The rationale of meta-heuristics

Contents

1 Preface

In these notes the basic concepts belonging to the theory of computability and the complexity of algorithmic problems will be discussed. These are key concepts for understanding the rationale of heuristics. In fact heuristic methods for solving optimisation problems in Operations Research are developed in direct response to the mere fact that typical real life problems cannot be solved in centuries even millenia by the fastest computers if exact solutions are to be found. Therefore, it is of the utmost importance to have means for estimating the complexity of algorithms. Usually if a problem is NP-complete, then heuristic methods are called for.

These notes will explain the concepts on which these estimates of complexity are based.

We have not aimed at being encyclopaedic in this regard; the emphasis was rather that of providing the reader with useful techniques and concepts. The ideas are beautiful and of a wide scope. The author hopes the reader will enjoy this encounter with some of the most striking conceptual developments of the previous century. The subject is still under intense development and many problems, whether it be in engineering, philosophy, mathematics, biology and computer science, need to be resolved for the understanding of the complexity of algorithmic problems.

2 Introduction

Algorithms and computability constitute the key concepts of computer science. An algorithm is a precise recipe for performing some task, such as the elementary algorithm for adding two numbers we all were taught as children. The first part of these notes will deal with the theory of algorithms as developed by metamathematicians such as Gödel, Turing, Church and Kleene which later became the cornerstone of computer science.

The initial aim of this theory was to reach an understanding of the formalisability of mathematical thought. Out of this endeavour there crystallised a concept of computabilty and algorithms. The design of computers and its software was deeply influenced by these developments. The second part of these notes will deal with computational complexity, which is the study of the amount of resources (time and space) required for performing a given computational task.

Finally, we shall discuss the so-called No Free Lunch theorem in heuristics.

The discussion will be interspersed with exercises. These exercises are then collected at the end of the notes as an assignment in Section13. The reader is advised to start working on the exercises as they appear in the notes and then to collect them together for the final assignment.

3 Computability

The starting point for our discussion is the informal notion of a *computable* function, as well as the closely related informal notion of a *decidable* problem.

Informally, we call a function f *(effectively) computable* when there exists an *effective procedure* or *algorithm* (for us the two terms will be synonymous) that produces the value of f correctly for each possible argument.

We assume that an algorithm operates on a well defined formal structure (vectors, matrices, trees and other data structures) and that each such structure contains numerical values and symbols.

The algorithm determines a computation which depends on the *input*, and a *computation* is a sequence, finite or infinite, of configurations.

A *configuration* is simply a formal structure with a number of numerical values and symbols. Essentially, the algorithm determines how a configuration occurring in the computation changes to the next configuration.

More precisely, the algorithm consists of a finite number of *computational rules* which prescribe in a deterministic way which is the next configuration after any given configuration. Furthermore, the algorithm determines in which conditions or states the computation halts, and which is the output of the computation.

The algorithm computes the function f if the input x, which is a code for the initial configuration, leads to a halting state with output $f(x)$. As we shall see in the course of our discussion, that for many purposes we may assume that both x and $f(x)$ are nonnegative integers or finite sequences of nonnegative integers.

We consider functions $f(x_1, \ldots, x_k),\ k \geq 1$, the variables x_i of which range over nonnegative integers and which assume nonnegative integers as values. For our purposes, we may restrict the domain and range of f in this fashion; this situation can always be attained by a suitable encoding of the objects (configurations) we are considering. This will become clear in the course of our discussion.

We also allow the possibility of the function f being *partial*: the domain of f does not necessarily consist of *all* k-tuples $(x_1, \ldots, x_k)$ of nonnegative integers - the value of $f(x_1, \ldots, x_k)$ may be undefined for some arguments $(x_1, \ldots, x_k)$. The term *total* is used for functions which is defined for all arguments $(x_1, \ldots, x_k)$.

In accordance with the preceding discussion, an effective procedure A_f for computing f must meet the following two conditions:

1. The description of A_f must be finite in length and sufficiently detailed so that the sequence of configuration changes is uniquely determined by the input. The execution of A_f is uniquely and unambiguously determined by the description of A_f and the input.

2. If the k-tuple $(x_1, \ldots, x_k)$ is in the domain of f then, after the execution of finitely many discrete computation steps, the procedure A_f halts and produces the value $f(x_1, \ldots, x_k)$. If $(x_1, \ldots, x_k)$ is not in the domain of f, then A_f may produce an answer indicating that the k-tuple is not in the domain of f, get stuck at some stage, or might run forever, never halting. However, A_f NEVER stops while producing a nonnegative answer as an alleged value of $f(x_1, \ldots, x_k)$ when $(x_1, \ldots, x_k)$ is not in the domain of f.

The execution of A_f may be depicted as a computer, with an unbounded memory and no time restriction, following the instructions of a program.

The restrictions imposed on computable functions do not entail any restrictions concerning the efficiency of computations: effectivity should be distinguished from efficiency. Thus, although we require that the effective procedure A_f produces the output after finitely many steps, no upper bound is placed in advance on the number of steps before the halting state is reached. Similarly, no upper bound is imposed on the the amount of memory space needed to carry out the procedure for a given argument. Issues concerning time and space requirements for the execution of algorithms is what the theory *computational complexity* is about.

Example

All basic arithmetic functions are effectively computable. For instance, let $f(0) = 1$ and let $f(x),\ x \geq 1$, be the xth prime number in order of magnitude. Clearly, the primality of any natural number n can be effectively tested. An effective procedure for computing $f(x)$ consists of the following recursion: Initially, we know that $f(1) = 2$. If we have already computed $f(x)$, then $f(x + 1)$ is the first prime number in the sequence

$$f(x) + 1, f(x) + 2, \ldots .$$

The process will always halt since there are infinitely many prime numbers (why?). This means that f is a total function (it has a value for every input x. Obviously, this procedure can be described in any chosen programming language. This process is effective but at present no efficient process is known for computing this particular function f.

Exercises

1. Show that if $p_1, \ldots, p_n$ are the first n prime numbers then

   $$p_1 \cdots p_n + 1$$

   is divisible by a prime numbers which is different from $p_1, \ldots, p_n$. Deduce that there are infinitely many prime numbers.

2. Let Prime(x) be the Boolean-valued predicate which outputs *true* when x is a prime number, and *false* otherwise. Give an informal description of an effective procedure for computing Prime(x) for a given input x.

3. The function $f(x)$ yielding the xth prime number can be described as follows: $f(1) = 2$ and

   $$f(x + 1) = \min[y > f(x) : \text{Prime}(y)].$$

   This is a recursive (self-referential) description because the computation of $f(x + 1)$ calls on the value of the *same* function f at the the value x. Design a programme for computing $f(x)$ which is not self-referential in this sense.

4. (Easy) What is the domain of the function computed by the following algorithm? In other words, for which inputs will the program halt?

```
**input** $i \geq 0$
**begin**
        **repeat**
            $i := i + 1$
        **until** $i = 0$
**end**
```

4 Set theory and Infinity

Set theory provides a very rich framework in which almost all mathematical and computational activity can find its expression. The underlying idea is of entities, called sets, which have elements and of a set being uniquely determined by its elements, i.e., two sets are the same if and only if they have the same elements. This is not as obvious as it might appear after a first reading. It makes good mathematical sense to consider extensions which are not describable in terms of irreducible entities, such as elements, or points.

It is great fun to explore how mathematical entities can be represented by sets. There is a set, the empty set, that has no elements. This set is denoted by $\emptyset$. Let us now attempt to represent every natural number, which we view as an intuitive notion, as a set. Well, let 0 be the empty set, let 1 be the set $\{\emptyset\}$. Note that by this definition, 1, viewed as a set, does indeed have one element only. From now on, we identify 1 with $\{\emptyset\}$. Having done this, we identify 2 with the set $\{0,1\} = \{\emptyset, \{\emptyset\}\}$, next

$$3 = \{0,1,2\} = \{\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}\}$$

and so on. Within this set-theoretic way of looking at the natural numbers, it is the case that

$$n + 1 = n \cup \{n\}.$$

(The reader is invited to convince herself of the correctness of this statement.) This is an example of how an intuitive notion, a natural number, can be represented as a set. This is also an example of a general tendency in many parts of mathematics: to view sets as fundamental, and to express all mathematical entities as sets.

There is a system of axioms for set theory. This system is usually referred to as the ZFC-system (Z,F,C stand for Zermelo, Fraenkel, Choice, respectively.) The rules of the game is that any set-theoretic statement is a theorem provided it is a logical consequence of these axioms. These axioms are not inevitable laws of mathematical thought: their construction was the result of a careful introspection on the nature of set-theoretic thinking. Moreover, the acceptance of this system as a foundation of mathematics is not a universal one. In fact, not all mathematicians are of the opinion that mathematics is in need of a foundation.

Towards the end of the nineteenth century, the German mathematician Georg Cantor, pioneered the idea of working directly with infinite sets and to generalise the logic of finiteness to infinite mathematical entities. His ideas had an immense impact on the mathematics, science and technology of the twentieth century. As examples we mention that the removal of "noise" from electronic circuits and optical channels and the predictions of the fluctuations on the the stock exchange are solidly based on the ideas of Cantor. It is nowadays unthinkable to do quantum theory or probability theory without invoking the consequences of Cantor's work. Moreover, the philosophical scrutiny of Cantor's notion of infinity still is an important, and fascinating, activity.

Two sets X, Y are said to have the same cardinality, if their is an association, or function, that associates, with every $x \in X$, exactly one $y \in Y$, and, for every $y \in Y$, there is exactly one $x \in X$ such that y is associated with x. Such an association is called a *bijection* between X and Y.

Intuitively, two sets have the same cardinality if they have the "same number of elements". For example write ω for the set $\{0, 1, 2, \cdots\}$ of non-negative integers and let ω_e be the set $\{0, 2, 4, 6, \cdots\}$

of even numbers. The sets ω and ω_e have the same cardinality as can be seen from the following bijection:

$$0 \leftrightarrow 0, 1 \leftrightarrow 2, \cdots, n \leftrightarrow 2n, \cdots.$$

It follows that a set X can have the same cardinality as some proper subset thereof. This can certainly not happen when X is finite. (Why?)

In fact, one definition of an infinite set is that it contains a proper subset which has the same cardinality as the set itself. This definition of infinity goes back to Galileo.

A set which has the same cardinality as ω, the set of non-negative integers, is said to be a countably infinite set. Intuitively, a set X is countably infinite, if its elements can be listed as a sequence $x_1, x_2, \cdots$ in which no repetition occurs.

We claim that the set of rational numbers is countably infinite: As a first step towards a proof, we show that the set of positive rational numbers is countably infinite. Each positive rational number is uniquely expressible in the form p/q, where $p, q \geq 1$ and where p and q have no common divisors. Associate with every such rational number the natural number $2^p 3^q$. In this way, we have associated with every positive rational x a natural number $n(x)$, and the association is one-to-one: this means that if $x \neq y$, then $n(x) \neq n(y)$. List the natural numbers $n(x)$ as an increasing sequence $n(x_1) < n(x_2) < \ldots < n(x_j) < \ldots$. Then $x_1, x_2, \cdots, x_j, \cdots$ is a listing of the positive rationals. The set of all rational numbers can now be listed as follows:

$$0, -x_1, x_1, -x_2, x_2, \cdots, -x_j, x_j, \cdots.$$

Let us consider the following question: Do all infinite sets have the same cardinality? With every set X one can associate its *power set*, $P(X)$, which is the set of all subsets of X. (By definition, the empty set is a subset of every set X.) We now show that, for every set X, there is no bijection between X and $P(X)$, i.e., the cardinality of X and $P(X)$ are different:

Suppose that the cardinality of X and $P(X)$ were the same. Then there is a bijection f, say, between X and $P(X)$. This means that f associates with every $x \in X$ some unique subset, $f(x)$, of X and if A is a subset of X, then there is a unique element $x \in X$ such that A is associated with x under f, i.e. that $A = f(x)$. Let A be the following subset of X:

$$A = \{x \in X : x \notin f(x)\}.$$

Since A is a subset of X, there is some element x of X such that $f(x) = A$. Now ask the question: Is $x \in f(x)$? Well, if $x \in f(x)$, then $x \in A$ and, by the construction of A, we must conclude that $x \notin f(x)$. So, $x \notin f(x)$, i.e., $x \notin A$ since $A = f(X)$, and, again by the construction of A, we have that $x \in f(x)$. This means that neither the assumption $x \in f(x)$, nor its negation, $x \notin f(x)$, can be the case. This is, from the point of view of classical logic, an untenable situation. This state of affairs came from an assumption: namely that there is a bijection f between X and $P(X)$. We logically conclude that this cannot be the case. Hence the sets $X, P(X)$ do not have the same cardinality.

We can interpret this result as stating that for any infinite set X, the sets X and $P(X)$ represent different levels of infinity: $P(X)$ is "more infinite" that X. Now set $X_1 = \omega$, the set of non-negative integers, $X_2 = P(X_1)$, ... , $X_{n+1} = P(X_n)$, ... Then the sequence of sets $(X_n : n \geq 1)$ represents an infinite array of different levels of infinity.

An infinite binary sequence is a pattern $x_0 x_1 x_2 \ldots$, where every x_j is either zero or one. The set of all binary sequences is usually denoted by 2^ω. We claim that the sets $P(\omega)$ and 2^ω have the same cardinality. We associate with every subset A, an infinite binary sequence $x_1x_2\ldots$ such that, for every $j \geq 1$, it is the case that $x_j = 1$ when $j \in A$ and $x_j = 0$ when $j \notin A$. For example, if $A = \omega_e$, the set of even numbers, the associated infinite binary sequence is $101010101\ldots$. It is quite easily seen that this association is bijective. We can conclude that 2^ω is not a countably infinite set, for we have already shown that there is no bijection between ω and $P(\omega)$.

We leave it to the reader to convince him-/herself that we can now conclude that the set of real numbers is not countably infinite. (Hint: Think of the expansion of a real as a binary sequence.)

Cantor asked the following question: Is there an infinite subset X of the real numbers, which is not countably infinite and at the same time does not have the same cardinality as the set of real numbers? In other words, is there a level of infinity strictly between the countably infinite and the level of infinity of the reals? In fact he conjectured that there is no such set. This conjecture is known as the continuum hypothesis (CH). The answer to this question turned out to be very interesting. It is that, from the framework of the ZFC axiomatisation of set theory, both CH and its negation $\neg CH$ are *possible*. If ZFC is consistent (free of contradiction), there is a model of ZFC in which CH is true (Gödel) and there is a model of ZFC in which $\neg CH$ is true (Cohen). This implies that neither CH nor $\neg CH$ is a logical consequence of ZFC ... unless ZFC is inconsistent. In this sense, the statement, CH, is independent of ZFC. What is interesting, is that this statement "CH is independent of ZFC" is a logical consequence of ZFC but the statement "ZFC is consistent" can never be proven within ZFC.

The preceding paragraph has been a brisk introduction to the world of metamathematics. To some it is a paradise, to others, a schizophrenic nightmare.

Exercises

1. Show that every subset of a countably infinite set is either finite or countably infinite.

2. Show that the set consisting of all finite subsets of the natural numbers is a countably infinite set.

3. Show that all (total) functions on the nonnegative integers ω assuming values in ω is not countably infinite.

4. Give an informal argument for establishing that all total computable functions is countably infinite. Conclude that not all functions (in the mathematical sense) on ω taking values in ω are computable functions. (We shall later come across "concrete" examples of non-computable functions.)

5 Decidability

A decision problem Π is a problem which has either one of two answers: *true* or *false*. With each such Π, we can associate a set $S = S(\Pi)$ which is the set of all inputs for which the answer is *true*. With such Π one can also associate a function f, which is the characteristic function of $S(\pi)$,i.e., $f(x) = 1$ if $x \in S$ and $f(x) = 0$, otherwise. For example, if Π is the problem whether a nonnegative integer is a prime number, then the corresponding S is the set of prime numbers and the associated f is the function on ω with values in $\{0, 1\}$ which assumes the value 1 exactly on the set of prime numbers. We say that the decision problem Π is *decidable* if and only if the corresponding characteristic function f is a total and effectively computable function.

If the set S of true instances of the problem is not a set of nonnegative integers, it must first be effectively encoded as such a set.

Example

Consider words over the alphabet $\Sigma = \{a, b\}$, i.e., the set of finite sequences of letters a and b.

We say that a word w is *cube-free* if and only if w cannot be written as $w = yxxxz$, where x, y, z are words and y and z are allowed to be empty words. For instance, the words

$$abbabaabaababba \text{ and } abbabbaba$$

are cube-free, whereas the words

$$aaa, \; abbabaabaababbab \text{ and } bbabbabba$$

are not. The problem is to decide whether or not an arbitrary word w consisting of the letters a and b is cube-free.

It is fairly obvious how we can view this problem as a problem of computing values of a function $f(x)$ from ω to ω. We first view words as nonnegative integers in the so-called *dyadic* representation: the letters a and b become the digits 1 and 2, respectively, and, after this identification has been made, a word $c_0c_1 \cdots c_n$ becomes the number

$$c_0 2^n + c_1 2^{n-1} + \cdots + c_n.$$

It is easy to see that in this fashion a one-to-one correspondence is established between words over the alphabet $\{a, b\}$ and nonnegative integers. (In this correspondence, the number 0 is assigned to the empty word.) For instance, we associate 729 to the word $abbabbaba$ since

$$729 = 2^8 + 2.2^7 + 2.2^6 + 2^5 + 2.2^4 + 2.2^3 + 2^2 + 2.2 + 1.$$

Clearly, this a decidable problem. For a given n, we can find out for all cubes xxx up to a certain length whether or not they appear as a subword of w, the word corresponding to n. Algorithms more efficient than such an exhausted search can be devised for this problem.

6 Turing machines

We shall first formalize computability in terms of an imaginary computing machine, namely the *Turing machine* named after its inventor Alan Turing. This construction was made in the middle thirties of the previous century.

A Turing machine can be viewed as a black box, provided with a read-write head. The latter scans a potentially infinite tape that is marked off in squares, each square containing either a blank B or a letter from a fixed and finite alphabet. The tape is potentially infinite in both directions; this means that that the read-write head never reaches the end of it. However, at any time only finitely many squares can be non-blank. A Turing machine operates in discrete time steps. At each moment in time it is in a specific (memory) *state*, the number of all possible states being finite. Among the states is a specific *initial* state q_0 and a specific *final* state q_F.

To define a Turing machine TM, we have to specify the state set, the alphabet and finitely many instructions of the form

$$(q, a) \mapsto (q', a', m),$$

where the q s are states, the a are letters from the alphabet including a symbol for a blank square, and m assumes one of the two values L (right) or R (right).

The instruction means that if TM is in state q and scans a square containing a, then it replaces a by a', goes to the state q' and moves the read-write head to the right or left according to m. For example, the instruction

$$(q, a) \mapsto (q_1, b, R),$$

if TM is in state q, will change the word aba, with the first a being scanned by the read-write head, to the word bba, the state of the machine becomes q_1 and the read-write head will now scan the second b of the word bba. One can symbolise this transition as:

$$qaba \mapsto bq_1ba.$$

Similarly the same instruction will effect the transition

$$abqab \mapsto abbq_1b.$$

The word $abqab$ symbolises the total state affairs of the machine at a given time. It means that the contents of the tape is $abab$ (i.e four consecutive squares are filled, in sequence by a, b, a, b respectively, all the other squares being blank), that the machine is in state q and, finally that the read-write head is scanning the second a from the left. The symbol $abqab$ is called the *instantaneous description* of this state of affairs.

Example

Suppose our alphabet is the singleton $\{1\}$, a blank square is denoted by 0 and the machine has three states q_0, q_1, q_2, with q_0 being the initial state and q_2 the final state. Now define TM by the following sequence of instructions:

$$(q_0, 1) \mapsto (q_0, 1, R); \quad (q_0, 0) \mapsto (q_1, 1, L);$$

and

$$(q_1, 1) \mapsto (q_1, 1, L); \quad (q_1, 0) \mapsto (q_2, 0, R).$$

Then TM will transform $q_0 11$ to $q_2 111$. Indeed, the instantaneous descriptions will evolve as follows:

$$q_0 11 \mapsto 1 q_0 1 \mapsto 11 q_0 0 \mapsto 1 q_1 11 \mapsto q_1 111 \mapsto q_1 0111 \mapsto q_2 111.$$

The tape of a Turing machine is used both for input and output, as well as for memory.

We now define how a Turing machine computes values of functions $f(x_1, \ldots, x_k)$ considered above. We represent an integer n by a sequence of $n+1$ ones. (This *unary* notation is the simplest possible.) We denote the blank B by 0 in this context. Thus each tape contains either 0 or 1. We use $n+1$ rather than n ones to represent the number n to distinguish the blank symbol from the number 0. The symbol 1^{n+1} is used to designate $n+1$ ones. The argument $(x_1, x_2, \ldots, x_k)$ is written as

$$1^{x_1+1} 0 1^{x_2+1} 0 \cdots 0 1^{x_k+1}$$

on consecutive squares of the tape. Initially, TM scans the leftmost square containing an 1 in the state q_0.

A (partial) function is termed *Turing computable* if the following conditions are satisfied for some Turing machine TM:

1. If $f(x_1, \ldots, x_k)$ is defined, then TM eventually halts in the final state q_F scanning the left-most one in a sequence consisting of $f(x_1, \ldots, x_k) + 1$ ones, i.e., the computation will evolve from the instantaneous description $q_0 1^{x_1+1} 0 1^{x_2+1} 0 \cdots 0 1^{x_k+1}$ to the instantaneous description $q_F 1^{f(x_1, \ldots, x_k)+1}$, the remainder of
2. If $f(x_1, \ldots, x_k)$ is undefined, then TM never reaches the final state q_F.

Thus, the Turing machine in the preceding example computes the successor function $f(n) = n+1$, for all n.

Exercises

1. Consider the following Turing machine over $\{0, 1\}$ with 0 denoting a blank square:

$$\begin{aligned}
(q_0, 1) &\mapsto (q_0, 1, R) \\
(q_0, 0) &\mapsto (q_1, 1, R) \\
(q_1, 1) &\mapsto (q_1, 1, R) \\
(q_1, 0) &\mapsto (q_2, 0, L) \\
(q_2, 1) &\mapsto (q_3, 0, L) \\
(q_3, 1) &\mapsto (q_4, 0, L) \\
(q_4, 1) &\mapsto (q_4, 1, L) \\
(q_4, 0) &\mapsto (q_F, 0, R).
\end{aligned}$$

   If the input is $1^{x_1+1} 0 1^{x_2+1}$, what will be the output?

2. For a given $n \geq 0$, let TM be the Turing machine over the alphabet $\{0,1\}$ and states $q_0, \ldots, q_{n+2}$ with the instructions

$$\begin{aligned} (q_n, 0) &\mapsto (q_{n+1}, 1, L) \\ (q_{n+1}, 1) &\mapsto (q_{n+1}, 1, L) \\ (q_{n+1}, 0) &\mapsto (q_{n+2}, 0, R), \end{aligned}$$

and, if $n \geq 1$, we also have the instructions

$$(q_i, 0) \mapsto (q_{i+1}, 1, R) \quad i = 0, \ldots, n-1.$$

Assume that q_0 is the initial state and that q_{n+2} is the final state. What will the output be if we start with a blank tape? (This means that the initial instantaneous description is $q_0 0$.)

7 Church's thesis

We started our discussion with the intuitive notion of an algorithm or effective procedure, as well as the resulting, also informal, notions of an effectively computable function and of a decidable problem. We then introduced the formal notion of a Turing machine and the resulting notion of a partial Turing computable function.

Clearly, a Turing machine defines an effective procedure in the intuitive sense. This means that our formal notions are not too broad: everything decidable in the formal sense will also be decidable in the intuitive sense. But are the formal notions perhaps too narrow? Is there something effectively computable in the intuitive sense that cannot be computed by a Turing machine?

The statement that a Turing machine is an adequate formal model for the intuitive notion of an effective procedure is known as *Church's thesis*. Thus, Church's thesis asserts that the formalisation in terms of Turing machines is not too narrow. The evidence supporting Church's thesis is quite strong. The evidence includes:

1. All suggested, very diverse, alternative formulations of the class of effective procedures have turned out to be provably equivalent to the Turing machine formalization.
2. The class of mappings computable by Turing machines has strong invariance properties. Many modifications, variations and extensions of Turing machines have been proved to be equivalent to the original Turing machine model.
3. All intuitively effective procedures considered so far have found their formal counterpart in the framework of Turing machines. For example, anything that can be programmed in a programming language can be simulated by a Turing machine computation.

8 The beaver problem

We denote by $T(n)$ $n \geq 1$, the following set of Turing machines. A Turing machine TM with states $q_0, \ldots, q_n$ belongs to $T(n)$ if and only if TM halts in the final state q_n when started on the blank tape. Here q_0, respectively q_n, is the initial state, respectively, final state. For each TM in $T(n)$, we denote by $\mathbf{ones}(TM)$ the number of ones on the tape when TM halts. Define

$$\mathbf{beaver}(n) = \max\{\mathbf{ones}(TM) : TM \in T(n)\}.$$

Thus, $\mathbf{beaver}(n)$ equals the maximal number of ones that an $(n+1)$-state Turing machine as described above can print when started on a blank tape. Clearly, $T(n)$ is finite and $\mathbf{ones}(TM)$ is a specific number for each TM in $T(n)$. Thus, to obtain $\mathbf{beaver}(n)$, we only have to take the greatest among finitely many numbers. Therefore, it is clear that **beaver** is a total function. It is a remarkable fact that **beaver** cannot be computed by any Turing machine. Indeed, it can be shown that if f is any total Turing-computable function, then

$$f(n) < \mathbf{beaver}(n)$$

for all sufficiently large n. (A proof of this statement can be found on p7 of [RS].) Thus **beaver** grows eventually faster than any Turing-computable function. In particular, by Church's thesis, the function **beaver** is not computable. This provides a counterexample to the claim sometimes made: whenever a function has been defined precisely in finite terms, then it is effectively computable.

Exercise

1. Deduce from the fact that **beaver** is not algorithmically solvable that there is no algorithmic procedure for deciding whether or not a Turing machine with a given input would halt.

2. Argue informally that the set of computable functions on ω is a countable set. Find thus another proof of the fact that there is a function $f : \omega \to \omega$ which is not computable.

9 Time complexity: an introduction

The *time complexity* of an algorithm is a function of the length of the input. An algorithm is of time complexity $f(n)$ if, for all n and all inputs of length n, the execution of the algorithm takes at most $f(n)$ steps. If the inputs to the algorithm are natural numbers m, the (natural) length of an input m is the number of digits or bits in m. Consequently, its length is approximately $\log_2 m$. When processing natural numbers, it is more efficient to present them in binary form or, equivalently, as finite words over the alphabet $\{0, 1\}$.

Clearly, the time complexity depends on the machine model we have in mind. However, it can be shown that such fundamental notions such as polynomial time complexity are independent of the model. This concerns only models chosen with good taste. For example, one should not include in one step an abstract subroutine for a complex problem, such as the testing of the primality of a given nonnegative integer.

There are often slow and fast algorithms for the same problem. In general, it is a major mathematical challenge to establish good lower bounds for the time complexity of an algorithmic problem, i.e., to show that every algorithm for a specific problem is at least of a certain time complexity. The considerations below will mostly deal with upper bounds for the time complexity of algorithmic problems.

Natural complexity measures are obtained by considering Turing machines. How many steps (in terms of the length of the input) does a computation require? This is a natural formalization of the notion of time resource. Similarly the number of squares visited during a computation constitutes a natural space measure.

Consider a Turing machin TM that halts for all inputs. The *time complexity function* associated with TM is defined by

$$f_{TM}(n) = \max\{m : TM \text{ halts after } m \text{ steps for an input of length } n\}.$$

The Turing machine is said to be *polynomially bounded* if there is a polynomial $p(n)$ such that

$$f_{TM}(n) \leq p(n)$$

holds for all n. The notation $\mathbf{P}$ is used for all problems that can be solved by a polynomially bounded Turing machine.

A *nondeterministic Turing machine* NTM may have several possibilities for its behaviour, i.e., several possible instructions for a given pair (q, a). It means that an input gives rise to several computations. This can be visualised as the machine making guesses or using an arbitrary (but finite) number of parallel processors.

As an illustration, a way for testing, in a nondeterminstic way, whether or not a given nonnegative number n is a composite number might be informally described as follows: for given n test, in parallel, for all $1 < k < n$ whether k is a divisor of n. Whenever the answer is *true* , enter the halting state, else enter an infinite loop. This nondeterministic algorithm will enter some halting state if and only if n is indeed composite. Thus, if n is a prime number, this procedure will yield no output. It should be clear (at least in principle) how to implement this nondeterministic algorithm on a nondeterministic Turing machine.

What is a problem? Informally, it is a question or a task, for instance: "Does a given network have a circuit that goes once through every node?" The problems of interest to us are those that can be answered by either "yes" or "no". We shall encode objects by words over finite alphabets so that most problems can be represented as: Given a word w, does it have a certain property? Thus the problem is fully specified by describing the "certain property". This, in turn, is fully described by just the set of words that have the property. Therefore, we have the following mathematical definition: a *problem* is a subset Π of Σ^*, where Σ^* denotes the set of words over the alphabet (finite set) Σ.

If we consider any problem Π the corresponding "informal" problem is: Given word w, does w belong to Π?

The class **NP** is the class of problems for which a positive answer has a "certificate" from which the correctness of the *positive* answer can be derived in polynomial time. We shall now make this more precise.

The class **NP** consists of those problems $\Pi \subset \Sigma^*$ for which there exists a problem Π' in **P** and a polynomial $p(x)$ such that, for any word $w \in \Sigma^*$, it is the case that $w \in \Pi$ if and only if there exists a word v, the certificate, such that $(w, v) \in \Pi'$ and such that $|v| \leq p(|w|)$.

For example let $COMP$ be the set of composite natural numbers represented in binary of course. Then a certificate v for a composite number n is a proper factor v of n. The verification that v is a proper factor of n only takes polynomial time. (Use the Euclidian algorithm, for example.) It follows that the problem $COMP$ belongs to **NP**.

Exercise

1. A knapsack vector $A = (a_1, \ldots, a_n)$ is an ordered n-tuple, of distinct positive integers a_i. The knapsack problem is: Given a natural number α, is there a subset of A, the elements of which sum up to α? (Since we are talking about a subset, each a_i may appear in the sum at most once.) Explain carefully why this problem belongs to **NP**.

10 NP-completeness

In a sense to be made explicit below, the **NP**-complete problems are the hardest problems in **NP**. To explain this concept, we first need the notion of a polynomial-time reduction: Let Π and Π' be two problems, and let A be an algorithm. We say that A is a polynomial time reduction of Π' to Π if A is a polynomial-time algorithm, so that, if A yields, upon input w, the output v, then $w \in \Pi'$ if and only if $v \in \Pi$. Moreover, we require that the size $|v|$ of the output v is polynomially bounded by the size $|w|$ of the input w. In other words, a polynomial-time reduction algorithmically transforms solutions of the problem Π' to solutions of the problem Π and, moreover, only positive solutions of Π' are thus transformed; in addition, the transformation takes place in polynomial time and the size of the output depends polynomially on the size of the input.

A problem is said to be **NP**-*complete*, if it is in **NP** and, for each problem Π' in **NP**, there exists a polynomial-time reduction of Π' to Π (and such a polynomial reduction can be effectively found from the description of Π').

It is not difficult to see that if Π belongs to **P** and there is a polynomial-time reduction of Π' to Π, then Π' also belongs to **P**. It has the implication that if any one **NP**-complete problem can be solved in polynomial time, then all the problems in **NP** can be solved in polynomial time. It is a major open problem whether any, hence all, **NP** problems can be solved in polynomial time. In cryptography, an **NP**-complete problem is frequently considered to be intractable and, indeed, the security of many crypto-systems is based on the hypothetical intractability of **NP**-complete problems. An important example for cryptography of an **NP**-complete problem is the knapsack problem. It is, however, not known whether $COMPOSITE$ is **NP**-complete if Π belongs to **P** and there is a polynomial-time reduction of Π' to Π, then Π' also belongs to **P**. It has the implication that if any one **NP**-complete problem can be solved in polynomial time, then all the problems in **NP** can be solved in polynomial time. It is a major open problem whether any, hence all, **NP** problems can be solved in polynomial time. In cryptography, an **NP**-complete problem is frequently considered to be intractable and, indeed, the security of many crypto-systems is based on the hypothetical intractability of **NP**-complete problems. An important example for cryptography of an **NP**-complete problem is the knapsack problem. It is, however, not known whether $COMPOSITE$ is **NP**-complete.

For an extensive discussion and many examples of **NP**-complete problems the reader may consult the beautiful (and now classical) book [GJ].

Exercises

1. Show that, if Π belongs to **P** and there is a polynomial-time reduction of Π' to Π, then Π' also belongs to **P**.

2. Show that if Π belongs to **NP**, Π' is **NP**-complete, and there exists a polynomial-time reduction of Π' to Π, then Π is also **NP**-complete.

3. It is known that the knapsack problem is **NP**-complete. Moreover, it has already been noted that $COMPOSITE$ belongs to **NP**. It follows that there is a polynomial-time reduction from $COMPOSITE$ to the knapsack problem. Make this explicit. (Just apply the preceding definitions to this particular instance.)

11 Randomised algorithms

```
Algorithm : random Quicksort(S)
    Choose $y$ from $S$ uniformly and at random.
    $S_1 := \{x \in S : x < y\}$.
    $S_2 := \{x \in S : x > y\}$.
    If $S_1 \neq \emptyset$ then random Quicksort($S_1$).
    Output $y$.
    If $S_2 \neq \emptyset$ then random Quicksort($S_2$).
```

If $|S| = n$ then expected running time is $O(n \log n)$. Las Vegas algorithm. If process halts then correct answer.

A typical Monte-Carlo typical randomised algorithm:

Verify $AB = C$ for given $n \times n$ matrices A, B, C. The naïve method for checking this requires $O(n^3)$ arithmetic operations.

We next represent, for every $k \geq 1$, a randomised algorititthm that will decide the relation $AB = C$ in $O(kn^2)$ steps, with a probability of error which is bounded by $\frac{1}{2^k}$.

```
Algorithm: CheckingMatrixMultiplication
    Choose $r = (r_1, \ldots, r_n) \in \{0,1\}^n$ at random.
        If $A(Br) = Cr$ then TRUE else FALSE.
```

Execution time : $O(n^2)$. This is the number of multiplications and additions required to check whether or not $A(Br) = Cr$ for a given r.

If the output of the program is **false**, then indeed $AB \neq C$.

However it might happen that the output is **true** even though $AB \neq C$.

The probability of this hapenning is though $\leq \frac{1}{2}$. ****(Proof below).

In other words, if $AB \neq C$, the probability that a random choice of $r = (r_1, \ldots, r_n) \in \{0,1\}^n$ will yield the answer **true** is at most $\frac{1}{2}$.

All in all, the probability that this randomised algorithm gives the wrong answer is at most $\frac{1}{2}$ but if the output is **false**, then $AB \neq C$ with certainty.

If we repeat this algorithm k times each time randomly and independently selecting r the probability of getting the wrong answer is $(\frac{1}{2})^k$ which rapidly becomes negligibly small.

The number of computations required is then $O(kn^2)$.

So if we get during the k iterations the answer **false** at least once, the output is correct.

If we get the answer **true** during all k iterations, we can conclude that indeed that $AB = C$ with a probability of error at most $(\frac{1}{2})^k$.

**** Let $D = (d_{ij})$ be given by $AB - C$. Assume that $D \neq 0$. By interchanging rows and columns we may assume that $d_{11} \neq 0$. If $Dr = 0$, then

$$r_1 = -\frac{\sum_{j=2}^{n} r_j d_{1j}}{d_{11}}.$$

For fixed $r_2, \ldots r_n$ the latter event can hold for at most one value of r_1 (0 or 1) and, therefore with probability at most $\frac{1}{2}$. Hence

$$\text{Prob}(Dr = 0) \leq \frac{1}{2},$$

as required.****

12 Meta-heuristic algorithms, No Free Lunch phenomena

In combinatorial optimisation, one is given a finite set V and a function $f : V \to \mathbf{R}$. The task is to find an element $x \in V$ that maximises (or minimises) $f(x)$. At first sight, this may seem like a trivial task: since V is finite, all that is required is simply to go through all the $x \in V$ in some systematic fashion, to calculate $f(x)$ for each x, while keeping track of the maximum recorded up to the stage of the computation.

The reason for this brute force approach not to suffice, is that the set V is usually so large that the time (number of computational steps) required makes it totally infeasible, if not utterly impossible. Typically, the number of elements of V grows at least exponentially in some parameter n that describes the size of the problem in a natural way. For instance V could be the set of binary strings of length n, rendering V a set of cardinality 2^n. In this case the brute force method of calculating $f(x)$ for $x \in V$ is out of the question for all $x \in V$ even for moderately sized problems such as for $n = 100$, for instance. (Just ponder on the fact that 2^{100} nano-seconds are $4.01702815 \times 10^{10}$ millenia!)

Other less time-consuming methods are therefore called for. A common approach involves so-called *local search* in V. This requires the introduction of some (sometimes artificial) geometric/geographic structures upon V, which can quite frequently be accomplished by declaring (imposing) some links between some of the pairs of elements x, y of V. Once such a structure has been introduced, the set of all points y that are linked to a given x is called the *neighbourhood* of x. There is much freedom in defining the links but it needs to be done in such a way that each x has a neighbourhood of manageable size, and that the network of links become well-connected in the sense that there are paths consisting of successive links that connect many pairs of points and the paths are on average of manageable sizes. Fortunately, in many specific examples, natural link structures often more or less suggest themselves. This will become clear as we study heuristic algorithms in this module.

Given the link structure, the basic local search algorithm then proceeds as follows: Start at some arbitrary $x \in V$, compute f at x and at all its neighbours, and move to the neighbour y the f-value of which is the largest (unless they they are all smaller than $f(x)$ in which case we stay at x - which is a deadlock-situation). If possible, repeat the process until we reach a deadlock.

This type of algorithm is called a *hill-climber* as it can be pictured as a hiker in a mountainous landscape, always going in the direction of the steepest climb until a (local) top of a hillock is reached. Such hill-climbing sometimes works well, but a huge drawback is that the algorithm may reach a deadlock on a relatively modest hill without noticing the huge mountain peaks further away.

To deal with this drawback, a variety of modifications of the hill-climber algorithm have been proposed and are widely used. This modifications may include a simulated randomisation of some of the local choices in order to allow for occasional downhill steps (as in the famous simulated annealing algorithm), it may permit occasional long jumps in the landscape or it could use measures of simulated analogues of biological fitness-landscapes in order to make decisions as to which the next computational step would be. As we will see, many of these modifications are quite sophisticated and find inspiration from areas ranging from biology to statistical physics. These algorithms fall under the class of what is called blackbox or meta-heuristic algorithms.

These algorithms are not only used for pure optimisation problems but also for the purpose of locating some large (but not necessarily the largest) value of f. The goal might be to find some

$x \in V$ such that $f(x)$ exceeds some pre-given t. In other words we want to find some x such that $f(x)$ hits the target set

$$T := \{x \in V : \ f(x) \geq t\}.$$

More generally, one might want to find some x such that $f(x)$ is in some other subset T of V which will then be the target set for another problem. In many interesting search problems the target set T might be such that only a small fraction of the elements of V is in T. Sometimes the target set has some internal (mathematical or statistical) structure which can be exploited with quite favourable results.

We now turn to the so-called No Free Lunch (NFL) theorems of Wolpert and Macready (1997), who showed for these optimisation and search problems, no metaheuristic is better than any other, in a certain *average sense*. This might sound very surprising, so let we discuss the statement of the first NFL theorem.

Wolpert and Macready restrict themselves to the setting where the range of the functions f belongs to some fixed, prescribed and finite subset S of $\mathbf{R}$. This is a natural restriction because in any computer implementation everything is necessarily discrete.

The algorithms considered by Wolpert and Macready are of the following form: First, an element x_1 of V is chosen according to some rule (which may or may not involve random choices), whereupon $f(x_1)$ is computed. Then x_2 is chosen according to some rule that might take of both x_1 and $f(x_1)$ after which $f(x_2)$ is computed. After k steps, there is a record of $x_1, \ldots, x_k$ as well as of $f(x_1), \ldots, f(x_k)$, which could we used to compute first x_{k+1} in V and then $f(x_{k+1})$. The only other condition that the NFL theorem requires is that no element of V may be chosen more than once.

For given k, let E_k be any event defined in terms of the values of f at the first k values of x_i. The protype example is to let E_k be the event that at least one of the recorded values of f at the first k chosen points belong to a pre-given target set T. The first NFL theorem says essentially, that for given S, V, T, k and event E_k as above

> the average probability over all the functions f from V to S of the event E_k is independent of the choice of the meta-heuristic algorithm.

The theorem indicates that no meta-heuristic, to be applied to all possible functions from V to S will have an expected performance any better than a stupid blind random search! The moral of this theorem might be that we cannot expect to construct efficient optimisation or search meta-heuristc algorithms unless we exploit some specific properties of the function f - hence the name of this theorem. This also explains why meta-heuristic algorithms is such a vast area both of research and application.

The remainder of the discussion is mathematically more sophisticated and can be skipped by the reader. But perhaps some might be sufficiently intrigued to at least understand the precise statement of the NFL theorem.

There are many ways in which one can express the above admittedly vague statement in a rigorous fashion. I shall express the statement in a probabilistic framework. There are two advantages to this approach: Firstly, one can translate the statement into a purely combinatorial statement and secondly, for some applications to evolutionary algorithms the statement has meaning for understanding evolutionary processes in a noisy landscape (environment).

For finite sets V and S with S being a subset of the reals we write S^V for the set of functions mapping V into S. This notation was inspired by the following fact. If V has ν elements and S has σ elements then S^V has σ^ν elements. If we write $|X|$ for the number of elements (the cardinality) of a finite set, this means that

$$|S^V| = |S|^{|V|}.$$

The uniform probability distribution on a finite set X attaches the probability $|X|^{-1}$ to each element of X and such that the probability attached to k distinct elements of X is given by $|X|^{-k}$. In other words, we attach to each element of X the same probability in such a way that the attachment to distinct elements are statistically independent from one another.

Theorem 1 *Let V and S be finite sets with $S \subset \mathbf{R}$. Let A be any meta-heuristic applied to all the functions from V to S such that it generates, for each such $f \in S^V$, a sequence $x_1, x_2, \ldots$ (depending on f of course) in V for solving a search or optimisation problem for f. If we now impose the uniform distribution on S^V then, for each $k \leq |V|$, and any subset W of S^k, the probability of the event stating that the values $(f(x_1), \ldots, f(x_k))$ of f at the (seed) points $x_1, \ldots x_k$ generated by A belong to W, is independent from the choice of the meta-heuristic A. The probability depends on the sizes of S, V, the number k and the set W only.*

Mathematically

$$\text{IF } (A \text{ generates } (x_1, x_2, \ldots x_k)) \text{ THEN Prob}((f(x_1), \ldots, f(x_k)) \in W) = \phi(|S|, |V|, k, W),$$

for some function ϕ which is independent from the meta-heuristic A.

Proof. The essential idea is that with respect to the uniform distribution on S^V, for ANY *distinct* $v_1, \ldots, v_k \in V$, the distribution of the random variables $f(v_1), \ldots, f(v_k)$ on S^V with the uniform distribution is statistically similar to k independent uniformly distributed random variables assuming values in S. Consequently the distribution of $f(x_1), \ldots, f(x_k)$ will be the same, no matter how they were generated by A, as long as the seed-values x_i generated by A are distinct.

Each element v of V can be viewed as an S-valued random variable, to wit, the function

$$f \mapsto f(v), \; f \in S^V.$$

These random variables are independent and uniformly distributed on S. To prove this it suffices to show, for $s_1, \ldots, s_k$ in S and distinct $v_1, \ldots, v_k$ in V that

$$\text{Prob}(f(v_1) = s_1 \wedge \ldots \wedge f(v_k) = s_k) = \frac{1}{\sigma^k}.$$

The set of functions f such that $f(v_1) = s_1 \wedge \ldots \wedge f(v_k) = s_k$ is in a natural bijection with the set of functions $f \in S^{V - \{v_1, \ldots, v_k\}}$. There are $\sigma^{\nu - k}$ such functions. Consequently,

$$\text{Prob}(f(v_1) = s_1 \wedge \ldots \wedge f(v_k) = s_k) = \frac{\sigma^{\nu-k}}{\sigma^\nu} = \frac{1}{\sigma^k},$$

as required.

Example: $1 \leq t < \sigma := |S|,\ S = \{1, \dots \sigma\},\ W := \{(s_1, \dots, s_k) \in S^k : s_1, \dots, s_{k-1} \leq t, s_k > t\}$, then the associated probability that the algorithm A will reach a point the value of which exceeds t for the first time after k steps is given by:

$$(\frac{t}{\sigma})^{k-1}(1 - \frac{t}{\sigma}),$$

ostensibly independent from A, even non-zero for $k = |V|$, case of exhaustive search!! This expression then, is the probability that the halting time on a function f under an arbitrary meta-heuristic will be k.

Exercise: Compute the probability that the halting time is at most k.

Even though the seeds may be stochastically generated, not necessarily uniformly, this stochastic process will have no impact on the theorem, or, more specifically, on the halting-time distribution.

```
 State the results in non-probabilistic language.
```

e.g

$$|\{f \in S^V : (f(x_1), \dots, f(x_k)) \in W\}| = \psi(|S|, |T|, k, W),$$

ψ being A-independent.

Exercise: Calculate this when $W = \{(s_1, \dots, s_k) \in S^k : s_1, \dots, s_{k-1} \leq t, s_k > t\}$.

Answer: $\sigma^\nu(\frac{t}{\sigma})^{k-1}(1 - \frac{t}{\sigma})$ when $\nu = |V|$. (Why?)

Suppose $V = \{v_1, \dots, v_\nu\}$. Suppose $\mathbf{P}$ is any probability measure on S^V such that, for any any permutation permutation π of $1, \dots, \nu$, the probability distributions (induced by $\mathbf{P}$) of the sequence of random variables $(f(v_1), \dots, f(v_\nu))$ and of the sequence $(f(v_{\pi(1)}), \dots, f(v_{\pi(\nu)})$ are the same. Then we call $\mathbf{P}$ a *symmetric* probability distribution on S^V.

Theorem 2 *Let V and S be finite sets with $S \subset \mathbf{R}$. Let A be any meta-heuristic applied to all the functions from V to S such that it generates, for each such $f \in S^V$, a sequence $x_1, x_2, \dots$ (depending on f of course) in V for solving a search or optimisation problem for f. If we now impose any symmetric probability distribution $\mathbf{P}$ on S^V then, for each $k \leq |V|$, and any subset W of S^k, the probability p of the event stating that the values $(f(x_1), \dots, f(x_k))$ of f at the (seed) points $x_1, \dots x_k$ generated by A belong to W, is independent from the choice of the meta-heuristic A. The probability depends on the sizes of S, V, the number k and the set W only.*

Proof: Take any permutation of $1, \dots, \nu$ that maps v_i to x_i for all $i = 1, \dots, k$. It follows from the symmetry of $\mathbf{P}$ that

$$p = \mathbf{P}((f(v_1), \dots, f(v_k)) \in W),$$

which is ostensibly independent from the meta-heuristic A.

For $f \in S^V$ and a permutation π of the set V, define the function πf by

$$v \mapsto f(\pi^{-1}v),\ v \in V.$$

This defines an group action of the symmetry group $\mathcal{S}_\nu$ on V onto S^V. A subset Y of S^V is said to be *permutation-closed* if it Y is invariant under this group action. This means that if $f \in Y$, then $\pi f \in Y$, for all $f \in Y$. This has the implication that Y must the disjoint union of orbits in S^V under the action of $\mathcal{S}_\nu$ on S^V. We now formulate and proof a no free lunch theorem for meta-heuristic algorithms restricted to a permutation-closed subset Y of S^V This will make clear that all the NFL-theorems are manifestations of symmetry and the Cartesian closedness of the category of finite sets. (Private note, to be elaborated upon later.

Theorem 3 *Let V and S be finite sets with $S \subset \mathbf{R}$. Let A be any meta-heuristic applied to a permutation-closed subset Y of S^V such that it generates, for each such $f \in S^V$, a sequences $x_1, x_2, \ldots$ (depending on f of course) in V for solving a search or optimisation problem for $f \in Y$. If we now impose the uniform distribution on Y then, for each $k \leq |V|$, and any subset W of S^k, the probability of the event stating that the values $(f(x_1), \ldots, f(x_k))$ of $f \in Y$ at the (seed) points $x_1, \ldots x_k$ generated by A belong to W, is independent from the choice of the meta-heuristic A.*

Proof. For two sequences of distinct elements $v_1, \ldots, v_k$ and $x_1, \ldots, x_k$ in V and for any any $s_1, \ldots, s_k$ in S we shall construct a bijection between the sets A and B, where:

$$A = \{f \in Y : \ f(x_i) = s_i, \ i = 1, \ldots, k\},$$

and

$$B = \{f \in Y : \ f(u_i) = s_i, \ i = 1, \ldots, k\}.$$

List V as $v_1, \ldots, v_k, v_{k+1}, \ldots v_\nu$ and also as $x_1, \ldots, x_k, x_{k+1}, \ldots x_\nu$ and let π be the permutation that maps x_i to u_i for all i. Then the mapping $f \mapsto \pi^{-1} f$ defines a bijection between A and B. The remainder of the proof proceeds as before.

Analogue for a symmetric probability on a permutation-closed set Y.
Ways of computing sizes of such Y. For instance if Y is the orbit of f under the action of the symmetry group then

$$|Y| = \frac{m!}{n_1! n_2! \ldots n_k!}.$$

13 ASSIGNMENT

1. Show that if $p_1, \ldots, p_n$ are the first n prime numbers then

$$p_1 \cdots p_n + 1$$

is divisible by a prime numbers which is different from $p_1, \ldots, p_n$. Deduce that there are infinitely many prime numbers.

2. Let Prime(x) be the Boolean-valued predicate which outputs *true* when x is a prime number, and *false* otherwise. Give an informal description of an effective procedure for computing Prime(x) for a given input x. Do you know an efficient procedure?

3. The function $f(x)$ yielding the xth prime number can be described as follows: $f(1) = 2$ and

$$f(x+1) = \min[y > f(x) : \text{Prime}(y)].$$

This is a recursive (self-referential) description because the computation of $f(x+1)$ calls on the value of the *same* function f at the the value x. Design a programme for computing $f(x)$ which is not self-referential in this sense.

4. (Easy) What is the domain of the function computed by the following algorithm?

```
input $i \geq 0$
begin
        repeat
            $i := i + 1$
        until $i = 0$
end
```

5. Show that every subset of a countably infinite set is either finite or countably infinite.

6. Show that the set consisting of all finite subsets of the natural numbers is a countably infinite set.

7. Show that the set of all (total) functions on the nonnegative integers ω assuming values in ω is not countably infinite.

8. Give an informal argument for establishing that all total computable functions is countably infinite. Conclude that not all functions (in the mathematical sense) on ω taking values in ω are computable functions. (We shall later come across "concrete" examples of non-computable functions.)

9. Consider the following Turing machine over $\{0, 1\}$ with 0 denoting a blank square:

$$\begin{aligned}
(q_0, 1) &\mapsto (q_0, 1, R)\\
(q_0, 0) &\mapsto (q_1, 1, R)\\
(q_1, 1) &\mapsto (q_1, 1, R)
\end{aligned}$$

$$\begin{aligned}(q_1,0) &\mapsto (q_2,0,L)\\(q_2,1) &\mapsto (q_3,0,L)\\(q_3,1) &\mapsto (q_4,0,L)\\(q_4,1) &\mapsto (q_4,1,L)\\(q_4,0) &\mapsto (q_F,0,R).\end{aligned}$$

If the input is $1^{x_1+1}01^{x_2+1}$, what will be the output?

10. For a given $n \geq 0$, let TM be the Turing machine over the alphabet $\{0,1\}$ and states $q_0,\ldots,q_{n+2}$ with the instructions

$$\begin{aligned}(q_n,0) &\mapsto (q_{n+1},1,L)\\(q_{n+1},1) &\mapsto (q_{n+1},1,L)\\(q_{n+1},0) &\mapsto (q_{n+2},0,R),\end{aligned}$$

and, if $n \geq 1$, we also have the instructions

$$(q_i,0) \mapsto (q_{i+1},1,R) \;\; i=0,\ldots,n-1.$$

Assume that q_0 is the initial state and that q_{n+2} is the final state. What will the output be if we start with a blank tape? (This means that the initial instantaneous description is $q_0 0$.)

11. Deduce from the fact that **beaver** is not algorithmically solvable that there is no algorithmic procedure for deciding whether or not a Turing machine with a given input would halt.

12. A knapsack vector $A = (a_1,\ldots,a_n)$ is an ordered n-tuple, of distincts positive integers a_i. The knapsack problem is: Given a natural number α, is there a subset of A, the elements of which sum up to α? (Since we are talking about a subset, each a_i may appear in the sum at most once.) Explain carefully why this problem belongs to **NP**.

13. Show that, if Π belongs to **P** and there is a polynomial-time reduction of Π' to Π, then Π' also belongs to **P**.

14. Show that if Π belongs to **NP**, Π' is **NP**-complete, and there exists a polynomial-time reduction of Π' to Π, then Π is also **NP**-complete.

15. It is known that the knapsack problem is **NP**-complete. Moreover, it has already been noted that $COMPOSITE$ belongs to **NP**. It follows that there is a polynomial-time reduction from $COMPOSITE$ to the knapsack problem. State fully what this means.

16. Argue informally that the set of computable functions on ω is a countable set. Find thus another proof of the fact that there is a function $f : \omega \to \omega$ which is not computable.

References

[] Some excellent sources on an introductory level for the theory of computability are:

[C] Cutland N.J., Computability. An introduction to recursive function theory. Cambridge University Press, New York 1992.

[BJ] Boolos G.S. and Jeffrey R.C., Computability and Logic. Cambridge University Press, New York 1989.

[RS] Rozenberg G. and Salomaa A., Cornerstones of Undecidability. Prentice Hall, New York 1994.

[] A succinct discussion of the theory of **NP**-completeness can be found in the final chapter of:

[CCPS] Cook W.J., Cunningham W.H., Pulleybank W.R. and Schrijver A., Combinatorial Optimisation. John Wiley and Sons, New York 1998.

[] The classical source on computational complexity is:

[GJ] Garey M.R. and Johnson D.S., Computers and Intractability: A guide to the theory of NP-completeness. Freeman, San Francisco 1979.