Introduction to theory of computation

Tom Carter

http://cogs.csustan.edu/~ tom/SFI-CSSS

Complex Systems Summer School

June, 2002

Our general topics: ←

- ⊚ Symbols, strings and languages
- ⊚ Finite automata
- ⊚ Regular expressions and languages
- ⊚ Markov models
- ⊚ Context free grammars and languages
- ⊚ Language generators and recognizers
- ⊚ The Chomsky hierarchy
- ⊚ Turing machines
- ⊚ Computability and tractability
- ⊚ Computational complexity
- ⊚ References

Introduction

←

What follows is an extremely abbreviated look at some of the important ideas of the general areas of automata theory, computability, and formal languages. In various respects, this can be thought of as the elementary foundations of much of computer science. It also includes a wide variety of tools, and general categories of tools . . .

Symbols, strings and languages ←

- The classical theory of computation traditionally deals with processing an input string of symbols into an output string of symbols. Note that in the special case where the set of possible output strings is just {'yes', 'no'}, (often abbreviated {T, F} or {1, 0}), then we can think of the string processing as string (pattern) recognition.

  We should start with a few definitions. The first step is to avoid defining the term 'symbol' – this leaves an open slot to connect the abstract theory to the world . . .

  We define:

  1. An *alphabet* is a finite set of symbols.

2. A *string* over an alphabet A is a finite ordered sequence of symbols from A. Note that repetitions are allowed. The length of a string is the number of symbols in the string, with repetitions counted. (e.g., $|\mathsf{aabbcc}| = 6$)

3. The empty string, denoted by ϵ, is the (unique) string of length zero. Note that the empty string ϵ is not the same as the empty set $\emptyset$.

4. If S and T are sets of strings, then $\mathsf{ST} = \{xy|\ x \in \mathsf{S} \text{ and } y \in \mathsf{T}\}$

5. Given an alphabet A, we define

$$\begin{aligned} \mathsf{A}^0 &= \{\epsilon\} \\ \mathsf{A}^{n+1} &= \mathsf{A}\mathsf{A}^n \\ \mathsf{A}^* &= \bigcup_{n=0}^{\infty} \mathsf{A}^n \end{aligned}$$

6. A *language* L over an alphabet A is a subset of A^*. That is, $\mathsf{L} \subset \mathsf{A}^*$.

- We can define the natural numbers, $\mathbb{N}$, as follows:

  We let

$$\begin{array}{rcl} 0 & = & \emptyset \\ 1 & = & \{\emptyset\} \\ 2 & = & \{\emptyset, \{\emptyset\}\} \\ \text{and} & \text{in} & \text{general} \\ n+1 & = & \{0, 1, 2, \ldots, n\}. \\ \text{Then} & & \\ \mathbb{N} & = & \{0, 1, 2, \ldots\}. \end{array}$$

- Sizes of sets and countability:

  1. Given two sets S and T, we say that they are the same size ($|S| = |T|$) if there is a one-to-one onto function $f: S \rightarrow T$.

  2. We write $|S| \leq |T|$ if there is a one-to-one (not necessarily onto) function $f: S \rightarrow T$.

3. We write $|\mathsf{S}| < |\mathsf{T}|$ if there is a one-to-one function $f : \mathsf{S} \rightarrow \mathsf{T}$, but there does not exist any such onto function.

4. We call a set S
   (a) Finite if $|\mathsf{S}| < |\mathbb{N}|$
   (b) Countable if $|\mathsf{S}| \leq |\mathbb{N}|$
   (c) Countably infinite if $|\mathsf{S}| = |\mathbb{N}|$
   (d) Uncountable if $|\mathbb{N}| < |\mathsf{S}|$.

5. Some examples:
   (a) The set of integers $\mathbb{Z} = \{0, 1, -1, 2, -2, \ldots\}$ is countable.
   (b) The set of rational numbers $\mathbb{Q} = \{p/q \mid p, q \in \mathbb{Z}, q \neq 0\}$ is countable.

(c) If S is countable, then so is SxS, the cartesian product of S with itself, and in general so is S^n for any $n < \infty$.

(d) For any nonempty alphabet A, A^* is countably infinite.

Exercise: Verify each of these statements.

6. Recall that the *power set* of a set S is the set of all subsets of S:

$$P(\mathsf{S}) = \{\mathsf{T} \mid \mathsf{T} \subset \mathsf{S}\}.$$

We then have the fact that for any set S,

$$|\mathsf{S}| < |P(\mathsf{S})|.$$

Pf: First, it is easy to see that

$$|\mathsf{S}| \leq |P(\mathsf{S})|$$

since there is the one-to-one function $f : \mathsf{S} \to P(\mathsf{S})$ given by $f(s) = \{s\}$ for $s \in \mathsf{S}$.

On the other hand, no function $f : \mathsf{S} \to P(\mathsf{S})$ can be onto. To show this, we need to exhibit an element of $P(\mathsf{S})$ that is not in the image of f. For any given f, such an element (which must be a subset of S) is

$$\mathsf{R}_f = \{x \in \mathsf{S} \mid x \notin f(x)\}.$$

Now suppose, for contradiction, that there is some $s \in \mathsf{S}$ with $f(s) = \mathsf{R}_f$. There are then two possibilities: either $s \in f(s) = \mathsf{R}_f$ or $s \notin f(s) = \mathsf{R}_f$. Each of these lead to a contradiction:

If $s \in f(s) = \mathsf{R}_f$, then by the definition of R_f, $s \notin f(s)$. This is a contradiction.

If $s \notin f(s) = \mathsf{R}_f$, then by the definition of R_f, $s \in \mathsf{R}_f = f(s)$. Again, a contradiction.

Since each case leads to a contradiction, no such s can exist, and hence f is not onto. QED

- From this, we can conclude for any countably infinite set S, P(S) is uncountable. Thus, for example, $P(\mathbb{N})$ is uncountable. It is not hard to see that the set of real numbers, $\mathbb{R}$, is the same size as $P(\mathbb{N})$, and is therefore uncountable.

  Exercise: Show this. (Hint: show that $\mathbb{R}$ is the same size as $(0, 1) = \{x \in \mathbb{R} \mid 0 < x < 1\}$, and then use the binary representation of real numbers to show that $|P(\mathbb{N})| = |(0, 1)|$.

- We can also derive a fundamental (non)computability fact:

  There are languages that cannot be recognized by any computation. In other words, there are languages for which there cannot exist any computer algorithm to determine whether an arbitrary string is in the language or not.

To see this, we will take as given that any computer algorithm can be expressed as a computer program, and hence, in particular, can be expressed as a finite string of ascii characters. Therefore, since ASCII* is countably infinite, there are at most countably many computer algorithms/programs. On the other hand, since a language is any arbitrary subset of A* for some alphabet A, there are uncountably many languages, since there are uncountably many subsets.

Finite automata ←

- This will be a quick tour through some of the basics of the abstract theory of computation. We will start with a relatively straightforward class of machines and languages – deterministic finite automata and regular languages.

  In this context when we talk about a machine, we mean an abstract rather than a physical machine, and in general will think in terms of a computer algorithm that could be implemented in a physical machine. Our descriptions of machines will be abstract, but are intended to be sufficiently precise that an implementation could be developed.

- A deterministic finite automaton (DFA) $\mathsf{M} = (\mathsf{S}, \mathsf{A}, s_0, \delta, \mathsf{F})$ consists of the following:

  S, a finite set of states,

  A, an alphabet,

  $s_0 \in \mathsf{S}$, the start state,

  $\delta : \mathsf{SxA} \rightarrow \mathsf{S}$, the transition function, and

  $\mathsf{F} \subset \mathsf{S}$, the set of final (or accepting) states of the machine.

  We think in terms of feeding strings from A^* into the machine. To do this, we extend the transition function to a function

  $$\widehat{\delta} : \mathsf{SxA}^* \rightarrow \mathsf{S}$$

  by

  $$\begin{aligned} \widehat{\delta}(s, \epsilon) &= s, \\ \widehat{\delta}(s, xa) &= \widehat{\delta}(\delta(s, a), x). \end{aligned}$$

We can then define the language of the machine by

$$L(\mathsf{M}) = \{x \in \mathsf{A}^* \mid \widehat{\delta}(s_0, x) \in \mathsf{F}\}.$$

In other words, $L(\mathsf{M})$ is the set of all strings in A^* that move the machine via its transition function from the start state s_0 into one of the final (accepting) states.

We can think of the machine M as a recognizer for $L(\mathsf{M})$, or as a string processing function

$$f_{\mathsf{M}} : \mathsf{A}^* \to \{1, 0\}$$

where $f_{\mathsf{M}}(x) = 1$ exactly when $x \in L(\mathsf{M})$.

- There are several generalizations of DFAs that are useful in various contexts. A first important generalization is to add a nondeterministic capability to the machines. A nondeterministic finite automaton (NFA) $\mathsf{M} = (\mathsf{S}, \mathsf{A}, s_0, \delta, \mathsf{F})$ is the same as a DFA except for the transition function:

S, a finite set of states,

A, an alphabet,

$s_0 \in \mathsf{S}$, the start state,

$\delta : \mathsf{S}\mathsf{x}\mathsf{A} \rightarrow P(\mathsf{S})$, the transition function,

$\mathsf{F} \subset \mathsf{S}$, the set of final (or accepting) states of the machine.

For a given input symbol, the transition function can take us to any one of a set of states.

We extend the transition function to $\widehat{\delta} : \mathsf{S}\mathsf{x}\mathsf{A}^* \rightarrow P(\mathsf{S})$ in much the same way:

$$\begin{aligned} \widehat{\delta}(s, \epsilon) &= s, \\ \widehat{\delta}(s, xa) &= \bigcup_{r \in \delta(s,a)} \widehat{\delta}(r, x). \end{aligned}$$

We define the language of the machine by

$$L(\mathsf{M}) = \{x \in \mathsf{A}^* \mid \widehat{\delta}(s_0, x) \cap \mathsf{F} \neq \emptyset\}.$$

A useful fact is that DFAs and NFAs define the same class of languages. In particular, given a language L, we have that L $= L($M$)$ for some DFA M if and only if L $= L($M$')$ for some NFA M$'$.

Exercise: Prove this fact.

In doing the proof, you will notice that if L $= L($M$) = L($M$')$ for some DFA M and NFA M$'$, and M$'$ has n states, then M might need to have as many as 2^n states. In general, NFAs are relatively easy to write down, but DFAs can be directly implemented.

- Another useful generalization is to allow the machine to change states without any input (often called ϵ-moves). An NFA with ϵ-moves would be defined similarly to an NFA, but with transition function

$$\delta : \mathsf{S}\mathsf{x}(\mathsf{A} \cup \{\epsilon\}) \to P(\mathsf{S}).$$

Exercise: What would an appropriate extended transition function $\widehat{\delta}$ and language L(M) be for an NFA with ϵ-moves?

Exercise: Show that the class of languages defined by NFAs with ϵ-moves is the same as that defined by DFAs and NFAs.

Here is a simple example. By convention, the states in F are double circled. Labelled arrows indicate transitions. Exercise: what is the language of this machine?

Regular expressions and languages ←

- In the preceding section we defined a class of machines (Finite Automata) that can be used to recognize members of a particular class of languages. It would be nice to have a concise way to describe such a language, and furthermore to have a convenient way to generate strings in such a language (as opposed to having to feed candidate strings into a machine, and hoping they are recognized as being in the language . . .).

  Fortunately, there is a nice way to do this. The class of languages defined by Finite Automata are called Regular Languages (or Regular Sets of strings). These languages are described by Regular Expressions. We define these as follows. We will use lower case letters for regular expressions, and upper case for regular sets.

- Definition: Given an alphabet A, the following are regular expressions / regular sets over A:

Expressions :	Sets :
$\emptyset$	$\emptyset$
ϵ	$\{\epsilon\}$
a, for $a \in$ A	$\{a\}$, for $a \in$ A
If r and s are regular, then so are :	If R and S are regular, then so are :
r + s	R $\cup$ S
rs	RS
r*	R*

and nothing else is regular.

We say that the regular expression on the left represents the corresponding regular set on the right. We say that a language is a regular language if the set of strings in the language is a regular set.

- A brief example:

  The regular expression

  $$(00+11)^*(101+110)$$

  represents the regular set (regular language)

  $\{101, 110, 00101, 00110, 11101, 11110, 0000101, 0000110, 0011101, 0011110, \ldots\}$.

  Exercise: What are some other strings in this language? Is 00110011110 in the language? How about 00111100101110?

- It is a nice fact that regular languages are exactly the languages of the finite automata defined in the previous section. In particular, a language L is a regular set (as defined above) if and only if L $= L($M$)$ for some finite automaton M.

  The proof of this fact is relatively straightforward.

  For the first half, we need to show that if L is a regular set (in particular, if it is represented by a regular expression r), then L $= L($M$)$ for some finite automaton M. We can show this by induction on the size of the regular expression r. The basis for the induction is the three simplest cases: $\emptyset$, $\{\epsilon\}$, and $\{a\}$. (Exercise: find machines for these three cases.) We then show that, if we know how to build machines for R and S, then we can build machines for R $\cup$ S, RS, and R*. (Exercise: Show how to do these three – use NFAs with ϵ-moves.)

For the second half, we need to show that if we are given a DFA M, then we can find a regular expression (or a regular set representation) for $L(\mathsf{M})$. We can do this by looking at sets of strings of the form

$\mathsf{R}^k_{ij} = \{x \in \mathsf{A}^* \mid x$ takes M from state s_i to state s_j without going through (into and out of) any state s_m with $m \geq k\}$.

Note that if the states of M are $\{s_0, s_1, \ldots, s_{n-1}\}$, then

$$L(\mathsf{M}) = \bigcup_{s_j \in \mathsf{F}} \mathsf{R}^n_{0j}.$$

We also have

$$\mathsf{R}^0_{ij} = \{a \in \mathsf{A} \mid \delta(s_i, a) = s_j\}$$

(for $i = j$, we also get ϵ . . .),

and, for $k \geq 0$,

$$\mathsf{R}^{k+1}_{ij} = \mathsf{R}^k_{ij} \cup \mathsf{R}^k_{ik}(\mathsf{R}^k_{kk})^*\mathsf{R}^k_{kj}.$$

Exercise: Verify, and finish the proof.

Markov models

←

- An important related class of systems are Markov models (often called Markov chains). These models are quite similar to finite automata, except that the transitions from state to state are probabilistically determined, and typically we do not concern ourselves with final or accepting states.

  Markov models can often be thought of as models for discrete dynamical systems. We model the system as consisting of a finite set of states, with a certain probability of transition from a state to any other state.

The typical way to specify a Markov model is via a transition matrix:

$$\mathsf{T} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ p_{21} & p_{22} & \cdots & p_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n1} & p_{n2} & \cdots & p_{nn} \end{bmatrix}$$

where $0 \leq p_{ij} \leq 1$, and $\sum_j p_{ij} = 1$.

Each entry p_{ij} tells the probability the system will go from state s_i to state s_j in the next time step.

The transition probabilities over two steps are given by T^2. Over n steps, the probabilities are given by T^n.

(Examples)

(Hidden Markov models)

(Examples - voice recognition, biological sequences . . .)

Context free grammars and languages

←

- While regular languages are very useful, not every interesting language is regular. It is not hard to show that even such simple languages as balanced parentheses or palindromes are not regular. (Here is probably a good place to remind ourselves again that in this context, a language is just a set of strings . . .)

  A more general class of languages is the context free languages. A straightforward way to specify a context free language is via a context free grammar. Context free grammars can be thought of as being for context free languages the analogue of regular expressions for regular languages. They provide a mechanism for generating elements on the language.

- A context free grammar G = (V, T, S, P) consists of a two alphabets V and T (called variables and terminals, respectively), an element S $\in$ V called the start symbol, and a finite set of production rules P. Each production rule is of the form $A \to \alpha$, where A $\in$ V and $\alpha \in (\mathsf{V} \cup \mathsf{T})^*$.

  We can use such a production rule to generate new strings from old strings. In particular, if we have the string $\gamma A \delta \in (\mathsf{V} \cup \mathsf{T})^*$ with A $\in$ V, then we can produce the new string $\gamma\alpha\delta$. The application of a production rule from the grammar G is often written $\alpha \underset{\mathsf{G}}{\Rightarrow} \beta$, or just $\alpha \Rightarrow \beta$ if the grammar is clear. The application of 0 or more production rules one after the other is written $\alpha \overset{*}{\Rightarrow} \beta$.

  The language of the grammar G is then

  $$L(\mathsf{G}) = \{\alpha \in \mathsf{T}^* \mid \mathsf{S} \overset{*}{\Rightarrow} \alpha\}.$$

  The language of such a grammar is called a context free language.

- Here, as an example, is the language consisting of strings of balanced parenthese. Note that we can figure out which symbols are variables, since they all occur on the left side of some production.

$$\begin{aligned} \mathsf{S} &\rightarrow \mathsf{R} \\ \mathsf{R} &\rightarrow \epsilon \\ \mathsf{R} &\rightarrow (R) \\ \mathsf{R} &\rightarrow RR \end{aligned}$$

  Exercise: Check this. How would we modify the grammar if we wanted to include balanced '[]' and '{}' pairs also?

- Here is a palindrome language over the alphabet T = {a, b}:

$$\begin{aligned} \mathsf{S} &\rightarrow \mathsf{R} \\ \mathsf{R} &\rightarrow \epsilon \mid \mathsf{a} \mid \mathsf{b} \\ \mathsf{R} &\rightarrow \mathsf{aRa} \mid \mathsf{bRb} \end{aligned}$$

  (note the '|' to indicate alternatives . . .)

Exercise: What would a grammar for simple algebraic expressions (with terminals T = {x, y, +, -, *, (,)}) look like?

- A couple of important facts are that any regular language is also context free, but there are context free languages that are not regular.

  Exercise: How might one prove these facts?

- Context free grammars can easily be used to generate strings in the corresponsing language. We would also like to have machines to recognize such languages. We can build such machines through a slight generalization of finite automata. The generalization is to add a ‘pushdown stack’ to our finite automata. These more powerful machines are called pushdown automata, or PDAs . . .

-

Language generators and recognizers ←

- First, note that the function ln(x) has derivative $1/x$. From this, we find that the tangent to ln(x) at $x = 1$ is the line $y = x - 1$. Further, since ln(x) is concave down, we have, for $x > 0$, that

  $$\ln(x) \leq x - 1,$$

  with equality only when $x = 1$.

  Now, given two probability distributions, $P = \{p_1, p_2, \ldots, p_n\}$ and $Q = \{q_1, q_2, \ldots, q_n\}$, where $p_i, q_i \geq 0$ and $\sum_i p_i = \sum_i q_i = 1$, we have

  $$\begin{aligned} \sum_{i=1}^{n} p_i \ln\left(\frac{q_i}{p_i}\right) &\leq \sum_{i=1}^{n} p_i \left(\frac{q_i}{p_i} - 1\right) = \sum_{i=1}^{n} (q_i - p_i) \\ &= \sum_{i=1}^{n} q_i - \sum_{i=1}^{n} p_i = 1 - 1 = 0, \end{aligned}$$

  with equality only when $p_i = q_i$ for all i. It is easy to see that the inequality actually holds for any base, not just e.

The Chomsky hierarchy ←

- Let us work briefly with a simple model for an idealized gas. Let us assume that the gas is made up of N point particles, and that at some time t_0 all the particles are contained within a (cubical) volume V. Assume that through some mechanism, we can determine the location of each particle sufficiently well as to be able to locate it within a box with sides 1/100 of the sides of the containing volume V. There are 10^6 of these small boxes within V.

Turing machines ←

- In his classic 1948 papers, Claude Shannon laid the foundations for contemporary *information*, *coding*, and *communication* theory. He developed a general model for communication systems, and a set of theoretical tools for analyzing such systems.

  His basic model consists of three parts: a sender (or source), a channel, and a receiver (or sink). His general model also includes encoding and decoding elements, and noise within the channel.

Computability and tractability ←

- We have already observed that there are some problems that are not computable – in particular, we showed the existence of languages for which there cannot be an algorithmic recognizer to determine which strings are in the language. Another important example of a noncomputable problem is the so-called halting problem. In simple terms, the question is, given a computer program, does the program contain an infinite loop? There cannot be an algorithm that is guaranteed to correctly answer this question for all programs.

  More practically, however, we often are interested in whether a program can be executed in a 'reasonable' length of time, using a reasonable amount of resources such as system memory.

- We can generally categorize computational algorithms according to how the resources needed for execution of the algorithm increase as we increase the size of the input. Typical resources are time and (storage) space. In different contexts, we may be interested in worst-case or average-case performance of the algorithm. For theoretical purposes, we will typically be interested in large input sets . . .

- A standard mechanism for comparing the growth of functions with domain $\mathbb{N}$ is "big-Oh." One way of defining this notion is to associate each function with a set of functions. We can then compare algorithms by looking at their "big-Oh" categories.

- Given a function f, we define $O(f)$ by:

$$g \in O(f) \iff$$

$$\text{there exist } c > 0 \text{ and } N \geq 0 \text{ such that } |g(n)| \leq c|f(n)| \text{ for all } n \geq N.$$

- We further define $\theta(f)$ by:
  $g \in \theta(f)$ iff $g \in O(f)$ and $f \in O(g)$.

- In general we will consider the run-time of algorithms in terms of the growth of the number of elementary computer operations as a function of the number of bits in the (encoded) input. Some important categories – an algorithm's run-time f is:

  1. Logarithmic if $f \in \theta(\log(n))$.
  2. Linear if $f \in \theta(n)$.
  3. Quadratic if $f \in \theta(n^2)$.
  4. Polynomial if $f \in \theta(P(n))$ for some polynomial $P(n)$.
  5. Exponential if $f \in \theta(b^n)$ for some constant $b > 1$.
  6. Factorial if $f \in \theta(n!)$.

- Typically we say that a problem is *tractable* if (we know) there exists an algorithm whose run-time is (at worst) polynomial that solves the problem. Otherwise, we call the problem *intractable*.

- There are many problems which have the interesting property that if someone (an oracle?) provides you with a solution to the problem, you can tell in polynomial time whether what they provided you actually is a solution. Problems with this property are called Non-deterministically Polynomial, or NP, problems. One way to think about this property is to imagine that we have arbitrarily many machines available. We let each machine work on one possible solution, and whichever machine finds the (a) solution lets us know.

- There are some even more interesting NP problems which are universal for the class of NP problems. These are called NP-complete problems. A problem S is NP-complete if S is NP and, there exists a polynomial time algorithm that allows us to translate any NP problem into an instance of S. If we could find a polynomial time algorithm to solve a single NP-complete problem, we would then have a polynomial time solution for each NP problem.

- Some examples:

  1. Factoring a number is NP. First, we recognize that if M is the number we want to factor, then the input size m is approximately log(M) (that is, the input size is the number of digits in the number). The elementary school algorithm (try dividing by each number less than $\sqrt{M}$) has run-time approximately $10^{\frac{m}{2}}$, which is exponential in the number of digits. On the other hand, if someone hands you two numbers they claim are factors of M, you can check by multiplying, which takes on the order of m^2 operations.

     It is worth noting that there is a polynomial time algorithm to determine whether or not a number is prime, but for composite numbers, this algorithm does not provide a factorization. Factoring is a particularly important example because various encryption algorithms such as RSA (used in the PGP software) depend for their security on the difficulty of factoring numbers with several hundred digits.

2. Satisfiability of a boolean expression is NP-complete. Suppose we have n boolean variables $\{b_1, b_2, \ldots, b_n\}$ (each with the possible values 0 and 1). We can form a general boolean expression from these variables and their negations:

$$f(b_1, b_2, \ldots, b_n) = \bigwedge_k \Big(\bigvee_{i,j \le n} (b_i, \sim b_j) \Big).$$

A solution to such a problem is an assignment of values 0 or 1 to each of the b_i such that $f(b_1, b_2, \ldots, b_n) = 1$. There are 2^n possible assignments of values. We can check an individual possible solution in polynomial time, but there are exponentially many possibilities to check. If we could develop a feasible computation for this problem, we would have resolved the traditional $\mathrm{P} \overset{?}{=} \mathrm{NP}$ problem . . .

Computational complexity

←

- Suppose we have a system for which we can measure certain macroscopic characteristics. Suppose further that the system is made up of many microscopic elements, and that the system is free to vary among various states. Given the discussion above, let us assume that with probability essentially equal to 1, the system will be observed in states with maximum entropy.

  We will then sometimes be able to gain understanding of the system by applying a *maximum information entropy* principle, and, using Lagrange multipliers, derive formulas for aspects of the system.

To top ←

References

[1] Brillouin, L., *Science and information theory* Academic Press, New York, 1956.

[2] Brooks, Daniel R., and Wiley, E. O., *Evolution as Entropy*, Toward a Unified Theory of Biology, Second Edition, University of Chicago Press, Chicago, 1988.

[3] Campbell, Jeremy, *Grammatical Man*, Information, Entropy, Language, and Life, Simon and Schuster, New York, 1982.

[4] Cover, T. M., and Thomas J. A., *Elements of Information Theory,* John Wiley and Sons, New York, 1991.

[5] DeLillo, Don, *White Noise*, Viking/Penguin, New York, 1984.

[6] Feller, W., *An Introduction to Probability Theory and Its Applications*, Wiley, New York,1957.

[7] Feynman, Richard, *Feynman lectures on computation*, Addison-Wesley, Reading, 1996.

[8] Gatlin, L. L., *Information Theory and the Living System*, Columbia University Press, New York, 1972.

[9] Haken, Hermann, *Information and Self-Organization, a Macroscopic Approach to Complex Systems*, Springer-Verlag, Berlin/New York, 1988.

[10] Hamming, R. W., Error detecting and error correcting codes, *Bell Syst. Tech. J.* **29** 147, 1950.

[11] Hamming, R. W., *Coding and information theory*, 2nd ed, Prentice-Hall, Englewood Cliffs, 1986.

[12] Hill, R., *A first course in coding theory* Clarendon Press, Oxford, 1986.

[13] Hodges, A., *Alan Turing: the enigma* Vintage, London, 1983.

[14] Hofstadter, Douglas R., *Metamagical Themas: Questing for the Essence of Mind and Pattern*, Basic Books, New York, 1985

[15] Jones, D. S., *Elementary information theory* Clarendon Press, Oxford, 1979.

[16] Knuth, Eldon L., *Introduction to Statistical Thermodynamics*, McGraw-Hill, New York, 1966.

[17] Landauer, R., Information is physical, *Phys. Today*, May 1991 23-29.

[18] Landauer, R., The physical nature of information, *Phys. Lett. A*, **217** 188, 1996.

[19] van Lint, J. H., *Coding Theory*, Springer-Verlag, New York/Berlin, 1982.

[20] Lipton, R. J., Using DNA to solve NP-complete problems, *Science*, **268** 542–545, Apr. 28, 1995.

[21] MacWilliams, F. J., and Sloane, N. J. A., *The theory of error correcting codes*, Elsevier Science, Amsterdam, 1977.

[22] Martin, N. F. G., and England, J. W., *Mathematical Theory of Entropy*, Addison-Wesley, Reading, 1981.

[23] Maxwell, J. C., *Theory of heat* Longmans, Green and Co, London, 1871.

[24] von Neumann, John, Probabilistic logic and the synthesis of reliable organisms from unreliable components, in *automata studies(Shanon,McCarthy eds)*, 1956 .

[25] Papadimitriou, C. H., *Computational Complexity*, Addison-Wesley, Reading, 1994.

[26] Pierce, John R., *An Introduction to Information Theory – Symbols, Signals and Noise*, (second revised edition), Dover Publications, New York, 1980.

[27] Roman, Steven, *Introduction to Coding and Information Theory*, Springer-Verlag, Berlin/New York, 1997.

[28] Sampson, Jeffrey R., *Adaptive Information Processing, an Introductory Survey*, Springer-Verlag, Berlin/New York, 1976.

[29] Schroeder, Manfred, *Fractals, Chaos, Power Laws, Minutes from an Infinite Paradise*, W. H. Freeman, New York, 1991.

[30] Shannon, C. E., A mathematical theory of communication *Bell Syst. Tech. J.* **27** 379; also p. 623, 1948.

[31] Slepian, D., ed., *Key papers in the development of information theory* IEEE Press, New York, 1974.

[32] Turing, A. M., On computable numbers, with an application to the Entscheidungsproblem, *Proc. Lond. Math. Soc. Ser. 2* **42**, 230 ; see also *Proc. Lond. Math. Soc. Ser. 2* **43**, 544, 1936.

[33] Zurek, W. H., Thermodynamic cost of computation, algorithmic complexity and the information metric, *Nature* **341** 119-124, 1989.

To top ←