect the fact that it is easier to encode exceptions in compressible points rather than in incompressible ones, which, for example, makes some NIAH/trap-functions much simpler than others). In Section 11.2 we define compressibility of functions and points, and in Section 12.3 we give a formal argument for why NIAH-functions make it hard to search under $\mathbf{m}$.

Having discarded KC as unsuitable for a searchability criterion, Borenstein and Poli instead propose the following definition of *KC-continuity* as an alternative.

**Definition 8.1.** A function $f$ is *KC-continuous* in a point $x$ if

$$f(x) = \arg\min_y K\left(g(x') = \begin{cases} y & \text{if } x' = x \\ f(x') & \text{otherwise.} \end{cases}\right)$$

The intention is to disallow the hiding of the maximum in one random point, and thus better capture the notion of searchability. It is unclear, however, to what extent the criteria works (beyond discarding NIAH-functions and trap-functions). They claim that a simple function can only be hard to search for a KC-based searcher if the function is KC-discontinuous at its maxima, but no proof is provided.

## 8.4 Almost No Free Lunch

Even though free lunches are proven for a large number of (classes of) functions, as discussed above, there are also negative results on the amount

[11] To the Lipschitz-criterion's defence, a NIAH function can only be Lipschitz if the needle is very low. So when it is sufficient to find a value *close* to the maximum, the Lipschitz functions may still be intuitively searchable.

of free lunch available. Such results are sometimes called Almost No Free Lunch (ANFL) theorems. The most important one is found in [DJW02], and shows that given an algorithm $a$ that does well on a function $f$, there are many only slightly more complex functions on which $a$ does extremely badly (this, of course, precedes and pertains to the above arguments by Borenstein and Poli). As a concrete example, they discuss the ONEMAX problem (in which the search space is all binary strings of length $n$, and $f(x) =$"number of 1's in $x$") and a slight modification of it called TRAP where $f(0^n) = n + 1$. They show that both Genetic Algorithms (GA) and Simulated Annealing (SA) will perform poorly on TRAP, despite its fairly simple description. Further, and without good argument, they conjecture that for every "reasonable" (not formally defined) search-algorithm, there is some "simple" problem on which it does badly. The claim is weak, as neither GA nor SA are universally biased towards simple/structured problems.

Whitley and Rowe [WR08] develop an ANFL theorem for classes, proving that if a class $F$ is not c.u.p. (and thus have free lunch) there is a "small" extension of $F$ on which NFL holds. This has some interest, since the "permutation closure" of $F$ may be significantly larger than $F$. Even so, it says little about the actual amount of free lunch on $F$, which normally is the quantity of interest.

Our results in Section 12 below can be interpreted as ANFL-theorems for the universal distribution.

### 8.5 Other investigations

A ubiquitous assumption so far has been the finiteness of the search space $X$ and the range $Y$. Auger and Teytaud [AT07] generalize the NFL problem to continuous (infinite) search spaces and conclude that free lunches exist for such domains. Different generalizations to the infinite case are possible, however, and not all provide free lunch [RVW09]. A finite search space together with an infinite co-domain $Y$ yields no free lunch [Str03].

One may also question the no-revisiting condition; for large search spaces it may be infeasible to keep all visited points in memory. Some consequences of forgetfulness is explored in [MH09] and in a study on multi-objective optimisation [CK03].

## 9 No free lunch

This section will give a detailed account of the "classical" NFL theorems by Wolpert and Macready [WM97], Schumacher et al. [SVW01] and Igel and Toussaint [IT04] (Theorem 9.5, 9.7 and 9.9 below). The classical NFL theorems are all similar in spirit, they only differ in generality. They essentially show that under certain conditions no search algorithm can outperform another. For example, this implies that a hill-climbing algorithm will find the

maximum equally slowly as a hill-descending algorithm! The NFL theorems thus renders any attempt at intelligent search moot (in the settings they apply to).

By the end of this section, we investigate how the possible difference in search performance—the amount of free lunch, so to speak—is affected by small deviations from the conditions required for NFL.

## 9.1 Two equivalent NFL-definitions

**Definition 9.1** (Performance measure)**.** A *performance measure* is a function $M : \mathcal{T}^y \to \mathbb{R}^+ \bigcup \{0\}$ that measures how different result vectors are valued. Generally, the *expected performance* of an algorithm $a$ on a problem P is the interesting quantity; we therefore extend the performance measure to algorithms $a$ and search lengths $n$ the following way.

$$M^{\mathrm{P}}(a,n) = \sum_{R \in \mathcal{T}_n^y} \mathrm{P}_a(R) M(R) \tag{26}$$

$$= \sum_{\substack{R \in \mathcal{T}_n^y \\ f:X \to Y}} \mathrm{P}(f) \cdot \mathrm{Pr}_a(R \,|\, f) M(R) \tag{27}$$

For deterministic algorithms, this reduces to

$$M^{\mathrm{P}}(a,n) = \sum_{f \in F} \mathrm{P}(f) M(T_n^y(a,f)) \tag{28}$$

When the search length $n$ is clear from the context, it may be omitted. In the beginning of Section 10, a *full-length performance measure* is defined as a performance measure only defined on full-length result vectors.

For some search problems P it can be shown that all algorithms perform equally well with respect to some (or all) performance measure(s). This is often phrased as "there is no free lunch available for P"; hence the term No Free Lunch (NFL).

**Definition 9.2** (Performance measure-NFL)**.** *NFL* holds for a search problem P and a performance measure $M$ if $M^{\mathrm{P}}(a,n) = M^{\mathrm{P}}(b,n)$ for every pair of search procedures $a$ and $b$, and every search length $n$. If NFL holds for *all* performance measures $M$, then NFL simply holds for the problem P. If NFL does not hold for a problem P (and performance measure $M$) we say that there is *free lunch* for P (under $M$).

The stronger version of NFL that holds for all performance measures may equivalently be defined in terms of result vectors. The equivalence will be verified shortly in Lemma 9.4.

**Definition 9.3** (Result vector-NFL)**.** *NFL* holds for a search problem P in the result-vector sense if every result vector is equally likely to be generated by all algorithms; that is, $\mathrm{P}_a(R) = \mathrm{P}_b(R)$ for every pair of search algorithms $a$ and $b$ and every result vector $R \in \mathcal{T}^y$.[12]

Despite being slightly less intuitive, the result-vector definition is often more convenient to use. Both because theorems about result vectors tend to be more useful as auxiliary results, and because theorems about result vectors often are (slightly) more direct to prove. The following lemma verifies the equivalence of the two definitions.

**Lemma 9.4** (Equivalence of NFL definitions [Str03])**.** *For a given search problem* P*, it holds that* $\mathrm{P}_a(R) = \mathrm{P}_b(R)$ *for all search procedures* $a$ *and* $b$ *and all result vectors* $R \in \mathcal{T}^y$ *if and only if* $M^{\mathrm{P}}(a, n) = M^{\mathrm{P}}(b, n)$ *for all performance measures* $M$*, all procedures* $a$*,* $b$ *and all search lengths* $n$*.*

*Proof.* Assume that NFL does not hold in the result-vector sense. Then there are two procedures $a$ and $b$ and a result vector $R_n$ of length $n$ such that $\mathrm{P}_a(R_n) \neq \mathrm{P}_b(R_n)$. Define the (degenerate) performance measure $M$ that is 1 only on the result vector $R_n$, and 0 otherwise. This means that

$$M^{\mathrm{P}}(a, n) = \sum_{R \in \mathcal{T}_n^y} \mathrm{P}_a(R) M(R) = \mathrm{P}_a(R_n) M(R_n) = \mathrm{P}_a(R_n)$$

The same applies to $b$, so $M^{\mathrm{P}}(b, n) = \mathrm{P}_b(R_n)$. By assumption, $\mathrm{P}_a(R_n) \neq \mathrm{P}_b(R_n)$, so $a$ and $b$ perform differently with respect to $M$.

Assume conversely that NFL does hold in the result-vector sense. Then $\mathrm{P}_a(R) = \mathrm{P}_b(R)$ for all $R \in \mathcal{T}^y$ and all procedures $a$ and $b$, which means that for any performance measure $M$ and search length $n$,

$$M^{\mathrm{P}}(a, n) = \sum_{R \in \mathcal{T}_n^y} \mathrm{P}_a(R) M(R) = \sum_{R \in \mathcal{T}_n^y} \mathrm{P}_b(R) M(R) = M^{\mathrm{P}}(b, n)$$

So NFL holds for all performance measures $M$. □

Since NFL holds in the result-vector sense if and only if it holds in the performance-measure sense, we will generally not distinguish between the two subsequently.

## 9.2 Uniform distributions

Wolpert and Macready [WM97] famously proved that if the sampling of a function $f : X \to Y$ is uniform, than NFL holds. Formally, let

$$u_F(f) = \begin{cases} 1/|F| & \text{for } f \in F \\ 0 & \text{otherwise} \end{cases}$$

[12] Observe that $\mathcal{T}^y$ includes result vectors of *all lengths*.

be the *uniform distribution* over a class of function $F$. The formal statement of Wolpert and Macready's result is:

**Theorem 9.5** (NFL for uniform distribution [WM97])**.** *NFL holds for the problem $u_F$, when $F = Y^X$.*

The theorem is a special case of the more general Theorem 9.7 and Theorem 9.9 below. See Appendix A for a proof.

Schumacher et al. [SVW01] extended Wolpert and Macready's theorem to more general classes of functions. Their result, sometimes known as the Sharpened NFL Theorem, has become pivotal to much of the subsequent NFL research. It depends on the notion of a function class closed under permutation (c.u.p.).

**Definition 9.6** (Closed under permutation)**.** A class $F$ of functions is *c.u.p.* if $f \in F \implies \sigma f \in F$ for all permutations $\sigma$.

**Theorem 9.7** (NFL for c.u.p. classes [SVW01])**.** *NFL holds for a search problem $u_F$ if and only if $F$ is c.u.p.*

The theorem is a special case of of Theorem 9.9 below. We defer the proof until Appendix A.

## 9.3 Non-uniform distributions

The c.u.p. NFL theorem (Theorem 9.7) gives a precise condition for NFL for *uniform* distributions. In this section we present an important result by Igel and Toussaint [IT04] that generalizes this to arbitrary distributions.

To generalize the c.u.p. property, *histograms* can be used to describe *base classes*, the smallest building blocks for c.u.p. function classes. A histogram for a function $f$ is a function $h_f : Y \to \mathbb{N}$ indicating how many $x$'s map to every $y$ (that is, $h_f(y) = |f^{-1}(y)|$). The set of all functions with the histogram $h$ is called the *base class* of $h$, and is denoted by $B_h$. Base classes are the smallest c.u.p. function classes in the sense that all base class are c.u.p., and any c.u.p. class is the union of some base classes [IT01]. Base classes entail the following useful generalizations of the c.u.p. property.

**Definition 9.8** (Block uniformity)**.** A problem distribution P is *block uniform* if every pair of functions $f, g \in Y^X$ in the same base class are equally likely; that is, if $f, g \in B_h \implies \mathrm{P}(f) = \mathrm{P}(g)$. More generally, for $\varepsilon \geq 0$ the distribution P is *$\varepsilon$-block uniform* if $f, g \in B_h \implies |\mathrm{P}(f) - \mathrm{P}(g)| \leq \varepsilon$.

Using block uniformity, Igel and Toussaint generalized the NFL Theorem 9.7 to non-uniform problems P. Our proof differs from the original proof in [IT04] by showing the result directly, rather than using Theorem 9.7 as a middle step. Our result is also more general as it applies to probabilistic algorithms; Igel and Toussaint's theorem only considered deterministic algorithms.

**Theorem 9.9** (Non-uniform NFL [IT04])**.** *NFL holds for a problem* P *if and only if* P *is block uniform.*

*Proof.* (IF) Suppose P is block uniform. Let $X = \{x_1, \ldots, x_{|X|}\}$ and let $F$ be the set of all functions $f : X \to Y$. Define a *base-class-respecting (function-)mapping* to be a mapping $\psi : F \to F$ such that $\psi f$ is always in the same base class as $f$. By the block uniformity of P, for any base-class respecting mapping $\psi$ holds that $\mathrm{P}(\psi f) = \mathrm{P}(f)$.

It turns out that deterministic algorithms $a$ naturally induces base-class-respecting mappings by their result vectors. For example, if $f = (4, 4, 5)$ and $a$ searches $X = \{x_1, x_2, x_3\}$ in the order $x_2, x_3, x_1$, then this naturally maps $f$ to $(4, 5, 4)$ (by the result vector $T^y_{|X|}(a, f) = \langle 4, 5, 4\rangle$).

Formally, define for every deterministic algorithm $a$ the mapping $\psi_a$ by

$$\psi_a f = T^y_{|X|}(a, f) \tag{29}$$

using the identification of functions with tuples described in Section 7.4. That is, $\psi_a f$ maps every $x_i$ to the $i$th component of the full-length result vector $a$ creates on $f$. Since the result vector contains every function value once, the histogram is preserved, so $\psi_a$ respects base-classes.

Further, and more surprisingly, $\psi_a$ is bijective. We will show this by showing that $\psi_a$ is surjective; this suffices since $F$ is finite.

Pick an $f$ in $F$. The following constructs a function $g$ such that $\psi_a g = f$. Let

$$g(a(\langle\rangle)) = f(x_1) \tag{30}$$

In words, $g$ takes value $f(x_1)$ on the point $a$ starts searching. The algorithm $a$ thus obtains the trace $T_1(a, g) = \langle (a(\langle\rangle), f(1))\rangle$ on $g$. At the second point $a$ searches when encountering $f(x_1)$ in its first probe, $g$ takes on the value $f(x_2)$:

$$g(a(T_1(a, g))) = f(x_2) \tag{31}$$

And so on. In general, we let

$$g(a(T_n(a, g))) = f(x_{n+1}) \tag{32}$$

for $n < |X|$. Then $a$ generates the result vector $T^y_{|X|}(a, g) = \langle f(x_1), f(x_2), \ldots, f(x_{|X|})\rangle$ on $g$. Therefore, $(\psi_a g)(x_i) = T^y_{|X|}(a, g)[i] = f(x_i)$ for all $i$, so $\psi_a g = f$. Hence $\psi_a$ is surjective, and thereby also bijective.

We will now use the mapping $\psi_a$ to show that $a$ generates every result vector $R$ with the same probability as the enumerating algorithm $e$. Recall that the enumerating algorithm $e$ searches $X$ in order, and thus creates the result vector $\langle f(x_1), \ldots, f(x_{|X|})\rangle$ exactly when $f$ is the true function (Section

7.4; Lemma 7.4). Importantly, this means that $T_n^y(a,g) = T_n^y(e,\psi_a g)$ for any $n$.

Now, for any $n$ and any result vector $R$ of length $n$,

$$\mathrm{P}_a(R) = \sum_{g\in F} \mathrm{P}(g)[[T_n^y(a,g) = R]] \tag{33}$$

$$= \sum_{g\in F} \mathrm{P}(g)[[T_n^y(e,\psi_a g) = R]] \tag{34}$$

by the just motivated equality. Since $\psi_a$ respects base-classes and is bijective, (34) equals

$$= \sum_{\psi_a g\in F} \mathrm{P}(\psi_a g)[[T_n^y(e,\psi_a g) = R]] \tag{35}$$

$$= \mathrm{P}_e(R) \tag{36}$$

This shows that any deterministic algorithm $a$ generates an arbitrary result vector $R$ with the same probability as the enumerating algorithm $e$. By transitivity, this shows that all *deterministic* algorithms generate the same result vectors with the same probability.

To generalize the result to probabilistic algorithms, let $c_R$ be the probability that a deterministic algorithm generates result vector $R$ and let $b$ be a probabilistic algorithm. By definition,

$$\mathrm{P}_b(R) = \sum_{a\in\mathcal{A}} \mathrm{Pr}_b(a)\mathrm{P}_a(R) \tag{37}$$

$$= c_R \sum_{a\in\mathcal{A}} \mathrm{Pr}_b(a) = c_R \tag{38}$$

That is, probabilistic algorithms also generate result vector $R$ with probability $c_R$. This shows that NFL holds for any block uniform problem P.

(ONLY IF) Assume that P is not block uniform. Then there are two functions $f$ and $\sigma f$ in the same base class $B_h$ such that $\mathrm{P}(f) > \mathrm{P}(\sigma f)$. Recall the enumerating algorithms $e$ and $e_\sigma$ from Section 7.4. By Lemma 7.4 we had that $e$ generates the result vector $R_f = \langle f(1),\ldots,f(|X|)\rangle$ exactly when $f$ is the true function, and that $e_\sigma$ generates $R_f$ exactly when $\sigma f$ is the true function. An immediate consequence is that $\mathrm{P}_e(R_f) = \mathrm{P}(f) > \mathrm{P}(\sigma f) = \mathrm{P}_{e_\sigma}(R_f)$. That is, $e$ and $e_\sigma$ generate $R_f$ with different probability, which means that NFL does not hold (Lemma 9.4). □

## 9.4 Continuity of NFL

An interesting generalization of Theorem 9.9 is what happens when the distribution P is *almost block uniform*, naturally formalized by $\varepsilon$-block uniformity (Definition 9.8). As we shall see, the amount of free lunch grows at most linearly with increasing $\varepsilon$-perturbations of the block uniformity.

First a lemma is required.

**Lemma 9.10** (Equal number of functions)**.** *For any base class $B_h$, any result vector $R \in \mathcal{T}^y$ and any two deterministic algorithms $a$ and $b$ holds that*

$$|\{f \in B_h : T_n^y(a, f) = R\}| = |\{f \in B_h : T_n^y(b, f) = R\}| \tag{39}$$

*That is, $a$ and $b$ produce the result vector $R$ on equally many functions in $B_h$.*

*Proof.* Assume the problem $\mathrm{P} = u_{B_h}$ is uniform over $B_h$ and 0 otherwise. Then $\mathrm{P}_a(R) = \mathrm{P}_b(R)$ for all algorithms $a$ and $b$ and all result vectors $R$. Let $R$ be a result vector of length $n$. Then

$$\begin{aligned}\mathrm{P}_a(R) &= \sum_{f:X\to Y} \mathrm{P}(f)[[T_n^y(a, f) = R]] \\ &= |\{f \in B_h : T_n^y(a, f) = R\}| \cdot 1/|B_h|\end{aligned}$$

for all algorithms $a$. Therefore all algorithms must produce the same result vector $R$ on equally many functions. □

**Theorem 9.11** (Continuity of NFL)**.** *If the problem* P *is $\varepsilon$-block uniform for some (small) $\varepsilon \geq 0$, then the amount of free lunch is bounded by*

$$|M^{\mathrm{P}}(a, n) - M^{\mathrm{P}}(b, n)| \leq \varepsilon \sum_{f\in F} M(T_n^y(a, f))$$

*for any performance measure $M$, any pair of algorithms $a$, $b$ and any search length $n$. In particular, the amount of free lunch goes to 0 as $\varepsilon$ goes to 0.*

*Proof.* Since P is $\varepsilon$-block uniform, there is a number $p_h$ for every $B_h$ such that $f \in B_h \implies \mathrm{P}(f) \in [p_h - \varepsilon/2, p_h + \varepsilon/2]$. Let $H_{a,R,F} = \{f \in F : T_n^y(a, f) = R\}$ be the set of functions in $F$ on which a deterministic algorithm $a$ generates the result vector $R$. (Note that it was the size of certain sets $H_{a,R,F}$ that was investigated in Lemma 9.10.)

Let $a$ and $b$ be two deterministic algorithms. Then,

$$M^{\mathrm{P}}(a, n) - M^{\mathrm{P}}(b, n) = \tag{40}$$

$$= \sum_{f\in F} \mathrm{P}(f)M(T_n^y(a, f)) - \sum_{g\in F} \mathrm{P}(g)M(T_n^y(b, g)) \tag{41}$$

$$= \sum_{B_h} \sum_{R\in\mathcal{T}_n^y} \Big( \sum_{f\in H_{a,R,B_h}} \mathrm{P}(f)M(T_n^y(a, f)) - \sum_{g\in H_{b,R,B_h}} \mathrm{P}(g)M(T_n^y(b, g))\Big) \tag{42}$$

by first expanding definitions, then splitting the sum over base classes and result vectors. Since $T_n^y(a,f) = R$ for all $f \in H_{a,R,B_h}$, (42) equals

$$= \sum_{B_h} \sum_{R \in \mathcal{T}_n^y} \left( \sum_{f \in F} \mathrm{P}(f) M(R) - \sum_{g \in F} \mathrm{P}(g) M(R) \right) \tag{43}$$

$$= \sum_{B_h} \sum_{R \in \mathcal{T}_n^y} M(R) \Big( \sum_{f \in H_{a,R,B_h}} \mathrm{P}(f) - \sum_{g \in H_{b,R,B_h}} \mathrm{P}(g) \Big) \tag{44}$$

which means that we can extract $M(R)$ (44). We now use the $\varepsilon$-block uniformness to bound (44) by

$$\leq \sum_{B_h} \sum_{R \in \mathcal{T}_n^y} M(R) \Big( \sum_{f \in H_{a,R,B_h}} (p_h + \varepsilon/2) - \sum_{g \in H_{b,R,B_h}} (p_h - \varepsilon/2) \Big) \tag{45}$$

By Lemma 9.10, $a$ and $b$ produce the result vector $R$ on equally many functions in a base class $B_h$, which implies that (45) equals

$$= \sum_{B_h} \sum_{R \in \mathcal{T}_n^y} M(R) |H_{a,R,B_h}| \big( p_h + \varepsilon/2 - (p_h - \varepsilon/2) \big) \tag{46}$$

The $\varepsilon$ may now be moved out and the summation rewritten

$$= \varepsilon \sum_{B_h} \sum_{R \in \mathcal{T}_n^y} |H_{a,R,B_h}| M(R) = \varepsilon \sum_{B_h} \sum_{R \in \mathcal{T}_n^y} \sum_{f \in B_h : T_n^y(a,f) = R} M(R) \tag{47}$$

$$= \varepsilon \sum_{B_h} \sum_{f \in B_h} M(T_n^y(a,f)) = \varepsilon \sum_{f \in F} M(T_n^y(a,f)) \tag{48}$$

This shows the theorem for deterministic algorithms $a$ and $b$. Thanks to Proposition 7.2, we easily obtain a generalization to probabilistic algorithms.

Fix a search length $n$. Then there are two real numbers $r_1$, $r_2$ such that for all deterministic algorithm $b$

$$r_1 \leq M^{\mathrm{P}}(b,n) \leq r_2 \tag{49}$$

with $r_2 - r_1 \leq \varepsilon \sum_{f \in F} M(T_n^y(a,f))$ for an arbitrary $a$, by equations (40)–(48).

Let $a'$ be a probabilistic algorithm. Then

$$M^{\mathrm{P}}(a',n) = \sum_{b \in \mathcal{A}} \mathrm{Pr}_{a'}(b) M(T_n^y(b,n)) \leq r_2 \sum_{b \in \mathcal{A}} \mathrm{Pr}_{a'}(b) = r_2 \tag{50}$$

By a similar argument, $r_1 \leq M^{\mathrm{P}}(a',n)$, so $r_1 \leq M^{\mathrm{P}}(a',n) \leq r_2$. The same holds, of course, for any other probabilistic algorithm $b'$. Therefore,

$$|M^{\mathrm{P}}(a',n) - M^{\mathrm{P}}(b',n)| \leq r_2 - r_1 \leq \varepsilon \sum_{f \in F} M(T_n^y(a',f)) \tag{51}$$

holds also for probabilistic algorithms. □

Theorem 9.11 can be interpreted as a continuity result. The amount of free lunch is continuous in the "block-uniform points" in the space of problem-distributions (with distance $d(\mathrm{P}_1, \mathrm{P}_2) = \max_{f:X\to Y} |\mathrm{P}_1(f) - \mathrm{P}_2(f)|$ between two problem distributions $\mathrm{P}_1$ and $\mathrm{P}_2$).

Although the growth of potential free lunch is bounded by a linear function in $\varepsilon$, the growth in $\varepsilon$ is typically rather fast. For example, if the search space is of size $|X| = 100'000$ and the average performance measure is about 1, then the bound is $\varepsilon 100'000$ for $\varepsilon$-block uniform problems. Perhaps a smaller bound could be proven? This question will be addressed shortly in Section 9.4.1.

**Corollary 9.12.** *Under the extra condition that there is an $k \in \mathbb{R}$ such that $M(R) \leq k$ for all performance measures $M$ and all result vectors $R$, then $|M^{\mathrm{P}}(a,n) - M^{\mathrm{P}}(b,n)| \leq \varepsilon k|F| \leq \varepsilon k|Y|^{|X|}$ for all performance measures $M$, where $|F|$ is the number of functions with probability $> 0$.*

*Proof.* This is an immediate corollary of Theorem 9.11: $|M^{\mathrm{P}}(a,n) - M^{\mathrm{P}}(b,n)| \leq \varepsilon \sum_{f\in F} M(T_n^y(a,f) \leq \varepsilon k|F| \leq \varepsilon k|Y|^{|X|}$. □

### 9.4.1 Tightness

The following example shows that the bound in Theorem 9.11 is tight for some problem distributions and some performance measures.

*Example* 5 (ANFL tightness). Let $F = B_h$ be a base class of bijective functions and assume without loss of generality that $X = Y = \{1, \ldots, |X|\}$. Assume $n = |X|$.

Now construct an $\varepsilon$-block uniform distribution P by

$$\mathrm{P}(f) = \begin{cases} p_h + \varepsilon/2 & \text{if } f(n-1) < f(n) \\ p_h - \varepsilon/2 & \text{if } f(n-1) > f(n) \end{cases}$$

where $p_h = 1/|B_h|$ so that $\sum_{f\in F} \mathrm{P}(f) = 1$ (exactly half of the functions receive an $\varepsilon$ increase and the other half an $\varepsilon$ decrease in probability, therefore they sum to 1). Since $f$ is bijective, either $f(n-1) < f(n)$ or $f(n-1) > f(n)$, so the cases are exhaustive. Note that if $B_h$ is large, $\varepsilon$ must to be small in order for $p_h - \varepsilon/2 = 1/|B_h| - \varepsilon/2 \geq 0$.

Define a performance measure $M$ according to

$$M(R) = \begin{cases} 1 & \text{if } R[n-1] < R[n] \\ 0 & \text{if } R[n-1] > R[n] \end{cases} \tag{52}$$

Let $R^< = \{R \in \mathcal{T}_n^y : R[n-1] < R[n]\}$ be the set of result vectors yielding value 1, and let $R^> = \mathcal{T}_n^y - R^<$ be the complement of $R^<$ (that is, the vectors yielding 0).

Finally, define two enumerating search procedures $a$ and $b$ that searches the $n$ first points of $X$ in the order $1, \ldots, n$ and $1, \ldots, n-2, n, n-1$ respectively (both ignoring the output of the function they are searching).

Applying the proof idea of Theorem 9.11 with $H_{a,R} = H_{a,R,F}$, we obtain

$$
\begin{aligned}
&M^{\mathrm{P}}(a,n) - M^{\mathrm{P}}(b,n) = && (53)\\
&= \sum_{f \in B_h} \mathrm{P}(f) M(T_n^y(a,f)) - \sum_{g \in B_h} \mathrm{P}(g) M(T_n^y(b,g)) && (54)\\
&= \sum_{R \in \mathcal{T}_n^y} \Big( \sum_{f \in H_{a,R}} p(f) M(T_n^y(a,f)) - \sum_{g \in H_{b,R}} p(g) M(T_n^y(b,g)) \Big) && (55)\\
&= \sum_{R \in \mathcal{T}_n^y} M(R) \Big( \sum_{f \in H_{a,R}} \mathrm{P}(f) - \sum_{g \in H_{b,R}} \mathrm{P}(g) \Big) && (56)\\
&= \sum_{R \in R^<} M(R) \Big( \sum_{f \in H_{a,R}} \mathrm{P}(f) - \sum_{g \in H_{b,R}} \mathrm{P}(g) \Big) + \\
&\qquad + \sum_{R \in R^>} M(R) \Big( \sum_{f \in H_{a,R}} \mathrm{P}(f) - \sum_{g \in H_{b,R}} \mathrm{P}(g) \Big) && (57)\\
&= \sum_{R \in R^<} 1 \cdot \Big( \sum_{f \in H_{a,R}} \mathrm{P}(f) - \sum_{g \in H_{b,R}} \mathrm{P}(g) \Big) && (58)\\
&= \sum_{R \in R^<} (p_h + \varepsilon/2 - (p_h - \varepsilon/2)) && (59)\\
&= \varepsilon |R^<| = \varepsilon |F|/2 = \varepsilon \sum_{f \in F} M(T_n^y(a,f)) && (60)
\end{aligned}
$$

The steps roughly follow the proof of Theorem 9.11. The differences are: As the function class $F$ now is a base class $B_h$ itself, there is no need to split the sum over base classes. In (58) the fact that $M(R) = 0$ for $R \in R^>$ is used. In (59) we use that the bijectiveness of the the elements of $F$ makes the sets $H_{a,R}$ and $H_{b,R}$ singletons. $\diamondsuit$

In a similar manner one can create a tightness proof for larger $\varepsilon$-block uniform classes of bijective functions. However, given a collection $B_{h_1}, \ldots, B_{h_k}$ for which the bound is tight, it is not necessary that the bound is tight for their union $\bigcup B_{h_i}$. Intuitively, if $B_h$ has a distribution defined as in the above example, and another base class $B_{h'}$ receives the diametric weighting of functions $f \in B_{h'}$:

$$
\mathrm{P}(f) = \begin{cases} p_{h'} + \varepsilon/2 & \text{if } f(n-1) > f(n) \\ p_{h'} - \varepsilon/2 & \text{if } f(n-1) < f(n) \end{cases} \tag{61}
$$

(the directions of the inequality signs are switched compared to (52)). In this case, it may not be possible to construct policies that differ to the maximum extent permitted by the bound.

The bijectiveness is also necessary to make the bound tight. For base classes on which at least two inputs generate the same output, any two procedures $a$ and $b$ will produce the same result vectors on some pair of functions, making the performance difference between $a$ and $b$ smaller than the bound in Theorem 9.11.

Nonetheless, the fact that the bound is (sometimes) tight shows that although the growth of free lunch is continuous, just a small perturbation of the block uniformity may allow one algorithm to widely outperform another.

# 10 Performance measures

So far we have only considered problems for which either NFL holds for all performance measures, or for which a free lunch is available for some performance measures. Often, however, we are interested in performing well under a fixed, *particular* performance measure of interest.

One natural such performance measure is the *probes-till-max*-measure $M_{\text{ptm}}$, which only depends on the number of probes until the maximum is found, and ignores other properties of the result vector. Unfortunately, this performance measure can only be defined on full-length result vectors. We therefore define a *full-length performance measure* as a performance measure defined only on full-length result vectors. $M_{\text{ptm}}$ can then be defined as a full-length performance measure:

**Definition 10.1** (Probes-till-max-measure)**.** The *probes-till-max measure* $M_{\text{ptm}}$ is defined by

$$M_{\text{ptm}}(R) = \min_i(R[i] = \max R) \tag{62}$$

Just as with other performance measures, we will be interested in the P-expected value $M^P_{\text{ptm}}(a) = \sum_R \mathrm{P}_a(R) M_{\text{ptm}}(R)$. Under $M_{\text{ptm}}$ a low score is normally considered better than a high score.

Other performance measures have been considered in the NFL-literature. The $M_{\text{ptm}}$-measure is essentially the performance measure used in [BP06]. A related performance measure based on how many of the first $k$ points exceed a certain threshold (e.g., the median) is used in [CO01, WR06, JC11]. Griffiths and Orponen [GO05] use a *maximization performance measure* $M_{\max}$, defined as:

$$M_{\max}(R) = \max R$$

The $M_{\max}$-measure is closely related to the $M_{\text{ptm}}$-measure. The main reason for preferring $M_{\text{ptm}}$ to $M_{\max}$ is that $M_{\max}$ does not lend itself as nicely to the asymptotic results we will aim for in Section 11. In [WM97], a *minimization performance measure* is discussed; it is also less suitable for asymptotic studies.

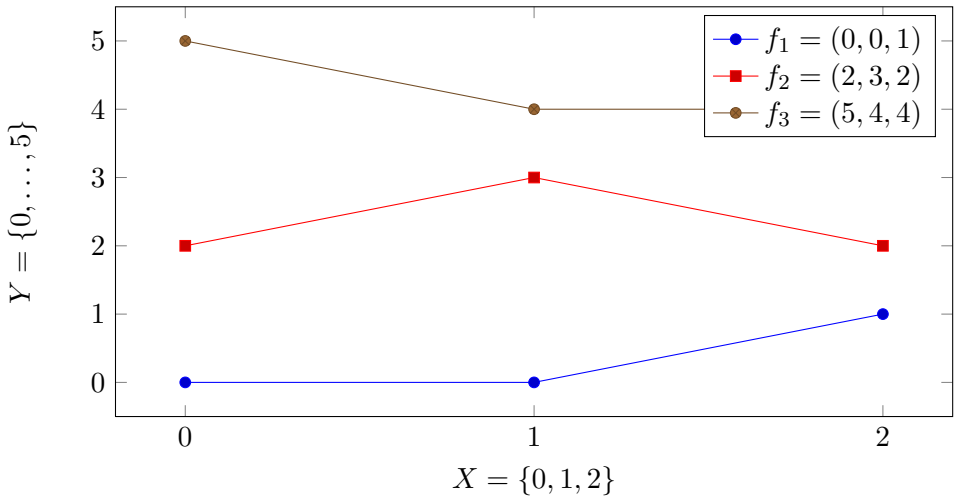


Figure 3: The three functions $f_1 = (0, 0, 1)$, $f_2 = (2, 3, 2)$ and $f_3 = (5, 4, 4)$ used in Example 6. After having searched one point, it becomes obvious which function is the true function. Depending on the performance measure, it may and may not be advantageous to use this information.

### 10.1 Theoretical considerations

Particular performance measures (if well-chosen) have the advantage of being of greater practical interest. In addition, they also add some theoretical dimensions to the NFL-problem, not visible when arbitrary performance measures are allowed.

Griffiths and Orponen [GO05] show that under $M_{\max}$, NFL may hold for classes of functions where NFL does not hold for all performance measures. This does *not* contradict Theorem 9.7, which only claims that *some* performance measure provides a free lunch for the class considered. Indeed, it is unsurprising that NFL will apply to wider ranges of function classes when a fixed performance measure is used. Griffiths and Orponen's conclusion is that classes satisfying NFL for $M_{\max}$ have significantly more intricate descriptions, compared to the standard NFL case.

Another difference is found in the cleverness required to exploit a free lunch. In fact, smarter algorithms may be required for exploiting a free lunch when using a particular performance measure such as $M_{\text{ptm}}$ or $M_{\max}$ compared to when arbitrary performance measures are permitted. This is demonstrated by the following example.

*Example* 6 (Particular versus freely chosen measures). Assume $X = \{1, 2, 3\}$ and that we have the set of functions $F = \{f_1 = (0, 0, 1), f_2 = (2, 3, 2), f_3 = (5, 4, 4)\}$ depicted in Figure 3).[13] Let P be uniform over $F$. Since P is

[13]Recall that the tuple $(y_1, y_2, y_3)$ denotes the function $f(1) = y_1$, $f(2) = y_2$, $f(3) = y_3$.

not block uniform ($F$ is not c.u.p.), a free lunch is available according to Theorem 9.9. Indeed, choosing the performance measure as $M(\langle 0,0,1\rangle) = M(\langle 2,3,2\rangle) = (\langle 5,4,4\rangle) = 1$ and 0 otherwise, makes the enumerating algorithm $e$ that simply searches $X$ in the order $1,2,3$ outperform any other algorithm (it will have optimal $M$-performance).

However, under $M_{\text{ptm}}$, all enumerating algorithms perform equally well, since any point is as likely to be the maximum. But it is possible to construct an algorithm $b$ that outperforms other algorithms by letting it make an *informed choice* (Definition 7.5) after its first probe (that I assume is at $1 \in X$). Let $b$ choose its second search point according to

$$b(\langle 1, f(1)\rangle) = \begin{cases} 3 & \text{if } f(1) = 0 \\ 2 & \text{otherwise; i.e. } f(1) = 2 \text{ or } f(1) = 5 \end{cases}$$

The rationale is: If $f(1) = 0$, then the function must be $(0,0,1)$, so the maximum must be at 3. If instead $f(1) = 2$, then the maximum must be at 2, since the only consistent function is $(2,3,2)$. Finally, if $f(1) = 5$, then that must be the maximum and it does not matter how the search proceeds.

This way $b$ will find the maximum in at most two steps and find the maximum in one step with probability $\frac{1}{3}$. This means that $b$'s expected number of probes-till-max $M^{\text{P}}_{\text{ptm}}(b) = \frac{1}{3}\cdot 1 + \frac{2}{3}\cdot 2 = \frac{5}{3}$. Thereby $b$ outperforms all the enumerating algorithms, as they expect to find the maximum in $\frac{1}{3}\cdot 1 + \frac{1}{3}\cdot 2 + \frac{1}{3}\cdot 3 = 2$ probes. ♢

This example illustrates that "more cleverness" may be required to exploit a free lunch under $M_{\text{ptm}}$ than under arbitrary performance measures. The same example applies with minor modifications to $M_{\max}$ as well, since $b$ will have a higher expected maximum after having searched two points, compared to the enumerating algorithms. So sometimes informed, "clever" choices (i.e., choices depending on the seen $f(x)$-values) are required to exploit a free lunch under $M_{\text{ptm}}$ and $M_{\max}$.

In the above example, it was also the case that when arbitrary performance measures were considered, no cleverness was required. The following proposition shows that this is true in general when arbitrary performance measures are permitted. The proof is essentially a reiteration of the proof of Theorem 9.9, where the enumerating algorithms $e$ and $e_\sigma$ generate the result vector $\langle f(x_1), \ldots, f(x_{|X|})\rangle$ with different probability.

**Proposition 10.2.** *If NFL does not hold for a problem* $\text{P}_{X,Y}$ *on a problem context* $X$,$Y$*, then there is free lunch for an enumerating algorithm* $a$ *(under some performance measure).*

*Proof.* Assume that NFL does not hold for a problem context $X$,$Y$. Then $\text{P}_{X,Y}$ is not block uniform by Theorem 9.9, so there are two functions $f$

and $\sigma f$ in the same base class $B_h$ such that $\mathrm{P}(f) > \mathrm{P}(\sigma f)$, where $\sigma$ is a permutation on $X$. By Lemma 7.4 we had that $e$ generates the result vector $R_f = \langle f(1), \ldots, f(|X|)\rangle$ exactly when $f$ is the true function, and that $e_\sigma$ generates $R_f$ exactly when $\sigma f$ is the true function. An immediate consequence is that $\mathrm{P}_e(R_f) = \mathrm{P}(f) > \mathrm{P}(\sigma f) = \mathrm{P}_{e_\sigma}(R_f)$. That is, the enumerating algorithms $e$ and $e_\sigma$ generate $R_f$ with different probability, which means that there is free lunch for some enumerating algorithm (under some performance measure). □

Indeed, enumerating algorithms can sometimes obtain the "maximum possible amount" of free lunch, as was the case in Example 5.

In conclusion, specific performance measures can be considered for both practical and theoretical reasons. They are more practically relevant in the sense that they measure aspects that we care about in practice (such as how long it takes to find a maximum). But they also have theoretical interest, as they illuminate theoretical subtleties invisible from an arbitrary-measure perspective.

# 11 Universal free lunch

The black-box optimisation problem is to a large extent an induction problem: Given data about how a function behaves on the *particular* points already probed, an algorithm should infer some idea about the *general* or *global* behavior of the function in order to make a good choice for its next search point. As the NFL theorems demonstrate, it is necessary to have some bias rendering some functions more likely than others. Without such a bias (that is, with a uniform or block uniform prior) all generalizations become equally likely, and it becomes impossible to outperform random search or enumeration.

In most cases, we have the intuition that from seeing a number of points we can, in principle, extrapolate the global behavior of the function. Just as in the case of sequence prediction in Part I, it also seems to be the case that we consider the "simplest" continuation of the points as the most likely true function (cf. Occam's razor).

In sequence prediction, the universal measure $\mathbf{M}$ turned out to have excellent induction performance (Theorem 5.11 on page 29). The optimisation problem differs from the sequence prediction task in some important respects. In sequence prediction the next bit is independent of previous guesses. This is not the case in optimisation, where the selection of probing points affects the knowledge. This entails an exploration-exploitation dilemma, as there is sometimes a payoff between points that are likely to be the maximum and points that yield more information. Another difference stem from optimisation being a *finite* setting: Functions $f : X \to Y$ are

naturally represented by (finite) strings, and only a finite number of guesses (probes) are to be made.

The finiteness makes it natural to use the discrete universal measure $\mathbf{m}$ in place of $\mathbf{M}$. The finiteness also entails certain problems with the reference machine. For any finite collection of strings, there is a reference machine that gives all the strings the same complexity, and thus causes $\mathbf{m}$ to give them the same probability. For this reason, all our results will be of the form: For a fixed reference machine there are (sufficiently large) search spaces for which the result holds.

Another difference is that in sequence prediction we could do well in any computable environment. This is not possible in finite settings. Instead the goal will be to do well in $\mathbf{m}$-expectation.

One way to understand the difference between a uniform prior and $\mathbf{m}$ is that $\mathbf{m}$ gives high weight to structured, compressible functions whereas a uniform prior gives high weight to unstructured problems. If a function is sampled from the uniform distribution, then the probability that this function is algorithmically random approaches one as the size of the domain increases.[14] Therefore a uniform prior represents a strong belief in that the true function will be almost entirely without structure, and therefore hard to optimise (cf. Section 5.6.1). In contrast, $\mathbf{m}$ follows Occam's razor in giving higher weight to structured functions.

The main goal of the remainder of the thesis will be an understanding of optimisation under the prior $\mathbf{m}$. The first step will be to adapt the optimisation setting to $\mathbf{m}$. This involves devising string-encodings for problem contexts and functions. It also involves generalizing the definitions for search problem, performance measure and search algorithm, to enable asymptotic studies.

## 11.1 Adapting the optimisation problem

For the remainder of this thesis we will assume that $X$ and $Y$ are ordered, finite subsets of $\mathbb{B}^+$, containing at least 0 and 1 as elements[15] (recall that $\mathbb{B}^+$ is the set of all non-empty, finite strings). Let $\mathcal{X}$ be the set of all such search spaces, and $\mathcal{Y}$ be the set of all ranges.

As the universal distribution requires investigations to be asymptotic, we generalize problems, performance measures and algorithms to be defined on all possible $X$ and $Y$.

Recall that $\mathcal{T}$ denotes the set of all traces between (implicit) spaces $X$ and $Y$. Let $\mathcal{T}(X, Y)$ be the set of traces between $X$,$Y$, making the problem

[14] For any $c \in (0, 1)$, the fraction of $(n - nc)$-compressible strings of length $n$ goes to zero with growing $n$—see Section 3.8 on page 20.

[15] The assumption that both $X$ and $Y$ contain 0 and 1 is only for readability. It is possible to substitute $0 \in X$ for $\min X$ and $0 \in Y$ with $\min Y$, and to substitute 1 for the second smallest elements, with maintained validity of proofs and definitions.

context explicit. Subscripts and superscripts retain their meaning, so, for example $\mathcal{T}_n(X,Y)$ is the set of all search traces of length $n$ on the specified problem context. Let $\mathcal{T}(\mathcal{X},\mathcal{Y}) = \bigcup_{(X,Y)\in\mathcal{X}\times\mathcal{Y}} \mathcal{T}(X,Y)$ be the set of all traces on any context.

A *generalized (optimisation) problem* P is a collection of distributions such that $\mathrm{P}_{XY}$ is a distribution over the set $\{f : X \to Y\}$ for every $(X,Y) \in \mathcal{X}\times\mathcal{Y}$.

Generalized search algorithms receive the problem context as an extra argument. *Deterministic generalized search algorithm* are modelled by functions

$$a : (X,Y,T) \in \mathcal{X}\times\mathcal{Y}\times\mathcal{T}(\mathcal{X},\mathcal{Y}) \mapsto x \in X - T^x \tag{63}$$

Similarly, *probabilistic generalized search algorithms* are modelled by conditional distributions

$$\Pr_b(x \,|\, X,Y,T) \tag{64}$$

where $T \in \mathcal{T}(X,Y)$. As before, we require that no distribution $\Pr_b(\cdot \,|\, X,Y,T)$ has support outside $X - T^x$ (cf. Section 7.2 on page 34).

The full-length trace a deterministic algorithm generates is now determined by the problem context and the function. We write $T(X,Y,a,f)$ for the full-length trace a deterministic algorithm $a$ generates on $X$,$Y$ and $f$. The measures $\Pr_{XYa}(T \,|\, f)$ and $\mathrm{P}_{XYa}(T)$ can be constructed as $\Pr_a$ and $\mathrm{P}_a$ for every $X$,$Y$ (Section 7).

To ease the exposition slightly, and since $M_{\mathrm{ptm}}$ is a full-length measure, in the remainder of this thesis we restrict our attention to full-length result vectors. Let $\mathcal{R}(X,Y)$ be the set of all full-length result vectors on $X,Y$, and let $\mathcal{R} = \bigcup_{(X,Y)\in\mathcal{X}\times\mathcal{Y}} \mathcal{R}(X,Y)$. A *generalized performance measure* is defined by a function $M : \mathcal{X}\times\mathcal{Y}\times\mathcal{R} \to \mathbb{R}^+ \bigcup \{0\}$. As usual, we are interested in the P-expected performance of different algorithms.

$$M^{\mathrm{P}}_{XY}(a) = \sum_{R\in\mathcal{R}(X,Y)} \mathrm{P}_{XYa}(R) M_{XY}(R) \tag{65}$$

For deterministic algorithms, this reduces to

$$M^{\mathrm{P}}_{XY}(a) = \sum_{f:X\to Y} \mathrm{P}_{XY}(f) M_{XY}(T^y(X,Y,a,f)) \tag{66}$$

As a concrete example, the generalised version of $M_{\mathrm{ptm}}$ is defined as

$$M^{P}_{\mathrm{ptm},XY}(a) = \sum_{R\in\mathcal{R}(X,Y)} \mathrm{P}_{XYa}(R) M_{\mathrm{ptm},XY}(R) \tag{67}$$

with

$$M_{\mathrm{ptm},XY}(R) = \min_i (R[i] = \max R) \tag{68}$$

## 11.2 The universal distribution

Using the assumed order on $X = \{x_1, \ldots, x_{|X|}\}$ and $Y = \{y_1, \ldots, y_{|Y|}\}$, it is natural to encode $X$ and $Y$ as $|X|$- and $|Y|$-tuples, according to the code devised in Section 2.4.2 on page 11. Similarly, a function $f : X \to Y$ mapping $f(x_i) = y_i$ is encoded as an $|X|$-tuple $(y_1, \ldots y_{|X|})$.

The string encodings of functions and functions allow us to define the *complexity of a function* $f : X \to Y$. Let $n = |X|$ and $m = |Y|$.

$$K(f\,|\,X,Y) = K(\overline{n}\,\overline{f(x_1)}\ldots\overline{f(x_{n-1})}f(x_n)\,|\,\overline{\overline{n}\,\overline{x_1}\ldots\overline{x_n}\,\overline{m}\,\overline{y_1}\ldots\overline{y_m}}) \tag{69}$$

The complexity of a search point $x \in X$ with respect to $X$,$Y$ is defined as $K(x\,|\,X,Y)$. A function $f : X \to Y$ is *compressible* if $K(f\,|\,X,Y) < |X|\log_2|Y|$ and a point $x \in X$ is *compressible* with respect to $X, Y$ if $K(x\,|\,X,Y) < \log_2|X|$.[16]

Using the complexity of functions, we define the universal probability of a function $f : X \to Y$ as

$$\mathbf{m}_{XY}(f) = c_{\mathbf{m}} \cdot 2^{-K(f\,|\,X,Y)} = \frac{2^{-K(f\,|\,X,Y)}}{\sum_{g:X\to Y} 2^{-K(g\,|\,X,Y)}} \tag{70}$$

The normalisation factor $c_{\mathbf{m}}$ ensures that $\mathbf{m}_{XY}$ is a proper distribution and defines an optimisation problem (cf. the definition of optimisation problem on page 33). It slightly abuses notation compared to the standard definition of $\mathbf{m}$ (Definition 5.5 on page 26). The harm should be minor, as it is still biased towards simplicity in a similar manner, and the dominance property from Proposition 5.7 on page 27 still holds. The main reason the unnormalised version is sometimes preferred is that the normalised version of $\mathbf{m}$ is not semi-computable.

## 11.3 Free lunch under arbitrary measure

Streeter shows that there is free lunch for $\mathbf{m}$ under certain conditions [Str03]. We prove a similar result, but with more easily interpretable conditions (in terms of the size of $X$). We also use a different proof than Streeter.

**Theorem 11.1** (Universal free lunch)**.** *There exists an $n \in \mathbb{N}$ such that there is free lunch for the problem $\mathbf{m}$ for any $X$,$Y$ satisfying $|X| \geq n$.*

*Proof.* It will be shown that $\mathbf{m}$ is not block uniform for problem contexts with sufficiently large $X$, which by Theorem 9.9 implies that NFL does not hold.

[16] Note that a point $x \in X$ may be incompressible with respect to some $X, Y_1$ while being compressible while being compressible with respect to $X, Y_2$. To see this, any point $x \in X$ is generally compressible with respect to $X$,$Y$ with $Y = \{0, 1, x\}$.

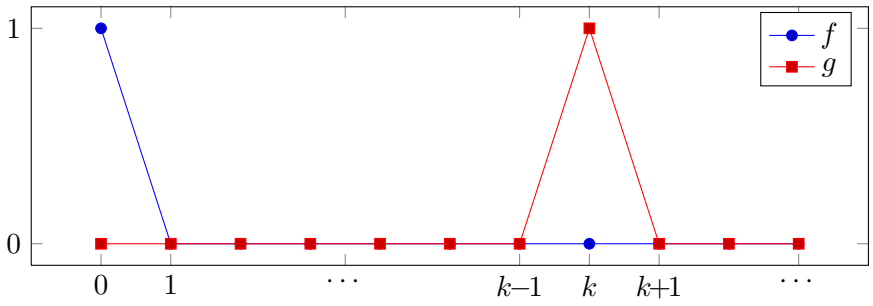


Figure 4: Functions of type $f$ have complexity bounded by a constant $c_f$ independent of $X$ and $Y$. In contrast, the complexity of functions of type $g$ grow logarithmically with $|X|$. See the proof of Theorem 11.1 for details.

Pick an arbitrary problem context $X$,$Y$ and let $X = \{x_1, \ldots, x_n\}$. Consider two functions $f$ and $g$ in the base class $B_h$ of functions with one value 1 and the rest of the values 0. Let $f$ be 1 at $x_1$ and let $g$ be 1 at some point $x_k$ where $k$ is chosen so that $K(g\,|\,X, Y) \geq \log_2 n$. To see that such a $g$ exists, note that there are $n$ different functions in $B_h$. In any prefix-code there are at most $n$ code words of length $\leq \log_2 n$ by Kraft's inequality, since $n2^{-\log_2 n} = 1$ (see e.g. [LV08, p. 76]). Thus at least one of the $B_h$-functions must have a shortest code word longer than $\log_2 n$ in the prefix-code of the reference machine. That is, at least one $B_h$-function $g$ has at least complexity $K(g\,|\,X, Y) \geq \log_2 n$.

Let for every problem context $X$,$Y$ the functions $f_{XY}$ and $g_{XY}$ be functions of "type" $f$ and $g$ above. Then $K(g_{XY}\,|\,X, Y) \geq \log_2(|X|)$. Meanwhile, $K(f_{XY}\,|\,X, Y) \leq c_f$ for some constant $c_f$ independent of the problem context, since there is a program computing $f_{XY}$ for any provided problem context. So for search spaces with $\log_2(|X|) > c_f$, this means that $f_{XY}$ will have lower complexity than $g_{XY}$, and thus that $\mathbf{m}_{XY}$ will assign different probabilities to $f_{XY}$ and $g_{XY}$. But $f_{XY}$ and $g_{XY}$ are elements of the same base class. Therefore $\mathbf{m}$ is not block uniform for sufficiently large search spaces $|X| > 2^{c_f}$. By Theorem 9.9, this implies that there is free lunch for $\mathbf{m}$ under *some* performance measure. □

Indeed, $\mathbf{m}$ is not even $\varepsilon$-block uniform for any reasonably small $\varepsilon$ for large search spaces. This may lead one to suspect that there is a significant amount of free lunch under $\mathbf{m}$ for large search spaces; however, Section 12 below indicates that this is not necessarily the case.

### 11.4 Free lunch under $M_{\text{ptm}}$

As has been discussed, in practice we often care about a particular performance measure. The following definitions and lemmas build up to Theorem

11.5, which investigates whether there is free lunch under the performance measure $M_{\text{ptm}}$.

**Definition 11.2** (Incompressible search points)**.** For any problem context $X$,$Y$, let $\mathcal{D}_{XY} = \{x \in X : K(x\,|\,X,Y) \geq \log_2(|X|)\}$ be the set of all *incompressible search points*. Subsets $D \subseteq \mathcal{D}_{XY}$ of the incompressible search points will be called *incompressibility sets (of $X$,$Y$)*.

The following lemma ensures the existence of incompressibility sets of a certain size. It is a simple reformulation of Proposition 3.9 on page 19.

**Lemma 11.3.** *For any $X$,$Y$ there exists an incompressibility set $D$ of size $|X|/2$ (that is, at least $|X|/2$ points of $X$ are incompressible).*

Functions that only have incompressible maxima (except, possibly, for a maximum at 0) will play an important role since they are guaranteed to have high complexity. The reason for excluding 0 will be apparent in the proof of Theorem 11.5.

**Lemma 11.4.** *If $g : X \to Y$ is such that $\{x : g(x) = \max g\} \subseteq \mathcal{D}_{XY} \cup \{0\}$, then it holds that $K(g\,|\,X,Y) \geq \log_2(|X|) - c$, where $c$ depends only on the reference machine and not on $g$, $X$ or $Y$.*

*Proof.* Let $g$ be a function whose maxima are in $\mathcal{D}_{XY} \cup \{0\}$, and let $x_m \in X - \{0\}$ be the first maximum of $g$ not at 0. Then $x_m$ can be coded by means of $g$ with constant length procedure FirstMax($g$) that computes the first maximum not at 0 for a given function $g$. Hence $K(x_m\,|\,X,Y) \leq K(g\,|\,X,Y) + \ell(\text{FirstMax}) + c$. The constant $c$ depends only on the reference machine, and absorbs the cost of initializing FirstMax with a provided description of $g$.

By assumption, $x_m$ belongs to $\mathcal{D}_{XY}$, so $K(x_m\,|\,X,Y) \geq \log_2|X|$. Combined and rearranged, this gives $K(g\,|\,X,Y) \geq \log_2|X| - \ell(\text{FirstMax}) - c$. The lemma now follows by absorbing $\ell(\text{FirstMax})$ into $c$. □

We are now ready for the main theorem of this section, that there is free lunch for $M_{\text{ptm}}$ on the problem **m**. The key idea is to show that there is a trace after which two unexplored points have different probability of being the maximum.

**Theorem 11.5.** *There is free lunch for the problem* **m** *under the performance measure $M_{\text{ptm}}$ for sufficiently large search spaces.*

*Proof.* The first step of the proof is to construct an event $G$ (defined in a moment) that makes the point 0 likelier to be a maximum than a certain point $x_m \neq 0$ (also defined in a moment).

Fix some $k \geq 2$ and some $X$,$Y$ with $|X| \geq 2k$. Choose some incompressibility set $D_k \subseteq \mathcal{D}_{XY}$ of size $k$. Let $Q = X - D_k - \{0\}$.

The function $f = (1, 0, \ldots, 0)$ has complexity bounded by a constant $c_f$ independent of $X$,$Y$. Accordingly, $f$'s universal probability is always at least $2^{-c_f}$:

$$\mathbf{m}_{XY}(f) \geq 2^{-c_f} \tag{71}$$

Let $G = \{g \in Y^X : x \in Q \implies g(x) = f(x) = 0\}$ be the set of functions $g$ consistent with $f$ on $Q$. Let $x_m$ be the first point in $D_k$ (in the lexicographic order) and let $G_1 = \{g \in G : g(x_m) = \max g\}$ be the subset of functions in $G$ with a maximum at $x_m$. Note that while $\mathbf{m}_{XY}(G) \geq \mathbf{m}_{XY}(f) \geq 2^{-c_f}$, the functions in $G_1$ all have high complexity by Lemma 11.4. Note also that the cardinality of $G_1$ is less than $|Y|^{k+1}$ since the elements of $G_1$ are fixed on $Q$ and $|X - Q| = k + 1$.

Now, a maximum at 0 is likely (is $\mathcal{O}(1)$ with respect to $X$) since the complexity of $f$ is $\mathcal{O}(1)$ with respect to $X$:

$$\mathbf{m}_{XY}(\max \text{ at } 0 \,|\, G) \geq \mathbf{m}_{XY}(f \,|\, G) \geq \mathbf{m}_{XY}(f) \geq 2^{-c_f}$$

A maximum at $x_m$, on the other hand, is less likely since only functions in $G_1$ can have a maximum there:

$$\begin{aligned}
\mathbf{m}_{XY}(\max \text{ at } x_m \,|\, G) &= \mathbf{m}_{XY}(\max \text{ at } x_m, G)/\mathbf{m}_{XY}(G) \\
&= \mathbf{m}_{XY}(G_1)/\mathbf{m}_{XY}(G) \\
&\leq \mathbf{m}_{XY}(G_1) \cdot 2^{c_f} \\
&= c_{\mathbf{m}} \sum_{g \in G_1} 2^{-K(g \,|\, X, Y)} \cdot 2^{c_f}
\end{aligned} \tag{72}$$

Using the lower bound on the complexity from Lemma 11.4, (72) is bounded by

$$\begin{aligned}
&\leq c_{\mathbf{m}} \sum_{g \in G_1} 2^{-\log_2|X| + c} \cdot 2^{c_f} \\
&= c_{\mathbf{m}} |G_1| \cdot 2^{-\log_2|X| + c} \cdot 2^{c_f}
\end{aligned} \tag{73}$$

and since the cardinality of $G_1$ is less than $|Y|^{k+1}$, (73) is bounded by

$$\begin{aligned}
&\leq c_{\mathbf{m}} |Y|^{k+1} 2^{-\log_2|X| + c} \cdot 2^{c_f} \\
&= \frac{c_{\mathbf{m}} |Y|^{k+1} 2^{c + c_f}}{|X|}
\end{aligned} \tag{74}$$

the last equality by elementary simplification.

As (74) goes to 0 with growing search space (and fixed $k$ and $Y$), this shows that for large enough search spaces, 0 is more likely to be the maximum than $x_m$.

Now all that remains is to use this fact to create two algorithms that perform differently under $M_{\text{ptm}}$. Let $a$ start by enumerating $Q$ in order. If the perceived function points are consistent with $f$, it proceeds at 0 and then at $x_m$ and then enumerates the remaining points $X - D_k - 0$. If the trace is not consistent with $f$ it simply enumerates the remaining points. Define $b$ the same way, with the only exception that after $Q$ it searches $x_m$ before 0 in case the trace is consistent with $f$.

This way, $a$ and $b$ will perform the same except when encountering a function in $G$, in which case $a$ will have a strictly better chance of finding the maximum at step $|Q| + 1$. If neither $a$ nor $b$ finds a maximum at step $|Q| + 1$ they will perform the same, since if neither 0 nor $x_m$ is a maximum, then neither $a$ nor $b$ will find a maximum at step $|Q| + 2$ either. Finally, on step $|Q|+3$ and onwards their behavior will again be identical, and therefore also their $M_{\text{ptm}}$ performance.

So $a$ has a strictly better chance at step $|Q|+1$ and $a$ and $b$'s performance is identical on all other steps and in all other situations. This shows that there is a (possibly small) free lunch for $M_{\text{ptm}}$ on $\mathbf{m}$ for sufficiently large search spaces. □

# 12 Upper bounds on universal free lunch

Theorems 11.1 and 11.5 show that there is free lunch under the universal distribution. This section will bound the amount of free lunch available, and show that it is only possible to outperform random search by a constant factor. First we show that the performance of computable search algorithms deteriorates linearly with the worst-case scenario and the size of the search space. This result applies to computable performance measures in general, and has a concrete interpretation for $M_{\text{ptm}}$, where it implies that as the size of the domain is increased, a non-zero fraction of the domain must be probed before a maximum is found in expectation. This result should not be seen as too negative, since for random search the fraction of probes required is approximately half, but for other search algorithms it may be substantially smaller.

We also consider possible ways to circumvent the negative result described above result by means of incomputable search procedures. A further negative result for $M_{\text{ptm}}$ is obtained: It does not appear possible to find the maximum with only $o(|X|)$ probes. That is, the number of probes required to find the maximum in expectation grows linearly with the size of the search space, but again, the proportion may be substantially smaller than that required for a random algorithm.

## 12.1 Computable algorithms

To bound the amount of free lunch available for computable algorithms,

we will adapt a proof-technique for showing that average-case complexity is worst-case complexity under the universal distribution [LV08, Section 4.4].

**Definition 12.1** (Decidable performance measure)**.** A performance measure $M$ is *decidable* if there is an algorithm computing whether $M_{XY}(R_1) < M_{XY}(R_2)$ or not for every $X$,$Y$ and every $R_1, R_2 \in \mathcal{R}(X,Y)$.[17]

Fix a decidable performance measure $M$. Although no formal theorem relies on it, greater $M$-values will generally be assumed to mean worse performance.

**Definition 12.2** (Maximally bad function)**.** $f_{\text{bad}} : X \to Y$ is a *maximally bad* function for a deterministic algorithm $a$ on the problem context $X$,$Y$ with respect to a performance measure $M$ if

$$M_{XY}(T^y(X,Y,a,f_{\text{bad}})) = \max_{R \in \mathcal{R}(X,Y)} M_{XY}(R)$$

**Lemma 12.3.** *Given any performance measure $M$, any algorithm $a$ and any problem context $X$,$Y$, there exists a maximally bad function $f_{\text{bad}} : X \to Y$ for $a$ with respect to $M$.*

*Proof.* Assume there were a problem context $X$,$Y$ on which $a$ never performs maximally badly with respect to some performance measure $M$. Let $R_{\text{bad}}$ be a result vector on $X$,$Y$ such that $M_{XY}(R_{\text{bad}}) = \max_{R \in \mathcal{R}(X,Y)} M_{XY}(R)$, and let $f = R_{\text{bad}}$ (recall the identification of functions with result vectors from Section 7.4). Then the enumerating algorithm $e$ produces the result vector $R_{\text{bad}}$ when $f$ is the true function. By Lemma 9.10, all deterministic algorithms produces $R_{\text{bad}}$ on equally many functions. Hence $a$ must also produce $R_{\text{bad}}$ on some function. □

Define a procedure FindWorst$(a, X, Y)$ that given a computable search algorithm $a$ (specified by some binary string), a search space $X$ and a range $Y$, returns (the tuple for) a maximally bad function $f_{\text{bad},a,X,Y}$ for $a$ and $M$. FindWorst is a computable operation since $a$ is computable and $M$ is decidable: FindWorst need only simulate $a$ on all possible functions in $Y^X$, and output one that yields a worst result vector. FindWorst is computed by some program of some length $\ell$(FindWorst), where $\ell$(FindWorst) is independent of $X$ and $Y$.

This means that the conditional complexity of a maximally bad function for $a$ is $\mathcal{O}(1)$ with respect to $X$ and $Y$, since

$$K(f_{\text{bad},a,X,Y} \,|\, X, Y) \leq \ell(\text{FindWorst}) + \ell(a) + c = \mathcal{O}(1) \qquad (75)$$

[17] Computability of $M$ does not imply decidability. If the values $M_{XY}(R_1)$ and $M_{XY}(R_1)$ were both computable (Definition 4.1 on page 21) it can still be undecidable whether $M_{XY}(R_1) < M_{XY}(R_2)$ or not. Intuitively, this is because if $M_{XY}(R_1) = M_{XY}(R_2)$ we cannot in general be sure that they are actually equal by only approximating them arbitrarily well. See for instance [Wei00] for more details.

where the $c$ term depends only on the reference machine, and absorbs the cost for initializing FINDWORST with $a$, $X$ and $Y$.

Since the complexity of a maximally bad function does not grow with $X$, this means that $a$ will do badly with at least probability $2^{-(\ell(\text{FINDWORST})+\ell(a)+c)}$, no matter how large $X$ grows. That is, for any performance measure $M$, the performance of a computable deterministic algorithm $a$ grows linearly with the performance measure of the worst possible result vector. Formally:

**Theorem 12.4** (Almost NFL for $\mathbf{m}$)**.** *For every decidable performance measure $M$ and every computable algorithm $a$ there exists a constant $c_a > 0$ such that for all and problem contexts $X$,$Y$*

$$M_{XY}^{\mathbf{m}}(a) \geq c_a \max_{R \in \mathcal{R}(X,Y)} M_{XY}(R) \tag{76}$$

*for all $X$ and $Y$.*

*Proof.* Fix a problem context $X$,$Y$. The proof utilizes FINDWORST for a short description of a maximally bad function for $a$.

$$\begin{aligned}
M_{XY}^{\mathbf{m}}(a) &= \sum_{fX \to Y} \mathbf{m}_{XY}(f) M_{XY}(T^y(X,Y,a,f)) \\
&= c_{\mathbf{m}} \sum_{f:X \to Y} 2^{-K(f|X,Y)} M_{XY}(T^y(X,Y,a,f)) \\
&\geq \sum_{f:X \to Y} 2^{-K(f|X,Y)} M_{XY}(T^y(X,Y,a,f)) \\
&\geq 2^{-K(f_{\text{bad},a,X,Y}|X,Y)} M_{XY}(T^y(X,Y,a,f_{\text{bad},a,X,Y})) \\
&\geq c_a \cdot M_{XY}(T^y(X,Y,a,f_{\text{bad},a,X,Y})) \\
&= c_a \max_{R \in \mathcal{R}(X,Y)} M_{XY}(R)
\end{aligned}$$

where $f_{\text{bad},a,X,Y}$ is the output of FINDWORST$(a,X,Y)$ and therefore is maximally bad and has complexity bounded by a constant by inequality (75). □

This theorem shows that for every performance measure $M$, there is only a constant amount of free lunch available in an asymptotic sense. Since the result is asymptotic it has no impact on measures whose value does not grow with $X$. However, the "semi-assumption" of higher values being worse is not necessary: If the converse is the case and high values are better, then the proposition shows that all algorithms will do well. Indeed, this is also an NFL result, as it implies that random search (and even algorithms designed to do poorly!) will perform well.

Applied to the performance measure $M_{\text{ptm}}$, Theorem 12.4 has a fairly concrete interpretation: For any computable deterministic search algorithm $a$, the expected number of probes until a maximum is found grows linearly with $|X|$.

**Corollary 12.5.** *For every computable algorithm $a$ there exists a constant $c_a > 0$ such that $M^{\mathbf{m}}_{\text{ptm},XY}(a) \geq c_a \cdot |X|$ for all algorithms $a$ and all problem contexts $X$,$Y$.*

*Proof.* Let $a$ be a search algorithm. Then there is a function $f_{\text{bad},a,X,Y}$ such that $a$ does not find the maximum of $f_{\text{bad},a,X,Y}$ until the very last probe. Hence $M_{\text{ptm},XY}(T^y(X, Y, a, f_{\text{bad},a,X,Y})) = |X|$. By Theorem 12.4, there is a constant $c_a > 0$ such $M^{\mathbf{m}}_{\text{ptm},XY}(a) > c_a M_{\text{ptm}}(T^y(X, Y, a, f_{\text{bad},a,X,Y})) = c_a \cdot |X|$ for all $X$ and $Y$. That is, the expected number of probes $M^{\mathbf{m}}_{\text{ptm},XY}(a)$ grows linearly with $|X|$. □

The importance of this result should not be overstated. The constant $c_a$ may be very small; for example, if the description of the search algorithm $a$ is 100 bits long, then $c_a$ becomes of the order $2^{-100}$. The fact that an algorithm is always required to search such a fraction to find a maximum is mainly of theoretical importance, since in practice search spaces seldom grow large enough for this fraction to have a meaningful impact. Nonetheless, the result does illustrate the fundamental hardness of optimization, and shows that—in this sense—the universal distribution does not provide enough bias for sublinear maximum finding.

## 12.2 Needle-in-a-haystack functions

A problematic class of functions is the class of so-called *needle-in-a-haystack* (NIAH) functions. For a given $X$ and $Y$, a function $f : X \to Y$ is a NIAH-function if $f$ is taking on the value 0 for all $x \in X$ except one where $f(x)$ equals 1. The exception point is called the *needle*. The class of NIAH functions for a given $X$ and $Y$ is denoted $\text{NIAH}_{XY}$.

It should be intuitively clear that it is hard to find the maximum of a NIAH-function. Probing a NIAH-function, the output will generally just turn out to be 0 and provide no clues to where the needle might be. More formally, the function class $\text{NIAH}_{XY}$ is c.u.p., so NFL holds for the problem $u_{\text{NIAH},XY} = u_{\text{NIAH}_{XY}}$ by Theorem 9.7. The expected performance (of any algorithm) on the NIAH-problem can be calculated from a general result of Igel and Toussaint [IT03a]. They show that for any c.u.p. problem $u_F$ where $F$ only contains functions with exactly $m$ maxima, the expected number of probes to find a maximum is $(|X| + 1)/(m + 1)$. The NIAH-functions have exactly one maximum, which gives the following proposition.

**Proposition 12.6.** *For $u_{\text{NIAH},XY}$ and any algorithm $a$, the expected number of probes until the maximum is found is $M^{u_{\text{NIAH}}}_{\text{ptm},XY}(a) = (|X| + 1)/2$.*

One feature that makes the NIAH-class more problematic than other c.u.p. function classes is that the NIAH-functions all have fairly low complexity (as remarked by [SVW01, BP06]). The NIAH-functions have low complexity, since to encode a NIAH-function one only needs to encode that

it is NIAH (which takes a constant number of bits) and the place of the needle (which requires at most $\mathcal{O}(\log_2|X|)$ bits). A NIAH-function thus has complexity of order $\mathcal{O}(\log_2|X|)$; in comparison, a random function has complexity above $|X|\log_2|Y|$.

The NIAH-measure is also computable. This is intuitively obvious, but we prove it as a lemma to verify it against the formal definition of computable measure given in Definition 5.4 on page 26.

**Lemma 12.7** (Computability of the NIAH-measure)**.** *The* (uniform) NIAH-measure $u_{\mathrm{NIAH},XY}$ *over the NIAH-functions* $X \to Y$ *is a computable measure.*

*Proof.* The $u_{\mathrm{NIAH},XY}$ measure can be formally described as

$$u_{\mathrm{NIAH},XY}(f) = \begin{cases} 1/|X| & \text{if } f \text{ is a NIAH-function } X \to Y \\ 0 & \text{otherwise.} \end{cases}$$

for any $X$ and $Y$

The distribution $u_{\mathrm{NIAH},XY}$ is computable since there is an associated computable function $g(x, y)$ that outputs $1/|X|$ if $y$ is a prefix-code for two spaces $X$ and $Y$, and $x$ is the prefix-code of a NIAH-function between $X$ and $Y$. If these conditions are not satisfied, $g$ just outputs 0. □

The following lemma is an immediate consequence of Proposition 5.7 on page 27, which states that $\mathbf{m}$ dominates any computable measure.

**Lemma 12.8** ($u_{\mathrm{NIAH}}$ is dominated by $\mathbf{m}$)**.** *There is a constant* $c_{\mathrm{NIAH}} > 0$ *such that for all* $X$ *and* $Y$ *and all functions* $f : X \to Y$*,*

$$\mathbf{m}_{XY}(f) \geq c_{\mathrm{NIAH}} \cdot u_{\mathrm{NIAH},XY}(f)$$

## 12.3 Incomputable algorithms

Theorem 12.4, Corollary 12.5 rely on $a$ being describable as a computer program. Intuitively, if $a$ is not computable, then it is not possible to describe it as a program. This means that it is not possible to use (a description of) $a$ for a short description of $f_{\mathrm{bad},a,X,Y}$, which was the trick employed in Theorem 12.4. Incomputable search procedures may seem like remote objects of concern, but for example the (Bayes-)optimal procedure for a problem $\mathbf{m}_{XY}$ is incomputable (due to the incomputability of $\mathbf{m}$). Therefore, incomputable procedures do at least have theoretical interest.

The following theorem generalizes Corollary 12.5 to incomputable search procedures, showing that also they must search a linearly growing portion of $X$ to find the maximum.

**Theorem 12.9** (Almost NFL for $\mathbf{m}$ and $M_{\mathrm{ptm}}$)**.** *Let the problem be* $\mathbf{m}$*. For any search procedure* $a$ *(not necessarily computable), the expected number of probes until a maximum is found grows linearly with* $|X|$*.*

*Proof.* The proof follows from the dominance of $\mathbf{m}$ over $u_{\text{NIAH}}$.

$$M^{\mathbf{m}}_{\text{ptm},XY}(a) = \sum_{R \in \mathcal{R}(X,Y)} \mathbf{m}_{XYa}(R) M_{\text{ptm}}(R) \tag{77}$$

$$= \sum_{\substack{R \in \mathcal{R}(X,Y) \\ f:X \to Y}} \mathbf{m}_{XY}(f) \cdot \Pr_a(R \,|\, f) M_{\text{ptm}}(R) \tag{78}$$

$$\geq c_{\text{NIAH}} \cdot \sum_{\substack{R \in \mathcal{R}(X,Y) \\ f:X \to Y}} u_{\text{NIAH},XY}(f) \cdot \Pr_a(R \,|\, f) M_{\text{ptm}}(R) \tag{79}$$

$$= c_{\text{NIAH}} \cdot M^{u_{\text{NIAH}}}_{\text{ptm},XY}(a) = \frac{c_{\text{NIAH}} \cdot (|X| + 1)}{2} \tag{80}$$

(79) uses the multiplicative dominance of $\mathbf{m}$ over $u_{\text{NIAH}}$ just proved in Lemma 12.8. (80) uses Proposition 12.6 for the expected number of probes until the maximum of a NIAH-function is found. □

Conversely, no algorithm will need to search less than a constant portion of the search space to find the maximum. Theorem 12.9 can thus be interpreted as an asymptotic ANFL theorem for the universal distribution and $M_{\text{ptm}}$: All algorithms perform essentially the same in expectation under the universal distribution and $M_{\text{ptm}}$—up to a constant determining the size of portion. However, the remark from Theorem 12.5 is valid here as well: The difference in constants may be so large that the result has small practical implications.

# 13 Concluding remarks

## 13.1 Summary

To summarize the results of Part II, we started out by giving an account of the classical NFL theorems, which culminated in a precise condition for NFL (Theorem 9.9). We then investigated the potential growth of free lunch as the problem distribution moved away from block uniformity, and found that the growth was continuous but rapid (Section 9.4).

We also argued that in many cases the real quantity of interest is free lunch under some particular measure. $M_{\text{ptm}}$ was defined to be a performance measure quantifying the expected number of probes until a maximum was found.

Based on Part I and the characterization of optimisation as an induction problem, we set out to investigate whether the universal distribution could be used as a universal optimisation bias. The conclusion was that given the right formal setup, the universal distribution does feature a free lunch. In

particular, there was free lunch under $M_{\text{ptm}}$. However, although the universal distribution diverged significantly from block uniformity, asymptotic ANFL theorems could still be obtained (Theorem 12.4 and 12.9).

## 13.2 Optimisation and sequence prediction

The asymmetry between the optimisation problem and the sequence prediction task described in Section 5.6 on page 28 is striking. In sequence prediction, a predictor based on $\mathbf{M}$ widely outperforms a random predictor (a random predictor would have infinite expected prediction-distance); in contrast, the difference between an optimal and a random searcher was limited in optimisation.

One way to understand the difference is that optimisation is a finite setting. Intuitively, a sequence predictor may spend an unbounded but finite initial period on learning the true model, and then exploit this model indefinitely. This is not possible in optimisation.

The NIAH-functions used in Theorem 12.9 were problematic in optimisation because they had comparatively simple descriptions, but were hard to predict because they had hard-coded exception points. In sequence prediction this cannot happen. If an environment had hard-coded exception points, there would be a rule (pattern) to where the exceptions occurred. And it would only take the predictor a finite amount of time to find this pattern. Once the pattern had been found, the exception points would be perfectly predictable.

Adversarial functions of the type $f_{\text{bad},a,X,Y}$ used in Section 12.1 were another reason the free lunch was limited under the universal distribution. The incomputability of the $\mathbf{M}$-predictor allows it to perform well on all computable environments, as the incomputability makes it impossible to construct a computable adversarial environment. Computable approximations of the $\mathbf{M}$-predictor are vulnerable to adversarial environments, however.

## 13.3 Future research

In conclusion, the universal distribution does not provide sufficient bias for good (theoretical) optimisation performance. This naturally raises the question of whether there is some stronger bias, which is still universal in the sense that it is not biased towards a specific type of problem.

One possibility is to restrict the class of functions to exclude particularly troubling function classes (such as the NIAH-functions). Applying the universal distribution to such a restricted class of functions may potentially yield a principled, significant free lunch for optimisation.

The KC-continuity briefly discussed in Section 8.3 offers one such possible restriction. Quite possibly, however, a stronger criteria may be needed. Other possibilities include restring the function class to Lipschitz-continuous

functions, or to "simple" permutations of clearly searchable function-classes such as polynomials. Combinations of several criteria is also a possibility.

The adversarial functions might potentially be avoided by punishing running time in the prior. The complexity of a string $s$ may instead be defined along the lines of

$$Kt_U(s) = \min_w \{\ell(w) + \log t_U(q) : U(w) = s\}$$

where $t_U(q)$ is the running time of $q$. A universal "speed" prior based on $Kt$ was suggested by Schmidhuber [Sch02]. Given that maximally bad functions were sufficiently time-consuming to compute in comparison with other functions, this could be one way to avoid the issue with adversarial functions.

Finally, it would be instructive to investigate generalisations to infinite search spaces as in [AT07]. This would make the setting more similar to sequence prediction. One major difference would remain, however: In sequence prediction the outcome of the next bit is independent of previous guesses. This is not the case in optimisation, where the selection of probing points affects the knowledge.

# Appendices

## A Proofs

*Proof of Theorem 9.5.* The uniform distribution over the class of all functions is block uniform (Definition 9.8 on page 46). Therefore the theorem is a special case of Theorem 9.9. □

*Proof of Theorem 9.7.* The uniform distribution over a c.u.p.-class of functions is block uniform (Definition 9.8 on page 46). Therefore the theorem is a special case of Theorem 9.9. □

## B Lists of notation

### B.1 Abbreviations

| | |
|---|---|
| c.u.p. | closed under permutation |
| NFL | No Free Lunch |
| ANFL | Almost No Free Lunch |
| NIAH | Needle-in-a-haystack |

### B.2 Generic math notation

| | |
|---|---|
| $c, r$ | real-valued constants |
| $m, n, k$ | integer-valued constants |
| $i, j$ | indices |
| $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{R}$ | the sets of integer, rational and real numbers |
| $\mathbb{R}^+$ | the set of positive real numbers |
| $\mathcal{O}, \Theta, \Omega$ | big- $\mathcal{O}$, Theta and Omega notation |

### B.3 Kolmogorov complexity notation

| | |
|---|---|
| $\mathbb{B}$ | the set of the binary symbols $\{0, 1\}$ |
| $\mathbb{B}^*$ | the set of all finite binary strings |
| $\mathbb{B}^+$ | the set of all non-empty finite binary strings |
| $\mathbb{B}^n$ | the set of all finite binary strings of length $n$ |
| $\mathbb{B}^\infty$ | the set of all infinite binary sequences |
| $s, t, q$ | strings |
| $z$ | infinite sequence |

| | |
|---|---|
| $w$ | code-word |
| $q$ | provided information |
| $C$ | code |
| $\mathcal{C}$ | class of codes |
| $V$ | prefix-machine |
| $U$ | universal prefix-machine |
| $\ell$ | length of a string |
| $K_V$ | description length w.r.t. $V$ |
| $K$ | Kolmogorov complexity |

## B.4 Probability theory notation

| | |
|---|---|
| $\Omega$ | Sample space |
| $\Sigma$ | $\sigma$-algebra |
| $A, B$ | events |
| $\lambda, \nu, \rho$ | measures |
| $\mu, L$ | discrete and continuous Lebesgue-measure |
| $\mathbf{m}, \mathbf{M}$ | discrete and continuous universal distribution |
| $D_n$ | expected prediction-distance of prediction $n$ |

## B.5 No free lunch notation

| | |
|---|---|
| $X, Y$ | search space and co-domain, problem context |
| $x, y$ | members of $X$ and $Y$ respectively |
| $x^k, y^k$ | random variables for the $k$th search point and perceived value |
| $f, g$ | functions |
| $f_{\mathrm{bad},a,X,Y}$ | function on which $a$ does maximally badly |
| $F, G, H$ | sets of functions |
| $h$ | histogram |
| $B_h$ | the set of functions with histogram $h$ |
| $\sigma$ | permutation |
| $\Pi$ | set of permutations |
| $V[i{:}j], V[i]$ | extracts the $i$th to $j$th elements and the $i$th element, respectively |
| $T, S$ | search traces |
| $T_n$ | search trace of length $n$ |
| $Q, T_n^x$ | probing sequences, the $X$-components of $T_n$ |
| $R, T_n^y$ | result vectors, the $Y$-components of $T_n$ |

| | |
|---|---|
| $\mathcal{T}_n, \mathcal{T}_n^x, \mathcal{T}_n^y$ | the sets of all $T_n$, $T_n^x$ and $T_n^y$'s respectively |
| $\mathcal{T}(X,Y)$ | the set of traces for the problem context $X$,$Y$ |
| $\mathcal{R}, \mathcal{R}(X,Y)$ | the set of full-length result vectors (on $X$,$Y$) |
| $a, b$ | search algorithms |
| $\mathcal{A}$ | the set of deterministic search algorithms |
| $e, e_\sigma$ | enumerating search procedures |
| $T_n(a,f)$ | the trace of length $n$ that $a$ generates on $f$ |
| P | search problem (probability distribution over $Y^X$) |
| $\mathrm{Pr}_a$ | probability for algorithms |
| $\mathrm{P}_a$ | distribution over traces |
| $u_F$ | the uniform distribution over set of functions $F$ |
| $\mathrm{NIAH}_{X,Y}$ | the class of needle-in-a-haystack functions |
| $u_{\mathrm{NIAH},XY}$ | the (uniform) NIAH-problem |
| $\mathbf{m}_{XY}$ | the universal distribution-search problem |
| $\mathbf{m}_{XYa}$ | the trace-measure induced by $\mathbf{m}_{XY}$ and $a$ |
| $M$ | performance measures of search algorithms |
| $M_{XY}$ | generalized performance measure |
| $M_{\mathrm{ptm}}$ | the performance measure *probes-till-max found* |

# References


[AT07] Anne Auger and Olivier Teytaud. Continuous lunches are free! In *Proceedings of the 9th annual conference on Genetic and evolutionary computation GECCO'07*, 2007.

[BP06] Yossi Borenstein and Riccardo Poli. Kolmogorov complexity, optimization and hardness. In *Proceedings of the IEEE Congress on Evolutionary Computation CEC'06*, pages 112–119. IEEE, 2006.

[CK03] David Corne and Joshua Knowles. Some multiobjective optimizers are better than others. *Proceedings of the IEEE Congress on Evolutionary Computation CEC'03*, 4:2506–2512, 2003.

[CO01] Steffen Christensen and Franz Oppacher. What can we learn from no free lunch? a first attempt to characterize the concept of a searchable function. In Lee Spector, editor, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO'01*, pages 1219–1226, San Fransisco, 2001.

[Cut80] Nigel Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.

[DJW02] Stefan Droste, Thomas Jansen, and Ingo Wegener. Optimization with randomized search heuristics – the (A)NFL Theorem, realistic scenarios, and difficult functions. *Theoretical Computer Science*, 287(1):131–144, 2002.

[DJW04] Stefan Droste, Thomas Jansen, and Ingo Wegener. Upper and lower bounds for randomized search heuristics in black-box optimization. Technical Report CI-162/04, University of Dortmund, 2004.

[EL13] Tom Everitt and Tor Lattimore. Universal induction and optimisation: No free lunch? Submitted 2013.

[Fri70] Avner Friedman. *Foundations of modern analysis*. Courier Dover Publications, 1970.

[GO05] Evan J Griffiths and Pekka Orponen. Optimisation , block designs and no free lunch theorems. *Information Processing Letters*, 94(2):55–61, 2005.

[Hut05] Marcus Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability*. Lecture Notes in Artificial Intelligence (LNAI 2167). Springer, 2005.

[IT01] Christian Igel and Marc Toussaint. On classes of functions for which no free lunch results hold. *Information Processing Letters*, 86(6):317–321, 2001.

[IT03a] Christian Igel and Marc Toussaint. Neutrality and self-adaptation. *Natural Computing*, 2(2):117–132, 2003.

[IT03b] Christian Igel and Marc Toussaint. Recent results on no-free-lunch theorems for optimization. *arXiv preprint cs/0303032*, 2003.

[IT04] Christian Igel and Marc Toussaint. A no-free-lunch theorem for non-uniform distributions of target functions. *Journal of Mathematical Modelling and Algorithms*, 3:312–322, March 2004.

[JC11] Pei Jiang and Ying-ping Chen. Free lunches on the discrete lipschitz class. *Theoretical Computer Science*, 412(17):1614–1628, April 2011.

[LH11] Tor Lattimore and Marcus Hutter. No free lunch versus occam's razor in supervised learning. In *Proceedings of the Solomonoff 85th Memorial Conference*, Melbourne, Australia, November 2011. Springer.

[LV08] Ming Li and Paul Vitanyi. *Kolmogorov Complexity and its Applications*. Springer Verlag, third edition, 2008.

[Mar09] Stephen Marsland. *Machine Learning: An Algorithmic Perscpective*. Chapman & Hall, 2009.

[McG06] Simon McGregor. No free lunch and algorithmic randomness. In *GECCO'06*, pages 2–4. ACM, 2006.

[MH09] James A R Marshall and Thomas G Hinton. Beyond no free lunch: realistic algorithms for arbitrary problem classes. *CoRR*, abs/0907.1:1–9, 2009.

[Mue06] Markus Mueller. Stationary algorithmic probability. *Theoretical Computer Science*, 2(1):13, 2006.

[Res98] Sidney I Resnick. *A probability path*. Springer, 1998.

[RH11] Samuel Rathmanner and Marcus Hutter. A philosophical treatise of universal induction. *Entropy*, 13(6):1076–1136, 2011.

[RS95] Nicholas J Radcliffe and Patrick D Surry. Fundamental limitations on search algorithms: Evolutionary computing in perspective. *Computer Science Today*, pages 275–291, 1995.

[RVW09] Jonathan E Rowe, Michael D Vose, and Alden H Wright. Reinterpreting no free lunch. *Evolutionary computation*, 17(1):117–129, January 2009.

[Sch94] Cullen Schaffer. A conservation law for generalization performance. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 259–265. Morgan Kaufmann, 1994.

[Sch02] Jürgen Schmidhuber. The speed prior: A new simplicity measure yielding near-optimal computable predictions. In *Proceedings of the 15th Annual Conference on Computational Learning Theory COLT 2002*, volume 2375 of *Lecture Notes in Artificial Intelligence*, pages 216–228. Springer, 2002.

[Sol64a] Ray J Solomonoff. A formal theory of inductive inference. Part I. *Information and Control*, 7(1):1–22, 1964.

[Sol64b] Ray J Solomonoff. A formal theory of inductive inference. Part II. Applications of the systems to various problems in induction. *Information and Control*, 7(2):224–254, 1964.

[Sol78] Ray J Solomonoff. Complexity-based induction systems: Comparisons and convergence theorems. *IEEE Transactions on Information Theory*, IT-24(4):422–432, 1978.

[Str03] Matthew J Streeter. Two broad classes of functions for which a no free lunch result does not hold. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO'03*, pages 1418–1430, 2003.

[SVW01] Christopher W Schumacher, Michael D Vose, and L Darrell Whitley. The no free lunch and problem description length. In Lee Spector, editor, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO'01*, pages 565–570. Morgan Kaufmann, 2001.

[WB69] Satosi Watanabe and D Boulton. *Knowing and guessing; a quantative study of inference of information.* Wiley, New York, 1969.

[Wei00] Klaus Weihrauch. *Computable Analysis: An Introduction.* Springer, 2000.

[WM95] David H Wolpert and William G Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

[WM97] David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):270–283, 1997.

[WR06] L Darrell Whitley and Jonathan E Rowe. Subthreshold-seeking local search. *Theoretical Computer Science*, 361(1):2–17, August 2006.

[WR08] L Darrell Whitley and Jonathan E Rowe. Focused no free lunch theorems. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation GECCO'08*, page 811, New York, New York, USA, 2008. ACM Press.