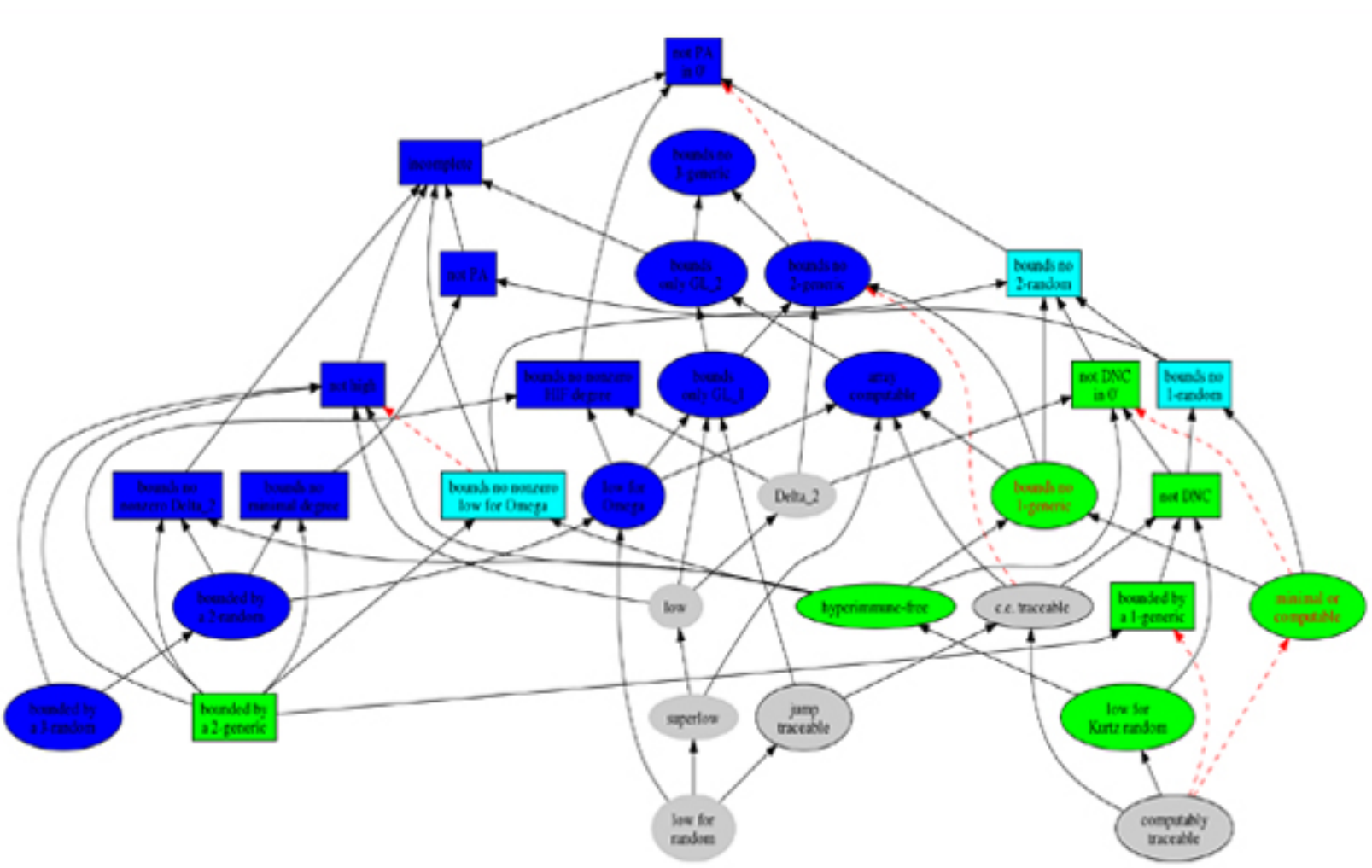
Computability Theory & Automata Theory



Kristy Plante

Tomas Jensen

First Edition, 2012

ISBN 978-81-323-1266-6

Published by:
College Publishing House
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Introduction to Computability Theory

Computability theory, also called **recursion theory**, is a branch of mathematical logic that originated in the 1930s with the study of computable functions and Turing degrees. The field has grown to include the study of generalized computability and definability. In these areas, recursion theory overlaps with proof theory and effective descriptive set theory.

The basic questions addressed by recursion theory are "What does it mean for a function from the natural numbers to themselves to be computable?" and "Can noncomputable functions be classified into a hierarchy based on their level of noncomputability?". The answers to these questions have led to a rich theory that is still being actively researched.

The field is also closely related to computer science. Recursion theorists in mathematical logic often study the theory of relative computability, reducibility notions and degree structures described here. This contrasts with the theory of subrecursive hierarchies, formal methods and formal languages that is common in the study of computability theory in computer science. There is considerable overlap in knowledge and methods between these two research communities, however, and no firm line can be drawn between them.

Computable and uncomputable sets

Recursion theory originated with work of Kurt Gödel, Alonzo Church, Alan Turing, Stephen Kleene and Emil Post in the 1930s.

The fundamental results the researchers obtained established Turing computability as the correct formalization of the informal idea of effective calculation. These results led Stephen Kleene (1952) to coin the two names "Church's thesis" (Kleene 1952:300) and "Turing's Thesis" (p. 376). Nowadays these are often considered as a single hypothesis, the **Church–Turing thesis**, which states that any function that is computable by an algorithm is a computable function. Although initially skeptical, by 1946 Gödel argued in favor of this thesis.

"Tarski has stressed in his lecture (and I think justly) the great importance of the concept of general recursiveness (or Turing's computability). It seems to me that this importance is largely due to the fact that with this concept one has for the first time succeeded in

giving an absolute notion to an interesting epistemological notion, i.e., one not depending on the formalism chosen."(Gödel 1946 in Davis 1965: 84)

With a definition of effective calculation came the first proofs that there are problems in mathematics that cannot be effectively decided. Church (1936a, 1936b) and Turing (1936), inspired by techniques used by Gödel (1931) to prove his incompleteness theorems, independently demonstrated that the Entscheidungsproblem is not effectively decidable. This result showed that there is no algorithmic procedure that can correctly decide whether arbitrary mathematical propositions are true or false.

Many problems of mathematics have been shown to be undecidable after these initial examples were established. In 1947, Markov and Post published independent papers showing that the word problem for semigroups cannot be effectively decided. Extending this result, Pyotr Novikov and William Boone showed independently in the 1950s that the word problem for groups is not effectively solvable: there is no effective procedure that, given a word in a finitely presented group, will decide whether the element represented by the word is the identity element of the group. In 1970, Yuri Matiyasevich proved Matiyasevich's theorem, which implies that Hilbert's tenth problem has no effective solution; this problem asked whether there is an effective procedure to decide whether a Diophantine equation over the integers has a solution in the integers. The list of undecidable problems gives additional examples of problems with no computable solution.

The study of which mathematical constructions can be effectively performed is sometimes called **recursive mathematics**; the *Handbook of Recursive Mathematics* (Ershov *et al.* 1998) covers many of the known results in this field.

Turing computability

The main form of computability studied in recursion theory was introduced by Turing (1936). A set of natural numbers is said to be a **computable set** (also called a **decidable**, **recursive**, or **Turing computable** set) if there is a Turing machine that, given a number n, halts with output 1 if n is in the set and halts with output 0 if n is not in the set. A function f from the natural numbers to themselves is a **recursive** or **(Turing) computable function** if there is a Turing machine that, on input n, halts and returns output $f(n)$. The use of Turing machines here is not necessary; there are many other models of computation that have the same computing power as Turing machines; for example the μ-recursive functions obtained from primitive recursion and the μ operator.

The terminology for recursive functions and sets is not completely standardized. The definition in terms of μ-recursive functions as well as a different definition of *rekursiv* functions by Gödel led to the traditional name *recursive* for sets and functions computable by a Turing machine. The word **decidable** stems from the German word **Entscheidungsproblem** which was used in the original papers of Turing and others. In contemporary use, the term "computable function" has various definitions: according to Cutland (1980), it is a partial recursive function (which can be undefined for some

inputs), while according to Soare (1987) it is a total recursive (equivalently, general recursive) function.

Not every set of natural numbers is computable. The halting problem, which is the set of (descriptions of) Turing machines that halt on input 0, is a well known example of a noncomputable set. The existence of many noncomputable sets follows from the facts that there are only countably many Turing machines, and thus only countably many computable sets, but there are uncountably many sets of natural numbers.

Although the Halting problem is not computable, it is possible to simulate program execution and produce an infinite list of the programs that do halt. Thus the halting problem is an example of a **recursively enumerable set**, which is a set that can be enumerated by a Turing machine (other terms for recursively enumerable include **computably enumerable** and **semidecidable**). Equivalently, a set is recursively enumerable if and only if it is the range of some computable function. The recursively enumerable sets, although not decidable in general, have been studied in detail in recursion theory.

Areas of research in recursion theory

Beginning with the theory of recursive sets and functions described above, the field of recursion theory has grown to include the study of many closely related topics. These are not independent areas of research: each of these areas draws ideas and results from the others, and most recursion theorists are familiar with the majority of them.

Other reducibilities

An ongoing area of research in recursion theory studies reducibility relations other than Turing reducibility. Post (1944) introduced several **strong reducibilities**, so named because they imply truth-table reducibility. A Turing machine implementing a strong reducibility will compute a total function regardless of which oracle it is presented with. **Weak reducibilities** are those where a reduction process may not terminate for all oracles; Turing reducibility is one example.

The strong reducibilities include:

- One-one reducibility: A is **one-one reducible** (or **1-reducible**) to B if there is a total computable injective function f such that each n is in A if and only if $f(n)$ is in B.
- Many-one reducibility: This is essentially one-one reducibility without the constraint that f be injective. A is **many-one reducible** (or **m-reducible**) to B if there is a total computable function f such that each n is in A if and only if $f(n)$ is in B.
- Truth-table reducibility: A is truth-table reducible to B if A is Turing reducible to B via an oracle Turing machine that computes a total function regardless of the oracle it is given. Because of compactness of Cantor space, this is equivalent to

saying that the reduction presents a single list of questions (depending only on the input) to the oracle simultaneously, and then having seen their answers is able to produce an output without asking additional questions regardless of the oracle's answer to the initial queries. Many variants of truth-table reducibility have also been studied.

Further reducibilities (positive, disjunctive, conjunctive, linear and their weak and bounded versions) are discussed in the article Reduction (recursion theory).

The major research on strong reducibilities has been to compare their theories, both for the class of all recursively enumerable sets as well as for the class of all subsets of the natural numbers. Furthermore, the relations between the reducibilities has been studied. For example, it is known that every Turing degree is either a truth-table degree or is the union of infinitely many truth-table degrees.

Reducibilities weaker than Turing reducibility (that is, reducibilities that are implied by Turing reducibility) have also been studied. The most well known are arithmetical reducibility and hyperarithmetical reducibility. These reducibilities are closely connected to definability over the standard model of arithmetic.

Rice's theorem and the arithmetical hierarchy

Rice showed that for every nontrivial class C (which contains some but not all r.e. sets) the index set $E = \{e$: the eth r.e. set W_e is in $C\}$ has the property that either the halting problem or its complement is many-one reducible to E, that is, can be mapped using a many-one reduction to E. But, many of these index sets are even more complicated than the halting problem. These type of sets can be classified using the arithmetical hierarchy. For example, the index set FIN of class of all finite sets is on the level Σ_2, the index set REC of the class of all recursive sets is on the level Σ_3, the index set COFIN of all cofinite sets is also on the level Σ_3 and the index set COMP of the class of all Turing-complete sets Σ_4. These hierarchy levels are defined inductively, Σ_{n+1} contains just all sets which are recursively enumerable relative to Σ_n; Σ_1 contains the recursively enumerable sets. The index sets given here are even complete for their levels, that is, all the sets in these levels can be many-one reduced to the given index sets.

Numberings

A numbering is an enumeration of functions; it has two parameters, e and x and outputs the value of the e-th function in the numbering on the input x. Numberings can be partial-recursive although some of its members are total recursive, that is, computable functions. Acceptable or Gödel numberings are those into which all others can be translated. A Friedberg numbering (named after its discoverer) is a one-one numbering of all partial-recursive functions; it is necessarily not an acceptable numbering. Later research dealt also with numberings of other classes like classes of recursively enumerable sets. Goncharov discovered for example a class of recursively enumerable sets for which the numberings fall into exactly two classes with respect to recursive isomorphisms.

The priority method

Post's problem was solved with a method called the **priority method**; a proof using this method is called a **priority argument**. This method is primarily used to construct recursively enumerable sets with particular properties. To use this method, the desired properties of the set to be constructed are broken up into an infinite list of goals, known as **requirements**, so that satisfying all the requirements will cause the set constructed to have the desired properties. Each requirement is assigned to a natural number representing the priority of the requirement; so 0 is assigned to the most important priority, 1 to the second most important, and so on. The set is then constructed in stages, each stage attempting to satisfy one of more of the requirements by either adding numbers to the set or banning numbers from the set so that the final set will satisfy the requirement. It may happen that satisfying one requirement will cause another to become unsatisfied; the priority order is used to decide what to do in such an event.

Priority arguments have been employed to solve many problems in recursion theory, and have been classified into a hierarchy based on their complexity (Soare 1987). Because complex priority arguments can be technical and difficult to follow, it has traditionally been considered desirable to prove results without priority arguments, or to see if results proved with priority arguments can also be proved without them. For example, Kummer published a paper on a proof for the existence of Friedberg numberings without using the priority method.

The lattice of recursively enumerable sets

When Post defined the notion of a simple set as an r.e. set with an infinite complement not containing any infinite r.e. set, he started to study the structure of the recursively enumerable sets under inclusion. This lattice became a well-studied structure. Recursive sets can be defined in this structure by the basic result that a set is recursive if and only if the set and its complement are both recursively enumerable. Infinite r.e. sets have always infinite recursive subsets; but on the other hand, simple sets exist but do not have a coinfinite recursive superset. Post (1944) introduced already hypersimple and hyperhypersimple sets; later maximal sets were constructed which are r.e. sets such that every r.e. superset is either a finite variant of the given maximal set or is co-finite. Post's original motivation in the study of this lattice was to find a structural notion such that every set which satisfies this property is neither in the Turing degree of the recursive sets nor in the Turing degree of the halting problem. Post did not find such a property and the solution to his problem applied priority methods instead; Harrington and Soare (1991) found eventually such a property.

Automorphism problems

Another important question is the existence of automorphisms in recursion-theoretic structures. One of these structures is that one of recursively enumerable sets under inclusion modulo finite difference; in this structure, A is below B if and only if the set difference $B - A$ is finite. Maximal sets (as defined in the previous paragraph) have the

property that they cannot be automorphic to non-maximal sets, that is, if there is an automorphism of the recursive enumerable sets under the structure just mentioned, then every maximal set is mapped to another maximal set. Soare (1974) showed that also the converse holds, that is, every two maximal sets are automorphic. So the maximal sets form an orbit, that is, every automorphism preserves maximality and any two maximal sets are transformed into each other by some automorphism. Harrington gave a further example of an automorphic property: that of the creative sets, the sets which are many-one equivalent to the halting problem.

Besides the lattice of recursively enumerable sets, automorphisms are also studied for the structure of the Turing degrees of all sets as well as for the structure of the Turing degrees of r.e. sets. In both cases, Cooper claims to have constructed nontrivial automorphisms which map some degrees to other degrees; this construction has, however, not been verified and some colleagues believe that the construction contains errors and that the question of whether there is a nontrivial automorphism of the Turing degrees is still one of the main unsolved questions in this area (Slaman and Woodin 1986, Ambos-Spies and Fejer 2006).

Frequency computation

This branch of recursion theory analyzed the following question: For fixed m and n with $0 < m < n$, for which functions A is it possible to compute for any different n inputs $x_1, x_2, ..., x_n$ a tuple of n numbers $y_1,y_2,...,y_n$ such that at least m of the equations $A(x_k) = y_k$ are true. Such sets are known as (m, n)-recursive sets. The first major result in this branch of Recursion Theory is Trakhtenbrot's result that a set is computable if it is (m, n)-recursive for some m, n with $2m > n$. On the other hand, Jockusch's semirecursive sets (which were already known informally before Jockusch introduced them 1968) are examples of a set which is (m, n)-recursive if and only if $2m < n + 1$. There are uncountably many of these sets and also some recursively enumerable but noncomputable sets of this type. Later, Degtev established a hierarchy of recursively enumerable sets that are $(1, n + 1)$-recursive but not $(1, n)$-recursive. After a long phase of research by Russian scientists, this subject became repopularized in the west by Beigel's thesis on bounded queries, which linked frequency computation to the above mentioned bounded reducibilities and other related notions. One of the major results was Kummer's Cardinality Theory which states that a set A is computable if and only if there is an n such that some algorithm enumerates for each tuple of n different numbers up to n many possible choices of the cardinality of this set of n numbers intersected with A; these choices must contain the true cardinality but leave out at least one false one.

Inductive inference

This is the recursion-theoretic branch of learning theory. It is based on Gold's model of learning in the limit from 1967 and has developed since then more and more models of learning. The general scenario is the following: Given a class S of computable functions, is there a learner (that is, recursive functional) which outputs for any input of the form $(f(0),f(1),...,f(n))$ a hypothesis. A learner M learns a function f if almost all hypotheses are

the same index *e* of *f* with respect to a previously agreed on acceptable numbering of all computable functions; *M* learns *S* if *M* learns every *f* in *S*. Basic results are that all recursively enumerable classes of functions are learnable while the class REC of all computable functions is not learnable. Many related models have been considered and also the learning of classes of recursively enumerable sets from positive data is a topic studied from Gold's pioneering paper in 1967 onwards.

Generalizations of Turing computability

Recursion theory includes the study of generalized notions of this field such as arithmetic reducibility, hyperarithmetical reducibility and α-recursion theory, as described by Sacks (1990). These generalized notions include reducibilities that cannot be executed by Turing machines but are nevertheless natural generalizations of Turing reducibility. These studies include approaches to investigate the analytical hierarchy which differs from the arithmetical hierarchy by permitting quantification over sets of natural numbers in addition to quantification over individual numbers. These areas are linked to the theories of well-orderings and trees; for example the set of all indices of recursive (nonbinary) trees without infinite branches is complete for level Π^1_1 of the analytical hierarchy. Both Turing reducibility and hyperarithmetical reducibility are important in the field of effective descriptive set theory. The even more general notion of degrees of constructibility is studied in set theory.

Continuous computability theory

Computability theory for digital computation is well developed. Computability theory is less well developed for analog computation that occurs in analog computers, analog signal processing, analog electronics, neural networks and continuous-time control theory, modelled by differential equations and continuous dynamical systems.

Relationships between definability, proof and computability

There are close relationships between the Turing degree of a set of natural numbers and the difficulty (in terms of the arithmetical hierarchy) of defining that set using a first-order formula. One such relationship is made precise by Post's theorem. A weaker relationship was demonstrated by Kurt Gödel in the proofs of his completeness theorem and incompleteness theorems. Gödel's proofs show that the set of logical consequences of an effective first-order theory is a recursively enumerable set, and that if the theory is strong enough this set will be uncomputable. Similarly, Tarski's indefinability theorem can be interpreted both in terms of definability and in terms of computability.

Recursion theory is also linked to second order arithmetic, a formal theory of natural numbers and sets of natural numbers. The fact that certain sets are computable or relatively computable often implies that these sets can be defined in weak subsystems of second order arithmetic. The program of reverse mathematics uses these subsystems to measure the noncomputability inherent in well known mathematical theorems.

Simpson (1999) discusses many aspects of second-order arithmetic and reverse mathematics.

The field of proof theory includes the study of second-order arithmetic and Peano arithmetic, as well as formal theories of the natural numbers weaker than Peano arithmetic. One method of classifying the strength of these weak systems is by characterizing which computable functions the system can prove to be total. For example, in primitive recursive arithmetic any computable function that is provably total is actually primitive recursive, while Peano arithmetic proves that functions like the Ackerman function, which are not primitive recursive, are total. Not every total computable function is provably total in Peano arithmetic, however; an example of such a function is provided by Goodstein's theorem.

Name of the subject

The field of mathematical logic dealing with computability and its generalizations has been called "recursion theory" since its early days. Robert I. Soare, a prominent researcher in the field, has proposed (Soare 1996) that the field should be called "computability theory" instead. He argues that Turing's terminology using the word "computable" is more natural and more widely understood than the terminology using the word "recursive" introduced by Kleene. Many contemporary researchers have begun to use this alternate terminology. These researchers also use terminology such as *partial computable function* and *computably enumerable* (*c.e.*) *set* instead of *partial recursive function* and *recursively enumerable* (*r.e.*) *set*. Not all researchers have been convinced, however, as explained by Fortnow and Simpson. Some commentators argue that both the names *recursion theory* and *computability theory* fail to convey the fact that most of the objects studied in recursion theory are not computable.

Rogers (1967) has suggested that a key property of recursion theory is that its results and structures should be invariant under computable bijections on the natural numbers (this suggestion draws on the ideas of the Erlangen program in geometry). The idea is that a computable bijection merely renames numbers in a set, rather than indicating any structure in the set, much as a rotation of the Euclidean plane does not change any geometric aspect of lines drawn on it. Since any two infinite computable sets are linked by a computable bijection, this proposal identifies all the infinite computable sets (the finite computable sets are viewed as trivial). According to Rogers, the sets of interest in recursion theory are the noncomputable sets, partitioned into equivalence classes by computable bijections of the natural numbers.

Chapter 2

Turing Reduction and Turing Degree

Turing reduction

In computability theory, a **Turing reduction** from a problem *A* to a problem *B*, named after Alan Turing, is a reduction which solves *A*, assuming *B* is already known (Rogers 1967, Soare 1987). It can be understood as an algorithm that could be used to solve *A* if it had available to it a subroutine for solving *B*. More formally, a Turing reduction is a function computable by an oracle machine with an oracle for *B*. Turing reductions can be applied to both decision problems and function problems.

If a Turing reduction of *A* to *B* exists then every algorithm for *B* can be used to produce an algorithm for *A*, by inserting the algorithm for *B* at each place where the oracle machine computing *A* queries the oracle for *B*. However, because the oracle machine may query the oracle a large number of times, the resulting algorithm may require more time asymptotically than either *M* or the oracle machine, and may require as much space as both together.

The first formal definition of relative computability, then called relative reducibility, was given by Alan Turing in 1939 in terms of oracle machines. Later in 1943 and 1952 Stephen Kleene defined an equivalent concept in terms of recursive functions. In 1944 Emil Post used the term "Turing reducibility" to refer to the concept.

Definition

Given two sets $A, B \subseteq \mathbb{N}$ of natural numbers, we say *A* is **Turing reducible** to *B* and write

$$A \leq_T B$$

if there is an oracle machine that computes the characteristic function of *A* when run with oracle *B*. In this case, we also say *A* is ***B*-recursive** and ***B*-computable**.

If there is an oracle machine that, when run with oracle *B*, computes a partial function with domain *A*, then *A* is said to be ***B*-recursively enumerable** and ***B*-computably enumerable**.

We say A is **Turing equivalent** to B and write $A \equiv_T B$ if both $A \leq_T B$ and $B \leq_T A$. The equivalence classes of Turing equivalent sets are called **Turing degrees**. The Turing degree of a set X is written $\mathbf{deg}(X)$.

Given a set $\mathcal{X} \subseteq \mathcal{P}(\mathbb{N})$, a set $A \subseteq \mathbb{N}$ is called **Turing hard** for $\mathcal{X}$ if $X \leq_T A$ for all $X \in \mathcal{X}$. If additionally $A \in \mathcal{X}$ then A is called **Turing complete** for $\mathcal{X}$.

Relation of Turing completeness to computational universality

Turing completeness, as just defined above, corresponds only partially to Turing completeness in the sense of computational universality. Specifically, a Turing machine is a universal Turing machine iff its halting problem (i.e., the set of inputs for which it eventually halts) is many-one complete. Thus, a necessary *but insufficient* condition for a machine to be computationally universal, is that the machine's halting problem be Turing-complete for the set $\mathcal{X}$ of recursively enumerable sets.

Example

Let W_e denote the set of input values for which the Turing machine with index e halts. Then the sets $A = \{e \mid e \in W_e\}$ and $B = \{(e,n) \mid n \in W_e\}$ are Turing equivalent (here (e,n) denotes an effective pairing function). A reduction showing $A \leq_T B$ can be constructed using the fact that $e \in A \Leftrightarrow (e,e) \in B$. Given a pair (e,n), a new index $i(e,n)$ can be constructed using the s_{mn} theorem such that the program coded by $i(e,n)$ ignores its input and merely simulates the computation of the machine with index e on input n. In particular, the machine with index $i(e,n)$ either halts on every input or halts on no input. Thus $i(e,n) \in A \Leftrightarrow (e,n) \in B$ holds for all e and n. Because the function i is computable, this shows $B \leq_T A$. The reductions presented here are not only Turing reductions but *many-one reductions*, discussed below.

Properties

- Every set is Turing equivalent to its complement
- Every computable set is Turing reducible to every other computable set. Because these sets can be computed with no oracle, they can be computed by an oracle machine that ignores the oracle it is given.
- The relation $\leq_T$ is transitive: if $A \leq_T B$ and $B \leq_T C$ then $A \leq_T C$. Moreover $A \leq A$ holds for every set A, and thus the relation $\leq_T$ is a preorder (it is not a partial order because $A \leq_T B$ and $B \leq_T A$ does not necessarily imply $A = B$).
- There are pairs of sets (A,B) such that A is not Turing reducible to B and B is not Turing reducible to A. Thus $\leq_T$ is not a linear order.
- There are infinite decreasing sequences of sets under $\leq_T$. Thus this relation is not well-founded.

- Every set is Turing reducible to its own Turing jump, but the Turing jump of a set is never Turing reducible to the original set.

The use of a reduction

Since every reduction from a set B to a set A has to determine whether a single element is in A in only finitely many steps, it can only make finitely many queries of membership in the set B. When the amount of information about the set B used to compute a single bit of A is discussed, this is made precise by the use function. Formally, the *use* of a reduction is the function that sends each natural number n to the largest natural number m whose membership in the set B was queried by the reduction while determining the membership of n in A.

Turing degree

In computer science and mathematical logic the **Turing degree** or **degree of unsolvability** of a set of natural numbers measures the level of algorithmic unsolvability of the set. The concept of Turing degree is fundamental in computability theory, where sets of natural numbers are often regarded as decision problems; the Turing degree of a set tells how difficult it is to solve the decision problem associated with the set.

Two sets are **Turing equivalent** if they have the same level of unsolvability; each Turing degree is a collection of Turing equivalent sets, so that two sets are in different Turing degrees exactly when they are not Turing equivalent. Furthermore, the Turing degrees are partially ordered so that if the Turing degree of a set X is less than the Turing degree of a set Y then any (noncomputable) procedure that correctly decides whether numbers are in Y can be effectively converted to a procedure that correctly decides whether numbers are in X. It is in this sense that the Turing degree of a set corresponds to its level of algorithmic unsolvability.

The Turing degrees were introduced by Emil Leon Post (1944), and many fundamental results were established by Stephen Cole Kleene and Post (1954). The Turing degrees have been an area of intense research since then. Many proofs in the area make use of a proof technique known as the **priority method**.

Turing equivalence

Rest here, the word *set* will refer to a set of natural numbers. A set X is said to be **Turing reducible** to a set Y if there is an oracle Turing machine that decides membership in X when given an oracle for membership in Y. The notation $X \leq_T Y$ indicates that X is Turing reducible to Y.

Two sets X and Y are defined to be **Turing equivalent** if X is Turing reducible to Y and Y is Turing reducible to X. The notation $X \equiv_T Y$ indicates that X and Y are Turing equivalent. The relation $\equiv_T$ can be seen to be an equivalence relation, which means that for all sets X, Y, and Z:

- $X \equiv_T X$
- $X \equiv_T Y$ implies $Y \equiv_T X$
- If $X \equiv_T Y$ and $Y \equiv_T Z$ then $X \equiv_T Z$.

Turing degree

A **Turing degree** is an equivalence class of the relation $\equiv_T$. The notation $[X]$ denotes the equivalence class containing a set X. The entire collection of Turing degrees is denoted $\mathcal{D}$.

The Turing degrees have a partial order $\leq$ defined so that $[X] \leq [Y]$ if and only if $X \leq_T Y$. There is a unique Turing degree containing all the computable sets, and this degree is less than every other degree. It is denoted **0** (zero) because it is the least element of the poset $\mathcal{D}$. (It is common to use boldface notation for Turing degrees, in order to distinguish them from sets. When no confusion can occur, such as with $[X]$, the boldface is not necessary.)

For any sets X and Y, X **join** Y, written, is defined to be the union of the sets: and.. The Turing degree of least upper bound of the degrees of X and Y. Thus $\mathcal{D}$is a join-semilattice. The least upper bound of degrees **a** and **b** is denoted. It is known that $\mathcal{D}$is not a lattice, as there are pairs of degrees with no greatest lower bound.

For any set X the notation X' denotes the set of indices of oracle machines that halt when using X as an oracle. The set X' is called the **Turing jump** of X. The Turing jump of a degree $[X]$ is defined to be the degree $[X']$; this is a valid definition because $X' \equiv_T Y'$ whenever $X \equiv_T Y$. A key example is $\mathbf{0}'$, the degree of the halting problem.

Basic properties of the Turing degrees

- Every Turing degree is countably infinite, that is, it contains exactly $\aleph_0$sets.
- There are $2^{\aleph_0}$distinct Turing degrees.
- For each degree **a** the strict inequality $\mathbf{a} < \mathbf{a}'$ holds.
- For each degree **a**, the set of degrees below **a** is at most countable. The set of degrees greater than **a** has size $2^{\aleph_0}$.

Structure of the Turing degrees

A great deal of research has been conducted into the structure of the Turing degrees. The following survey lists only some of the many known results. One general conclusion that can be drawn from the research is that the structure of the Turing degrees is extremely complicated.

Order properties

- There are **minimal degrees**. A degree **a** is *minimal* if **a** is nonzero and there is no degree between **0** and **a**. Thus the order relation on the degrees is not a dense order.
- For every nonzero degree **a** there is a degree **b** incomparable with **a**.
- There is a set of $2^{\aleph_0}$ pairwise incomparable Turing degrees.
- There are pairs of degrees with no greatest lower bound. Thus $\mathcal{D}$ is not a lattice.
- Every countable partially ordered set can be embedded in the Turing degrees.
- No infinite, strictly increasing sequence of degrees has a least upper bound.

Properties involving the jump

- For every degree **a** there is a degree strictly between **a** and **a′**. In fact, there is a countable sequence of pairwise incomparable degrees between **a** and **a′**.
- A degree **a** is of the form **b′** if and only if **0′** ≤ **a**.
- For any degree **a** there is a degree **b** such that **a** < **b** and **b′** = **a′**; such a degree **b** is called *low* relative to **a**.
- There is an infinite sequence $\mathbf{a}_i$ of degrees such that $\mathbf{a}'_{i+1} \leq \mathbf{a}_i$ for each *i*.

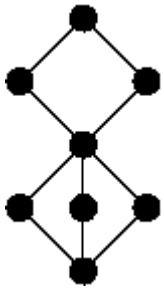
Logical properties

- Simpson (1977) showed that the first-order theory of $\mathcal{D}$ in the language or many-one equivalent to the theory of true second-order arithmetic. This indicates that the structure of $\mathcal{D}$ is extremely complicated.
- Shore and Slaman (1999) showed that the jump operator is definable in the first-order structure of the degrees with the language

Structure of the r.e. Turing degrees

A degree is called r.e. (recursively enumerable) if it contains a recursively enumerable set. Every r.e. degree is less than or equal to $\mathbf{0'}$ but not every degree less than $\mathbf{0'}$ is an r.e. degree.

- (G. E. Sacks, 1964) The r.e degrees are dense; between any two r.e. degrees there is a third r.e degree.
- (A. H. Lachlan, 1966a and C. E. M. Yates, 1966) There are two r.e. degrees with no greatest lower bound in the r.e. degrees.
- (A. H. Lachlan, 1966a and C. E. M. Yates, 1966) There is a pair of nonzero r.e. degrees whose greatest lower bound is **0**.
- (S. K. Thomason, 1971) Every finite distributive lattice can be embedded into the r.e. degrees. In fact, the countable atomless Boolean algebra can be embedded in a manner that preserves suprema and infima.
- (A. H. Lachlan and R. I. Soare, 1980) Not all finite lattices can be embedded in the r.e. degrees (via an embedding that preserves suprema and infima). The following particular lattice cannot be embedded in the r.e. degrees:



- (A. H. Lachlan, 1966b) There is no pair of r.e. degrees whose greatest lower bound is **0** and whose least upper bound is $\mathbf{0'}$. This result is informally called the *nondiamond theorem*.
- The first-order theory of the r.e. degrees in the language is many-one equivalent to the theory of true first order arithmetic.

Post's problem and the priority method

Emil Post studied the r.e. Turing degrees and asked whether there is any r.e. degree strictly between **0** and $\mathbf{0'}$. The problem of constructing such a degree (or showing that none exist) became known as **Post's problem**. This problem was solved independently by Friedberg and Muchnik in the 1950s, who showed that these intermediate r.e. degrees do exist. Their proofs each developed the same new method for constructing r.e degrees

which came to be known as the **priority method**. The priority method is now the main technique for establishing results about r.e. sets.

The idea of the priority method for constructing an r.e. set X is to list a countable sequence of *requirements* that X must satisfy. For example, to construct an r.e. set X between **0** and **0′** it is enough to satisfy the requirements A_e and B_e for each natural number e, where A_e requires that the oracle machine with index e does not compute $0'$ from X and B_e requires that the Turing machine with index e (and no oracle) does not compute X. These requirements are put into a *priority ordering*, which is an explicit bijection of the requirements and the natural numbers. The proof proceeds inductively with one stage for each natural number; these stages can be thought of as steps of time during which the set X is enumerated. At each stage, numbers may put into X or forever prevented from entering X in an attempt to *satisfy* requirements (that is, force them to hold once all of X has been enumerated). Sometimes, a number can be enumerated into X to satisfy one requirement but doing this would cause a previously satisfied requirement to become unsatisfied (that is, to be *injured*). The priority order on requirements is used to determine which requirement to satisfy in this case. The informal idea is that if a requirement is injured then it will eventually stop being injured after all higher priority requirements have stopped being injured, although not every priority argument has this property. An argument must be made that the overall set X is r.e. and satisfies all the requirements. Priority arguments can be used to prove many facts about r.e. sets; the requirements used and the manner in which they are satisfied must be carefully chosen to produce the required result.

Chapter 3

Reduction (recursion theory)

In computability theory, many **reducibility relations** (also called **reductions**, **reducibilities**, and **notions of reducibility**) are studied. They are motivated by the question: given sets A and B of natural numbers, is it possible to effectively convert a method for deciding membership in B into a method for deciding membership in A? If the answer to this question is affirmative then A is said to be **reducible to** B.

The study of reducibility notions is motivated by the study of decision problems. For many notions of reducibility, if any noncomputable set is reducible to a set A then A must also be noncomputable. This gives a powerful technique for proving that many sets are noncomputable.

Reducibility relations

A **reducibility relation** is a binary relation on sets of natural numbers that is

- Reflexive: Every set is reducible to itself.
- Transitive: If a set A is reducible to a set B and B is reducible to a set C then A is reducible to C.

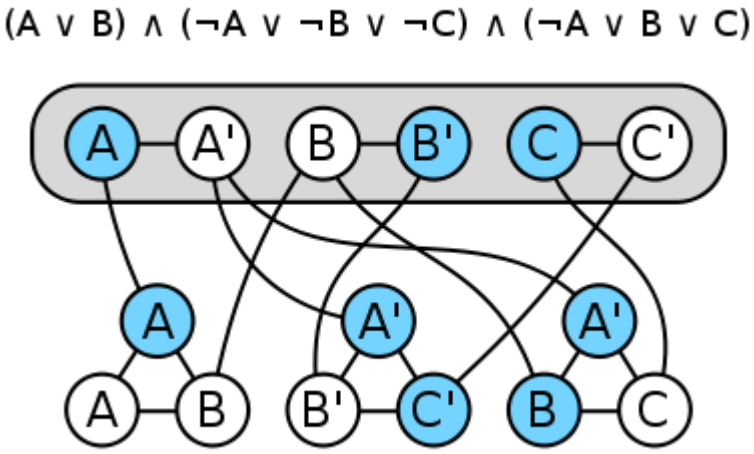
These two properties imply that a reducibility is a preorder on the powerset of the natural numbers. Not all preorders are studied as reducibility notions, however. The notions studied in computability theory have the informal property that A is reducible to B if and only if any (possibly noneffective) decision procedure for B can be effectively converted to a decision procedure for A. The different reducibility relations vary in the methods they permit such a conversion process to use.

Degrees of a reducibility relation

Every reducibility relation (in fact, every preorder) induces an equivalence relation on the powerset of the natural numbers in which two sets are equivalent if and only if each one is reducible to the other. In recursion theory, these equivalence classes are called the **degrees** of the reducibility relation. For example, the Turing degrees are the equivalence classes of sets of naturals induced by Turing reducibility.

The degrees of any reducibility relation are partially ordered by the relation in the following manner. Let $\leq$ be a reducibility relation and let **A** and **B** be two of its degrees. Then $\mathbf{A} \leq \mathbf{B}$ if and only if there is a set A in **A** and a set B in **B** such that $A \leq B$. This is equivalent to the property that for every set A in **A** and every set B in **B**, $A \leq B$, because any two sets in A are equivalent and any two sets in B are equivalent. It is common, as shown here, to use boldface notation to denote degrees.

Reduction (complexity)



Example of a reduction from a boolean satisfiability problem to a vertex cover problem. Blue vertices form a vertex cover which corresponds to truth values.

In computability theory and computational complexity theory, a **reduction** is a transformation of one problem into another problem. Depending on the transformation used this can be used to define complexity classes on a set of problems.

Intuitively, problem A is reducible to problem B if solutions to B exist and give solutions to A whenever A has solutions. Thus, solving A cannot be harder than solving B. We write $A \leq_m B$, usually with a subscript on the $\leq$ to indicate the type of reduction being used (m : mapping reduction,p : polynomial reduction).

Introduction

Often we find ourselves trying to solve a problem that is similar to a problem we've already solved. In these cases, often a quick way of solving the new problem is to transform each instance of the new problem into instances of the old problem, solve these using our existing solution, and then use these to obtain our final solution. This is perhaps the most obvious use of reductions.

Another, more subtle use is this: suppose we have a problem that we've proven is hard to solve, and we have a similar new problem. We might suspect that it, too, is hard to solve.

We argue by contradiction: suppose the new problem is easy to solve. Then, if we can show that every instance of the old problem can be solved easily by transforming it into instances of the new problem and solving those, we have a contradiction. This establishes that the new problem is also hard.

A very simple example of a reduction is from *multiplication* to *squaring*. Suppose all we know how to do is to add, subtract, take squares, and divide by two. We can use this knowledge, combined with the following formula, to obtain the product of any two numbers:

$$a \times b = \frac{\left((a+b)^2 - a^2 - b^2\right)}{2}$$

We also have a reduction in the other direction; obviously, if we can multiply two numbers, we can square a number. This seems to imply that these two problems are equally hard. This kind of reduction corresponds to Turing reduction.

However, the reduction becomes much harder if we add the restriction that we can only use the squaring function one time, and only at the end. In this case, even if we're allowed to use all the basic arithmetic operations, including multiplication, no reduction exists in general, because we may have to compute an irrational number like $\sqrt{2}$from rational numbers. Going in the other direction, however, we can certainly square a number with just one multiplication, only at the end. Using this limited form of reduction, we have shown the unsurprising result that multiplication is harder in general than squaring. This corresponds to many-one reduction.

Definition

Given two subsets *A* and *B* of **N** and a set of functions *F* from **N** to **N** which is closed under composition, *A* is called **reducible** to *B* under *F* if

$$\exists f \in F \ . \ \forall x \in \mathbb{N} \ . \ x \in A \Leftrightarrow f(x) \in B$$

We write

$$A \leq_F B$$

Let *S* be a subset of **P**(**N**) and ≤ a reduction, then *S* is called **closed** under ≤ if

$$\forall s \in S \ . \ \forall A \in P(\mathbb{N}) \ . \ A \leq s \Rightarrow A \in S$$

A subset *A* of **N** is called **hard** for *S* if

$$\forall s \in S \ . \ s \leq A$$

A subset A of **N** is called **complete** for S if A is hard for S and A is in S.

Properties

A reduction is a preordering, that is a reflexive and transitive relation, on **P**(**N**)×**P**(**N**), where **P**(**N**) is the power set of the natural numbers.

Types and applications of reductions

As described in the example above, there are two main types of reductions used in computational complexity, the many-one reduction and the Turing reduction. Many-one reductions map *instances* of one problem to *instances* of another; Turing reductions *compute* the solution to one problem, assuming the other problem is easy to solve. A many-one reduction is weaker than a Turing reduction. Weaker reductions are more effective at separating problems, but they have less power, making reductions harder to design.

A problem is complete for a complexity class if every problem in the class reduces to that problem, and it is also in the class itself. In this sense the problem represents the class, since any solution to it can, in combination with the reductions, be used to solve every problem in the class.

However, in order to be useful, reductions must be *easy*. For example, it's quite possible to reduce a difficult-to-solve NP-complete problem like the boolean satisfiability problem to a trivial problem, like determining if a number equals zero, by having the reduction machine solve the problem in exponential time and output zero only if there is a solution. However, this does not achieve much, because even though we can solve the new problem, performing the reduction is just as hard as solving the old problem. Likewise, a reduction computing a noncomputable function can reduce an undecidable problem to a decidable one. As Michael Sipser points out in *Introduction to the Theory of Computation*: "The reduction must be easy, relative to the complexity of typical problems in the class [...] If the reduction itself were difficult to compute, an easy solution to the complete problem wouldn't necessarily yield an easy solution to the problems reducing to it."

Therefore, the appropriate notion of reduction depends on the complexity class being studied. When studying the complexity class NP and harder classes such as the polynomial hierarchy, polynomial-time reductions are used. When studying classes within P such as NC and NL, log-space reductions are used. Reductions are also used in computability theory to show whether problems are or are not solvable by machines at all; in this case, reductions are restricted only to computable functions.

In case of optimization (maximization or minimization) problems, we often think in terms of approximation-preserving reductions. Suppose we have two optimization problems such that instances of one problem can be mapped onto instances of the other, in a way that nearly-optimal solutions to instances of the latter problem can be

transformed back to yield nearly-optimal solutions to the former. This way, if we have an optimization algorithm (or approximation algorithm) that finds near-optimal (or optimal) solutions to instances of problem B, and an efficient approximation-preserving reduction from problem A to problem B, by composition we obtain an optimization algorithm that yields near-optimal solutions to instances of problem A. Approximation-preserving reductions are often used to prove hardness of approximation results: if some optimization problem A is hard to approximate (under some complexity assumption) within a factor better than α for some α, and there is a β-approximation-preserving reduction from problem A to problem B, we can conclude that problem B is hard to approximate within factor α/β.

Reductions stronger than Turing reducibility

The strong reducibilities include

1. Many-one reduction

In computability theory and computational complexity theory, a **many-one reduction** is a reduction which converts instances of one decision problem into instances of a second decision problem. Reductions are thus used to measure the relative computational difficulty of two problems.

Many-one reductions are a special case and a stronger form of Turing reductions. With many-one reductions the oracle can be invoked only once at the end and the answer cannot be modified.

Many-one reductions were first used by Emil Post in 1944. Later Norman Shapiro used the same concept in 1956 under the name *strong reducibility*.

Definitions

Formal languages

Suppose A and B are formal languages over the alphabets Σ and Γ, respectively. A **many-one reduction** from A to B is a total computable function $f : \Sigma^* \to \Gamma^*$ that has the property that each word w is in A if and only if $f(w)$ is in B (that is, $A = f^{-1}(B)$).

If such a function f exists, we say that A is **many-one reducible** or **m-reducible** to B and write

$$A \leq_m B.$$

If there is an injective many-one reduction function then we say A is **1 reducible** or **one-one reducible** to B and write

$$A \leq_1 B.$$

Subsets of natural numbers

Given two sets $A, B \subseteq \mathbb{N}$ we say A is **many-one reducible** to B and write

$$A \leq_m B$$

if there exists a total computable function f with $A = f^{-1}(B)$. If additionally f is injective we say A is **1-reducible** to B and write

$$A \leq_1 B.$$

Many-one equivalence and 1 equivalence

If $A \leq_m B$ and $B \leq_m A$ we say A is **many-one equivalent** or **m-equivalent** to B and write

$$A \equiv_m B.$$

If $A \leq_1 B$ and $B \leq_1 A$ we say A is **1-equivalent** to B and write

$$A \equiv_1 B.$$

Many-one completeness (m-completeness)

A set B is called *many-one complete*, or simply **m-complete**, iff B is recursively enumerable and every recursively enumerable set A is m-reducible to B.

Many-one reductions with resource limitations

Many-one reductions are often subjected to resource restrictions, for example that the reduction function is computable in polynomial time or logarithmic space.

Given decision problems A and B and an algorithm N which solves instances of B, we can use a many-one reduction from A to B to solve instances of A in:

- the time needed for N plus the time needed for the reduction
- the maximum of the space needed for N and the space needed for the reduction

We say that a class **C** of languages (or a subset of the power set of the natural numbers) is *closed under many-one reducibility* if there exists no reduction from a language in **C** to a language outside **C**. If a class is closed under many-one reducibility, then many-one reduction can be used to show that a problem is in **C** by reducing a problem in **C** to it. Many-one reductions are valuable because most well-studied complexity classes are closed under some type of many-one reducibility, including P, NP, L, NL, co-NP,

PSPACE, EXP, and many others. These classes are not closed under arbitrary many-one reductions, however.

2. Truth-table reduction

In computability theory, a **truth-table reduction** is a reduction from one set of natural numbers to another. As a "tool", it is weaker than Turing reduction, since not every Turing reduction between sets can be performed by a truth-table reduction, but every truth-table reduction can be performed by a Turing reduction. For the same reason it is said to be a stronger reducibility than Turing reducibility, because it implies Turing reducibility. A **weak truth-table reduction** is a related type of reduction which is so named because it weakens the constraints placed on a truth-table reduction, and provides a weaker equivalence classification; as such, a "weak truth-table reduction" can actually be more powerful than a truth-table reduction as a "tool", and perform a reduction which is not performable by truth table.

A Turing reduction from a set B to a set A computes the membership of a single element in A by asking questions about the membership of various elements in B during the computation; it may adaptively determine which questions it asks based upon answers to previous questions. In contrast, a truth-table reduction or a weak truth-table reduction must present all of its (finitely many) oracle queries at the same time. In a truth-table reduction, the reduction also gives a boolean function (a truth table) which, when given the answers to the queries, will produce the final answer of the reduction. In a weak truth-table reduction, the reduction uses the oracle answers as a basis for further computation which may depend on the given answers but may not ask further questions of the oracle.

Equivalently, a weak truth-table reduction is a Turing reduction for which the use of the reduction is bounded by a computable function. For this reason, they are sometimes referred to as **bounded Turing** (bT) reductions rather than as weak truth-table (wtt) reductions.

Properties

As every truth-table reduction is a Turing reduction, if A is truth-table reducible to B ($A \leq_{tt} B$), then A is also Turing reducible to B ($A \leq_T B$). Considering also one-one reducibility, many-one reducibility and weak truth-table reducibility, one gets the following chain of implications:

- $A \leq_1 B \Rightarrow A \leq_m B \Rightarrow A \leq_{tt} B \Rightarrow A \leq_{wtt} B \Rightarrow A \leq_T B$; one-one reducibility implies many-one reducibility, which implies truth-table reducibility, which in turn implies weak truth-table reducibility, which in turn implies Turing reducibility.

Bounded reducibilities

A **bounded** form of each of the above strong reducibilities can be defined. The most famous of these is bounded truth-table reduction, but there are also bounded Turing, bounded weak truth-table and others. These first three are the most common ones and they are based on the number of queries. For example, a set A is bounded truth-table reducible to B if and only if the Turing machine M computing A relative to B computes a list of up to n numbers, queries B on these numbers and then terminates for all possible oracle answers; the value n is a constant independent of x. The difference between bounded weak truth-table and bounded Turing reduction is that in the first case, the up to n queries have to be made at the same time while in the second case, the queries can be made one after the other. For that reason, there are cases where A is bounded Turing reducible to B but not weak truth-table reducible to B.

Strong reductions in computational complexity

The strong reductions listed above restrict the manner in which oracle information can be accessed by a decision procedure but do not otherwise limit the computational resources available. Thus if a set A is decidable then A is reducible to any set B under any of the strong reducibility relations listed above, even if A is not polynomial-time or exponential-time decidable. This is acceptable in the study of recursion theory, which is interested in theoretical computability, but it is not reasonable for computational complexity theory, which studies which sets can be decided under certain asymptotical resource bounds.

The most common reducibility in computational complexity theory is polynomial-time reducibility; a set A is polynomial-time reducible to a set B if there is a polynomial-time function f such that for every n, n is in A if and only if $f(n)$ is in B. This reducibility is, essentially, a resource-bounded version of many-one reducibility. Other resource-bounded reducibilities are used in other contexts of computational complexity theory where other resource bounds are of interest.

Reductions weaker than Turing reducibility

Although Turing reducibility is the most general reducibility that is effective, weaker reducibility relations are commonly studied. These reducibilities are related to relative definability of sets over arithmetic or set theory. They include:

- Arithmetical reducibility: A set A is arithmetical in a set B if A is definable over the standard model of Peano arithmetic with an extra predicate for B. Equivalently, according to Post's theorem, A is arithmetical in B if and only if A is Turing reducible to $B^{(n)}$, the nth Turing jump of B, for some natural number n. The arithmetical hierarchy gives a finer classification of arithmetical reducibility.
- Hyperarithmetical reducibility: A set A is hyperarithmetical in a set B if A is Δ^1_1definable over the standard model of Peano arithmetic with a predicate for B. Equivalently, A is hyperarithmetical in B if and only if A is Turing reducible to $B^{(\alpha)}$, the αth Turing jump of B, for some B-recursive ordinal α.

- Relative constructibility: A set A is relatively constructible from a set B if A is in $L(B)$, the smallest transitive model of ZFC set theory containing B and all the ordinals.

Chapter 4

Reverse Mathematics and Kolmogorov Complexity

Reverse mathematics

Reverse mathematics is a program in mathematical logic that seeks to determine which axioms are required to prove theorems of mathematics. Its defining method can briefly be described as "going backwards from the theorems to the axioms", in contrast to the ordinary mathematical practice of deriving theorems from axioms. The reverse mathematics program was foreshadowed by results in set theory such as the classical theorem that the axiom of choice and Zorn's lemma are equivalent over ZF set theory. The goal of reverse mathematics, however, is to study ordinary theorems of mathematics rather than possible axioms for set theory.

Reverse mathematics is usually carried out using subsystems of second-order arithmetic, where many of its definitions and methods are inspired by previous work in constructive analysis and proof theory. The use of second-order arithmetic also allows many techniques from recursion theory to be employed; many results in reverse mathematics have corresponding results in computable analysis.

General principles

In reverse mathematics, one starts with a framework language and a base theory—a core axiom system—that is too weak to prove most of the theorems one might be interested in, but still powerful enough to develop the definitions necessary to state these theorems. For example, to study the theorem “Every bounded sequence of real numbers has a supremum” it is necessary to use a base system which can speak of real numbers and sequences of real numbers.

For each theorem that can be stated in the base system but is not provable in the base system, the goal is to determine the particular axiom system (stronger than the base system) that is necessary to prove that theorem. To show that a system S is required to prove a theorem T, two proofs are required. The first proof shows T is provable from S; this is an ordinary mathematical proof along with a justification that it can be carried out in the system S. The second proof, known as a **reversal**, shows that T itself implies S; this

proof is carried out in the base system. The reversal establishes that no axiom system S' that extends the base system can be weaker than S while still proving T.

Use of second-order arithmetic

Most reverse mathematics research focuses on subsystems of second-order arithmetic. The body of research in reverse mathematics has established that weak subsystems of second-order arithmetic suffice to formalize almost all undergraduate-level mathematics. In second-order arithmetic, all objects must be represented as either natural numbers or sets of natural numbers. For example, in order to prove theorems about real numbers, the real numbers must be represented as Cauchy sequences of rational numbers, each of which can be represented as a set of natural numbers.

The axiom systems most often considered in reverse mathematics are defined using axiom schemes called **comprehension schemes**. Such a scheme states that any set of natural numbers definable by a formula of a given complexity exists. In this context, the complexity of formulas is measured using the arithmetical hierarchy and analytical hierarchy.

The reason that reverse mathematics is not carried out using set theory as a base system is that the language of set theory is too expressive. Extremely complex sets of natural numbers can be defined by simple formulas in the language of set theory (which can quantify over arbitrary sets). In the context of second-order arithmetic, results such as Post's theorem establish a close link between the complexity of a formula and the (non)computability of the set it defines.

Another effect of using second-order arithmetic is the need to restrict general mathematical theorems to forms that can be expressed within arithmetic. For example, second-order arithmetic can express the principle "Every countable vector space has a basis" but it cannot express the principle "Every vector space has a basis". In practical terms, this means that theorems of algebra and combinatorics are restricted to countable structures, while theorems of analysis and topology are restricted to separable spaces. Many principles that imply the axiom of choice in their general form (such as "Every vector space has a basis") become provable in weak subsystems of second-order arithmetic when they are restricted. For example, "every field has an algebraic closure" is not provable in ZF set theory, but the restricted form "every countable field has an algebraic closure" is provable in RCA_0, the weakest system typically employed in reverse mathematics.

The big five subsystems of second order arithmetic

Second order arithmetic is a formal theory of the natural numbers and sets of natural numbers. Many mathematical objects, such as countable rings, groups, and fields, as well as points in effective Polish spaces, can be represented as sets of natural numbers, and modulo this representation can be studied in second order arithmetic.

Reverse mathematics makes use of several subsystems of second order arithmetic. A typical reverse mathematics theorem shows that a particular mathematical theorem *T* is equivalent to a particular subsystem *S* of second order arithmetic over a weaker subsystem *B*. This weaker system *B* is known as the **base system** for the result; in order for the reverse mathematics result to have meaning, this system must not itself be able to prove the mathematical theorem *T*.

Simpson (2009) describes five particular subsystems of second order arithmetic, which he calls the **Big Five**, that occur frequently in reverse mathematics. In order of increasing strength, these systems are named by the initialisms RCA_0, WKL_0, ACA_0, ATR_0, and Π^1_1-CA_0.

The following table summarizes the "big five" systems Simpson (2009, p.42)

Subsystem	Stands for	Ordinal	Corresponds roughly to	Comments
RCA_0	Recursive comprehension axiom	ω^ω	Constructive mathematics (Bishop)	The base system for reverse mathematics
WKL_0	Weak König's lemma	ω^ω	Finitistic reductionism (Hilbert)	Conservative over PRA for Π^0_2 sentences. Conservative over RCA_0 for Π^1_1 sentences.
ACA_0	Arithmetical comprehension axiom	ε_0	Predicativism (Weyl, Feferman)	Conservative over Peano arithmetic for arithmetical sentences
ATR_0	Arithmetical transfinite recursion	Γ_0	Predicative reductionism (Friedman, Simpson)	Conservative over Feferman's system IR for Π^1_1 sentences
Π^1_1-CA_0	Π^1_1 comprehension axiom	$\Psi_0(\Omega_\omega)$	Impredicativism	

The subscript $_0$ in these names means that the induction scheme has been restricted from the full second-order induction scheme (Simpson 2009, p. 6). For example, ACA_0 includes the induction axiom $(0\in X \wedge \forall n(n\in X \rightarrow n+1\in X)) \rightarrow \forall n\, n\in X$. This together with the full comprehension axiom of second order arithmetic implies the full second-order induction scheme given by the universal closure of $(\varphi(0) \wedge \forall n(\varphi(n) \rightarrow \varphi(n+1))) \rightarrow \forall n\, \varphi(n)$ for any second order formula φ. However ACA_0 does not have the full comprehension axiom, and the subscript $_0$ is a reminder that it does not have the full second-order induction scheme either. This restriction is important: systems with restricted induction have significantly lower proof-theoretical ordinals than systems with the full second-order induction scheme.

The base system RCA_0

RCA_0 is the fragment of second-order arithmetic whose axioms are the axioms of Robinson arithmetic, induction for Σ_1^0 formulas, and comprehension for Δ_1^0 formulas.

The subsystem RCA_0 is the one most commonly used as a base system for reverse mathematics. The initials "RCA" stand for "recursive comprehension axiom", where "recursive" means "computable", as in recursive function. This name is used because RCA_0 corresponds informally to "computable mathematics". In particular, any set of natural numbers that can be proven to exist in RCA_0 is computable, and thus any theorem which implies that noncomputable sets exist is not provable in RCA_0 To this extent, RCA_0 is a constructive system, although it does not meet the requirements of the program of constructivism because it is a theory in classical logic including the excluded middle.

Despite its seeming weakness (of not proving any noncomputable sets exist), RCA_0 is sufficient to prove a number of classical theorems which, therefore, require only minimal logical strength. These theorems are, in a sense, below the reach of the reverse mathematics enterprise because they are already provable in the base system. The classical theorems provable in RCA_0 include:

- Basic properties of the natural numbers, integers, and rational numbers (for example, that the latter form an ordered field).
- Basic properties of the real numbers (the real numbers are an Archimedean ordered field; any nested sequence of closed intervals whose lengths tend to zero has a single point in its intersection; the real numbers are not countable).
- The Baire category theorem for a complete separable metric space (the separability condition is necessary to even state the theorem in the language of second-order arithmetic).
- The intermediate value theorem on continuous real functions.
- The Banach–Steinhaus theorem for a sequence of continuous linear operators on separable Banach spaces.
- A weak version of Gödel's completeness theorem (for a set of sentences, in a countable language, that is already closed under consequence).
- The existence of an algebraic closure for a countable field (but not its uniqueness).
- The existence and uniqueness of the real closure of a countable ordered field.

The first-order part of RCA_0 (the theorems of the system that do not involve any set variables) is the set of theorems of first-order Peano arithmetic with induction limited to Σ^0_1 formulas. It is provably consistent, as is RCA_0, in full first-order Peano arithmetic.

Weak König's lemma WKL_0

The subsystem WKL_0 consists of RCA_0 plus a weak form of König's lemma, namely the statement that every infinite subtree of the full binary tree (the tree of all finite sequences of 0's and 1's) has an infinite path. This proposition, which is known as *weak König's*

lemma, is easy to state in the language of second-order arithmetic. WKL_0 can also be defined as the principle of Σ^0_1 separation (given two Σ^0_1 formulas of a free variable *n* which are exclusive, there is a class containing all *n* satisfying the one and no *n* satisfying the other).

The following remark on terminology is in order. The term "weak König's lemma" refers to the sentence which says that any infinite subtree of the binary tree has an infinite path. When this axiom is added to RCA_0, the resulting subsystem is called WKL_0 A similar distinction between particular axioms, on the one hand, and subsystems including the basic axioms and induction, on the other hand, is made for the stronger subsystems described below.

In a sense, weak König's lemma is a form of the axiom of choice (although, as stated, it can be proven in classical Zermelo–Fraenkel set theory without the axiom of choice). It is not constructively valid in some senses of the word constructive.

To show that WKL_0 is actually stronger than (not provable in) RCA_0, it is sufficient to exhibit a theorem of WKL_0 which implies that noncomputable sets exist. This is not difficult; WKL_0 implies the existence of separating sets for effectively inseparable recursively enumerable sets.

It turns out that RCA_0 and WKL_0 have the same first-order part, meaning that they prove the same first-order sentences. WKL_0 can prove a good number of classical mathematical results which do not follow from RCA_0, however. These results are not expressible as first order statements but can be expressed as second-order statements.

The following results are equivalent to weak König's lemma and thus to WKL_0 over RCA_0:

- The Heine–Borel theorem for the closed unit real interval, in the following sense: every covering by a sequence of open intervals has a finite subcovering.
- The Heine–Borel theorem for complete totally bounded separable metric spaces (where covering is by a sequence of open balls).
- A continuous real function on the closed unit interval (or on any compact separable metric space, as above) is bounded (or: bounded and reaches its bounds).
- A continuous real function on the closed unit interval can be uniformly approximated by polynomials (with rational coefficients).
- A continuous real function on the closed unit interval is uniformly continuous.
- A continuous real function on the closed unit interval is Riemann integrable.
- The Brouwer fixed point theorem (for continuous functions on a finite product of copies of the closed unit interval).
- The separable Hahn–Banach theorem in the form: a bounded linear form on a subspace of a separable Banach space extends to a bounded linear form on the whole space.
- The Jordan curve theorem

- Gödel's completeness theorem (for a countable language).
- Every countable commutative ring has a prime ideal.
- Every countable formally real field is orderable.
- Uniqueness of algebraic closure (for a countable field).

Arithmetical comprehension ACA_0

ACA_0 is RCA_0 plus the comprehension scheme for arithmetical formulas (which is sometimes called the "arithmetical comprehension axiom"). That is, ACA_0 allows us to form the set of natural numbers satisfying an arbitrary arithmetical formula (one with no bound set variables, although possibly containing set parameters). Actually, it suffices to add to RCA_0 the comprehension scheme for Σ_1 formulas in order to obtain full arithmetical comprehension.

The first-order part of ACA_0 is exactly first-order Peano arithmetic; ACA_0 is a *conservative* extension of first-order Peano arithmetic. The two systems are provably (in a weak system) equiconsistent. ACA_0 can be thought of as a framework of predicative mathematics, although there are predicatively provable theorems that are not provable in ACA_0. Most of the fundamental results about the natural numbers, and many other mathematical theorems, can be proven in this system.

One way of seeing that ACA_0 is stronger than WKL_0 is to exhibit a model of WKL_0 that doesn't contain all arithmetical sets. In fact, it is possible to build a model of WKL_0 consisting entirely of low sets using the low basis theorem, since low sets relative to low sets are low.

The following assertions are equivalent to ACA_0 over RCA_0:

- The sequential completeness of the real numbers (every bounded increasing sequence of real numbers has a limit).
- The Bolzano–Weierstrass theorem.
- Ascoli's theorem: every bounded equicontinuous sequence of real functions on the unit interval has a uniformly convergent subsequence.
- Every countable commutative ring has a maximal ideal.
- Every countable vector space over the rationals (or over any countable field) has a basis.
- Every countable field has a transcendence basis.
- König's lemma (for arbitrary finitely branching trees, as opposed to the weak version described above).
- Various theorems in combinatorics, such as certain forms of Ramsey's theorem.

Arithmetical Transfinite Recursion ATR_0

The system ATR_0 adds to ACA_0 an axiom which states, informally, that any arithmetical functional (meaning any arithmetical formula with a free number variable n and a free class variable X, seen as the operator taking X to the set of n satisfying the formula) can

be iterated transfinitely along any countable well ordering starting with any set. ATR_0 is equivalent over ACA_0 to the principle of Σ^1_1 separation. ATR_0 is impredicative, and has the proof-theoretic ordinal Γ_0, the supremum of that of predicative systems.

ATR_0 proves the consistency of ACA_0, and thus by Gödel's theorem it is strictly stronger.

The following assertions are equivalent to ATR_0 over RCA_0:

- Any two countable well orderings are comparable. That is, they are isomorphic or one is isomorphic to a proper initial segment of the other.
- Ulm's theorem for countable reduced Abelian groups.
- The perfect set theorem, which states that every uncountable closed subset of a complete separable metric space contains a perfect closed set.
- Lusin's separation theorem (essentially Σ^1_1 separation).
- Determinacy for open sets in the Baire space.

Π^1_1 comprehension Π^1_1-CA_0

Π^1_1-CA_0 is stronger than arithmetical transfinite recursion and is fully impredicative. It consists of RCA_0 plus the comprehension scheme for Π^1_1 formulas.

In a sense, Π^1_1-CA_0 comprehension is to arithmetical transfinite recursion (Σ^1_1 separation) as ACA_0 is to weak König's lemma (Σ^0_1 separation). It is equivalent to several statements of descriptive set theory whose proofs make use of strongly impredicative arguments; this equivalence shows that these impredicative arguments cannot be removed.

The following theorems are equivalent to Π^1_1-CA_0 over RCA_0:

- The Cantor–Bendixson theorem (every closed set of reals is the union of a perfect set and a countable set).
- Every Abelian group is the direct sum of a divisible group and a reduced group.

Additional systems

- Weaker systems than recursive comprehension can be defined. The weak system

  RCA^*_0 consists of elementary function arithmetic EFA (the basic axioms plus Δ^0_0 induction in the enriched language with an exponential operation) plus Δ^0_1 comprehension. Over RCA^*_0, recursive comprehension as defined earlier (that is, with Σ^0_1 induction) is equivalent to the statement that a polynomial (over a countable field) has only finitely many roots and to the classification theorem for finitely generated Abelian groups. The system RCA^*_0 has the same proof theoretic ordinal ω^3 as EFA and is conservative over EFA for Π_{02} sentences.

- Weak Weak König's Lemma is the statement that a subtree of the infinite binary tree having no infinite paths has an asymptotically vanishing proportion of the leaves at length n (with a uniform estimate as to how many leaves of length n exist). An equivalent formulation is that any subset of Cantor space that has positive measure is nonempty (this is not provable in RCA_0). $WWKL_0$ is obtained by adjoining this axiom to RCA_0. It is equivalent to the statement that if the unit real interval is covered by a sequence of intervals then the sum of their lengths is at least one. The model theory of $WWKL_0$ is closely connected to the theory of algorithmically random sequences. In particular, an ω-model of RCA_0 satisfies weak weak König's lemma if and only if for every set X there is a set Y which is 1-random relative to X.

- DNR (short for "diagonally non-recursive") adds to RCA_0 an axiom asserting the existence of a diagonally non-recursive function relative to every set. That is, DNR states that, for any set A, there exists a total function f such that for all e the eth partial recursive function with oracle A is not equal to f. DNR is strictly weaker than WWKL (Lempp *et al.*, 2004).

- Δ^1_1-comprehension is in certain ways analogous to arithmetical transfinite recursion as recursive comprehension is to weak König's lemma. It has the hyperarithmetical sets as minimal ω-model. Arithmetical transfinite recursion proves Δ^1_1-comprehension but not the other way around.

- Σ^1_1-choice is the statement that if $\eta(n,X)$ is a Σ^1_1 formula such that for each n there exists an X satisfying η then there is a sequence of sets X_n such that $\eta(n,X_n)$ holds for each n. Σ^1_1-choice also has the hyperarithmetical sets as minimal ω-model. Arithmetical transfinite recursion proves Σ^1_1-choice but not the other way around.

ω-models and β-models

The ω in ω-model stands for the set of non-negative integers (or finite ordinals). An ω-model is a model for a fragment of second-order arithmetic whose first-order part is the standard model of Peano arithmetic, but whose second-order part may be non-standard. More precisely.an ω-model is given by a choice $S \subseteq 2^{\omega}$ of subsets of ω. The first order variables are interpreted in the usual way as elements of ω, and +, × have their usual meanings, while second order variables are interpreted as elements of S. There is a standard ω model where one just takes S to consist of all subsets of the integers. However there are also other ω-models; for example, RCA_0 has a minimal ω-model where S consists of the recursive subsets of ω.

A β model is an ω model that is equivalent to the standard ω-model for Π_{11} and Σ_{11} sentences (with parameters).

Non-ω models are also useful, especially in the proofs of conservation theorems.

Kolmogorov complexity

In algorithmic information theory (a subfield of computer science), the **Kolmogorov complexity** of an object, such as a piece of text, is a measure of the computational resources needed to specify the object.

Kolmogorov complexity is also known as **descriptive complexity**, **Kolmogorov-Chaitin complexity**, **stochastic complexity**, **algorithmic entropy**, or **program-size complexity**.

For example, consider the following two strings of length 64, each containing only lowercase letters, numbers, and spaces:

```
abababababababababababababababababababababababababababababababab
4c1j5b2p0cv4w1x8rx2y39umgw5q85s7uraqbjfdppa0q7nieieqe9noc4cvafzf
```

The first string has a short English-language description, namely "ab 32 times", which consists of 11 characters. The second one has no obvious simple description (using the same character set) other than writing down the string itself, which has 64 characters.

Definition

To define Kolmogorov complexity, we must first specify a description language for strings. Such a description language can be based on any programming language, such as Lisp, Pascal, or Java Virtual Machine bytecode. If **P** is a program which outputs a string *x*, then **P** is a description of *x*. The length of the description is just the length of **P** as a character string. In determining the length of **P**, the lengths of any subroutines used in **P** must be accounted for. The length of any integer constant *n* which occurs in the program **P** is the number of bits required to represent *n*, that is (roughly) $\log_2 n$.

We could alternatively choose an encoding for Turing machines, where an *encoding* is a function which associates to each Turing Machine **M** a bitstring <**M**>. If **M** is a Turing Machine which on input *w* outputs string *x*, then the concatenated string <**M**> *w* is a description of *x*. For theoretical analysis, this approach is more suited for constructing detailed formal proofs and is generally preferred in the research literature. The binary lambda calculus may provide the simplest definition of complexity yet.

Any string *s* has at least one description, namely the program

```
function GenerateFixedString()
   return s
```

If a description of *s*, *d*(*s*), is of minimal length—i.e. it uses the fewest number of characters—it is called a **minimal description** of *s*. Then the length of *d*(*s*)—i.e. the number of characters in the description—is the **Kolmogorov complexity** of *s*, written *K*(*s*). Symbolically,

$$K(s) = |d(s)|.$$

We now consider how the choice of description language affects the value of K and show that the effect of changing the description language is bounded.

Theorem. If K_1 and K_2 are the complexity functions relative to description languages L_1 and L_2, then there is a constant c (which depends only on the languages L_1 and L_2) such that

$$\forall s \ |K_1(s) - K_2(s)| \leq c.$$

Proof. By symmetry, it suffices to prove that there is some constant c such that for all bitstrings s,

$$K_1(s) \leq K_2(s) + c.$$

Now, suppose there is a program in the language L_1 which acts as an interpreter for L_2:

```
function InterpretLanguage(string p)
```

where p is a program in L_2. The interpreter is characterized by the following property:

> Running InterpretLanguage on input p returns the result of running p.

Thus if **P** is a program in L_2 which is a minimal description of s, then InterpretLanguage(**P**) returns the string s. The length of this description of s is the sum of

1. The length of the program InterpretLanguage, which we can take to be the constant c.
2. The length of **P** which by definition is $K_2(s)$.

This proves the desired upper bound.

History and context

Algorithmic information theory is the area of computer science that studies Kolmogorov complexity and other complexity measures on strings (or other data structures).

The concept and theory of Kolmogorov Complexity is based on a crucial theorem first discovered by Ray Solomonoff who published it in 1960, describing it in "A Preliminary Report on a General Theory of Inductive Inference" (see ref) as part of his invention of Algorithmic Probability. He gave a more complete description in his 1964 publications, "A Formal Theory of Inductive Inference," Part 1 and Part 2 in *Information and Control* (see ref).

Andrey Kolmogorov later independently published this theorem in *Problems Inform. Transmission*, 1, (1965), 1-7. Gregory Chaitin also presents this theorem in J. ACM, 16 (1969). Chaitin's paper was submitted October 1966, revised in December 1968 and cites both Solomonoff's and Kolmogorov's papers.

The theorem says that among algorithms that decode strings from their descriptions (codes) there exists an optimal one. This algorithm, for all strings, allows codes as short as allowed by any other algorithm up to an additive constant that depends on the algorithms, but not on the strings themselves. Solomonoff used this algorithm, and the code lengths it allows, to define a string's `universal probability' on which inductive inference of a string's subsequent digits can be based. Kolmogorov used this theorem to define several functions of strings: complexity, randomness, and information.

When Kolmogorov became aware of Solomonoff's work, he acknowledged Solomonoff's priority (IEEE Trans. Inform Theory, 14:5(1968), 662-664). For several years, Solomonoff's work was better known in the Soviet Union than in the Western World. The general consensus in the scientific community, however, was to associate this type of complexity with Kolmogorov, who was concerned with randomness of a sequence while Algorithmic Probability became associated with Solomonoff, who focused on prediction using his invention of the universal a priori probability distribution.

There are several other variants of Kolmogorov complexity or algorithmic information. The most widely used one is based on self-delimiting programs and is mainly due to Leonid Levin (1974).

An axiomatic approach to Kolmogorov complexity based on Blum axioms (Blum 1967) was introduced by Mark Burgin in the paper presented for publication by Andrey Kolmogorov (Burgin 1982). This approach was further developed in the book (Burgin 2005) and applied to software metrics (Burgin and Debnath, 2003; Debnath and Burgin, 2003).

Some consider that naming the concept "Kolmogorov complexity" is an example of the Matthew effect.

Basic results

In the following discussion let $K(s)$ be the complexity of the string s.

It is not hard to see that the minimal description of a string cannot be too much larger than the string itself: the program GenerateFixedString above that outputs s is a fixed amount larger than s.

Theorem. There is a constant c such that

$$\forall s \; K(s) \leq |s| + c.$$

Incomputability of Kolmogorov complexity

The first result is that there is no way to effectively compute K.

Theorem. K is not a computable function.

In other words, there is no program which takes a string s as input and produces the integer $K(s)$ as output. We show this by contradiction by making a program that creates a string that should only be able to be created by a longer program. Suppose there is a program

```
function KolmogorovComplexity(string s)
```

that takes as input a string s and returns $K(s)$. Now consider the program

```
function GenerateComplexString(int n)
   for i = 1 to infinity:
      for each string s of length exactly i
         if KolmogorovComplexity(s) >= n
            return s
            quit
```

This program calls KolmogorovComplexity as a subroutine. This program tries every string, starting with the shortest, until it finds a string with complexity at least n, then returns that string. Therefore, given any positive integer n, it produces a string with Kolmogorov complexity at least as great as n. The program itself has a fixed length U. The input to the program GenerateComplexString is an integer n; here, the size of n is measured by the number of bits required to represent n which is $\log_2(n)$. Now consider the following program:

```
function GenerateParadoxicalString()
    return GenerateComplexString(n_0)
```

This program calls GenerateComplexString as a subroutine and also has a free parameter n_0. This program outputs a string s whose complexity is at least n_0. By an auspicious choice of the parameter n_0 we will arrive at a contradiction. To choose this value, note s is described by the program GenerateParadoxicalString whose length is at most

$$U + \log_2(n_0) + C$$

where C is the "overhead" added by the program GenerateParadoxicalString. Since n grows faster than $\log_2(n)$, there exists a value n_0 such that

$$U + \log_2(n_0) + C < n_0.$$

But this contradicts the definition of having a complexity at least n_0. That is, by the definition of $K(s)$, the string s returned by GenerateParadoxicalString is only supposed to be able to be generated by a program of length n_0 or longer, but

GenerateParadoxicalString is shorter than n_0. Thus the program named "KolmogorovComplexity" cannot actually computably find the complexity of arbitrary strings.

This is proof by contradiction where the contradiction is similar to the Berry paradox: "Let *n* be the smallest positive integer that cannot be defined in fewer than twenty English words." It is also possible to show the uncomputability of K by reduction from the uncomputability of the halting problem H, since K and H are Turing-equivalent.

In the programming languages community there is a corollary known as the Full employment theorem, stating there is no perfect size-optimizing compiler.

Chain rule for Kolmogorov complexity

The chain rule for Kolmogorov complexity states that

$$K(X,Y) = K(X) + K(Y|X) + O(\log(K(X,Y))).$$

It states that the shortest program that reproduces *X* and *Y* is no more than a logarithmic term larger than a program to reproduce *X* and a program to reproduce *Y* given *X*. Using this statement one can define an analogue of mutual information for Kolmogorov complexity.

Compression

It is straightforward to compute upper bounds for *K*(*s*): simply compress the string *s* with some method, implement the corresponding decompressor in the chosen language, concatenate the decompressor to the compressed string, and measure the resulting string's length.

A string *s* is compressible by a number *c* if it has a description whose length does not exceed | *s* | − *c*. This is equivalent to saying $K(s) \leq |s| - c$. Otherwise *s* is incompressible by *c*. A string incompressible by 1 is said to be simply *incompressible*; by the pigeonhole principle, incompressible strings must exist, since there are 2^n bit strings of length *n* but only $2^n - 1$ shorter strings, that is strings of length *n* − 1 or less.

For the same reason, most strings are complex in the sense that they cannot be significantly compressed: *K*(*s*) is not much smaller than | *s* | , the length of *s* in bits. To make this precise, fix a value of *n*. There are 2^n bitstrings of length *n*. The uniform probability distribution on the space of these bitstrings assigns exactly equal weight 2^{-n} to each string of length *n*.

Theorem. With the uniform probability distribution on the space of bitstrings of length *n*, the probability that a string is incompressible by *c* is at least $1 - 2^{-c+1} + 2^{-n}$.

To prove the theorem, note that the number of descriptions of length not exceeding $n - c$ is given by the geometric series:

$$1 + 2 + 2^2 + \cdots + 2^{n-c} = 2^{n-c+1} - 1.$$

There remain at least

$$2^n - 2^{n-c+1} + 1$$

many bitstrings of length n that are incompressible by c. To determine the probability divide by 2^n.

Chaitin's incompleteness theorem

We know that, in the set of all possible strings, most strings are complex in the sense that they cannot be described in any significantly "compressed" way. However, it turns out that the fact that a specific string is complex cannot be formally proved, if the string's complexity is above a certain threshold. The precise formalization is as follows. First fix a particular axiomatic system **S** for the natural numbers. The axiomatic system has to be powerful enough so that to certain assertions **A** about complexity of strings one can associate a formula $\mathbf{F_A}$ in **S**. This association must have the following property: if $\mathbf{F_A}$ is provable from the axioms of **S**, then the corresponding assertion **A** is true. This "formalization" can be achieved either by an artificial encoding such as a Gödel numbering or by a formalization which more clearly respects the intended interpretation of **S**.

Theorem. There exists a constant L (which only depends on the particular axiomatic system and the choice of description language) such that there does not exist a string s for which the statement

$$K(s) \geq L$$

(as formalized in **S**) can be proven within the axiomatic system **S**.

Note that by the abundance of nearly incompressible strings, the vast majority of those statements must be true.

The proof of this result is modeled on a self-referential construction used in Berry's paradox. The proof is by contradiction. If the theorem were false, then

Assumption (X): For any integer n there exists a string s for which there is a proof in **S** of the formula "$K(s) \geq n$" (which we assume can be formalized in **S**).

We can find an effective enumeration of all the formal proofs in **S** by some procedure

```
function NthProof(int n)
```

which takes as input n and outputs some proof. This function enumerates all proofs. Some of these are proofs for formulas we do not care about here (examples of proofs which will be listed by the procedure NthProof are the various known proofs of the law of quadratic reciprocity, those of Fermat's little theorem or the proof of Fermat's last theorem all translated into the formal language of **S**). Some of these are complexity formulas of the form $K(s) \geq n$ where s and n are constants in the language of **S**. There is a program

```
function NthProofProvesComplexityFormula(int n)
```

which determines whether the n^{th} proof actually proves a complexity formula $K(s) \geq L$. The strings s and the integer L in turn are computable by programs:

```
function StringNthProof(int n)
function ComplexityLowerBoundNthProof(int n)
```

Consider the following program

```
function GenerateProvablyComplexString(int n)
   for i = 1 to infinity:
      if  NthProofProvesComplexityFormula(i) and
ComplexityLowerBoundNthProof(i) ≥ n
         return StringNthProof(i)
         quit
```

Given an n, this program tries every proof until it finds a string and a proof in the formal system **S** of the formula $K(s) \geq L$ for some $L \geq n$. The program terminates by our **Assumption (X)**. Now this program has a length U. There is an integer n_0 such that $U + \log_2(n_0) + C < n_0$, where C is the overhead cost of

```
function GenerateProvablyParadoxicalString()
   return GenerateProvablyComplexString(n0)
   quit
```

The program GenerateProvablyParadoxicalString outputs a string s for which there exists an L such that $K(s) \geq L$ can be formally proved in **S** with $L \geq n_0$. In particular $K(s) \geq n_0$ is true. However, s is also described by a program of length $U + \log_2(n_0) + C$ so its complexity is less than n_0. This contradiction proves **Assumption (X)** cannot hold.

Similar ideas are used to prove the properties of Chaitin's constant.

Minimum message length

The minimum message length principle of statistical and inductive inference and machine learning was developed by C.S. Wallace and D.M. Boulton in 1968. MML is Bayesian (it incorporates prior beliefs) and information-theoretic. It has the desirable properties of statistical invariance (the inference transforms with a re-parametrisation, such as from polar coordinates to Cartesian coordinates), statistical consistency (even for very hard

problems, MML will converge to any underlying model) and efficiency (the MML model will converge to any true underlying model about as quickly as is possible). C.S. Wallace and D.L. Dowe (1999) showed a formal connection between MML and algorithmic information theory (or Kolmogorov complexity).

Kolmogorov randomness

Kolmogorov randomness (also called *algorithmic randomness*) defines a string (usually of bits) as being random if and only if it is shorter than any computer program that can produce that string. This definition of randomness is critically dependent on the definition of Kolmogorov complexity. To make this definition complete, a computer has to be specified, usually a Turing machine. According to the above definition of randomness, a random string is also an "incompressible" string, in the sense that it is impossible to give a representation of the string using a program whose length is shorter than the length of the string itself. However, according to this definition, most strings shorter than a certain length end up to be (Chaitin-Kolmogorovically) random because the best one can do with very small strings is to write a program that simply prints these strings.

Chapter 5

Church–Turing Thesis

In computability theory, the **Church–Turing thesis** (also known as the **Church-Turing conjecture**, **Church's thesis**, **Church's conjecture**, and **Turing's thesis**) is a combined hypothesis ("thesis") about the nature of functions whose values are effectively calculable; i.e. computable. In simple terms, it states that "everything computable is computable by a Turing machine."

Several attempts were made in the first half of the 20th Century to formalize the notion of computability:

- American mathematician Alonzo Church created a method for defining functions called the λ-calculus,
- British mathematician Alan Turing created a theoretical model for a machine that could carry out calculations from inputs,
- Church, along with mathematician Stephen Kleene and logician J.B. Rosser created a formal definition of a class of functions whose values could be calculated by recursion.

All three computational processes (recursion, the λ-calculus, and the Turing machine) were shown to be equivalent—all three approaches define the same class of functions. This has led mathematicians and computer scientists to believe that the concept of computability is accurately characterized by these three equivalent processes. Informally the Church–Turing thesis states that if some method (algorithm) exists to carry out a calculation, then the same calculation can also be carried out by a Turing machine (as well as by a recursively-definable function, and by a λ-function).

The Church–Turing thesis is a statement that characterizes the nature of computation and cannot be formally proven. Even though the three processes mentioned above proved to be equivalent, the fundamental premise behind the thesis -- the notion of what it means for a function to be "effectively calculable" (computable) -- is "a somewhat vague intuitive one". Thus, the "thesis" remains an hypothesis.

Despite the fact that it cannot be formally proven, the Church–Turing thesis now has near-universal acceptance.

Formal statement

Rosser 1939 addresses the notion of "effective computability" as follows: "Clearly the existence of CC and RC (Church's and Rosser's proofs) presupposes a precise definition of "effective". "Effective method" is here used in the rather special sense of a method each step of which is precisely predetermined and which is certain to produce the answer in a finite number of steps". Thus the adverb-adjective "effective" is used in a sense of "1a: producing a decided, decisive, or desired effect", and "capable of producing a result".

In the following, the words "effectively calculable" will mean "produced by any intuitively 'effective' means whatsoever" and "effectively computable" will mean "produced by a Turing-machine or equivalent mechanical device". Turing's 1939 "definitions" are virtually the same:

"†We shall use the expression "computable function" to mean a function calculable by a machine, and we let "effectively calculable" refer to the intuitive idea without particular identification with any one of these definitions."(cf. the footnote † in Turing 1939 (his Ordinals paper) in Davis 1965:160).

The thesis can be stated as follows:

Every effectively calculable function is a computable function.

Turing stated it this way:

"It was stated ... that 'a function is effectively calculable if its values can be found by some purely mechanical process.' We may take this literally, understanding that by a purely mechanical process one which could be carried out by a machine. The development ... leads to ... an identification of computability † with effective calculability" († is the footnote above, ibid).

History

One of the important problems for logicians in the 1930s was David Hilbert's Entscheidungsproblem, which asked if there was a mechanical procedure for separating mathematical truths from mathematical falsehoods. This quest required that the notion of “algorithm” or “effective calculability” be pinned down, at least well enough for the quest to begin. But from the very outset Alonzo Church's attempts began with a debate that continues to this day. Was the notion of “effective calculability” to be (i) an "axiom or axioms" in an axiomatic system, or (ii) merely a *definition* that “identified” two or more propositions, or (iii) an *empirical hypothesis* to be verified by observation of natural events, or (iv) or just *a proposal* for the sake of argument (i.e. a "thesis").

Circa 1930–1952

In the course of studying the problem, Church and his student Stephen Kleene introduced the notion of λ-definable functions, and they were able to prove that several large classes of functions frequently encountered in number theory were λ-definable. The debate began when Church proposed to Kurt Gödel that one should define the "effectively computable" functions as the λ-definable functions. Gödel, however, was not convinced and called the proposal "thoroughly unsatisfactory". Rather in correspondence with Church (ca 1934–5), Gödel proposed *axiomatizing* the notion of "effective calculability"; indeed, in a 1935 letter to Kleene, Church reported that:

"His [Gödel's] only idea at the time was that it might be possible, in terms of effective calculability as an undefined notion, to state a set of axioms which would embody the generally accepted properties of this notion, and to do something on that basis".

But Gödel offered no further guidance. Eventually, he would suggest his (primitive) recursion, modified by Herbrand's suggestion, that he (Gödel) had detailed in his 1934 lectures in Princeton NJ (Kleene and another student J. B. Rosser transcribed the notes.). But "he did not think that the two ideas could be satisfactorily identified "except heuristically".

Next, it was necessary to identify and prove the equivalence of two notions of effective calculability. Equipped with the λ-calculus and "general" recursion, Stephen Kleene with help of Church and J. B. Rosser produced proofs (1933, 1935) to show that the two calculi are equivalent. Church subsequently modified his methods to include use of Herbrand–Gödel recursion and then proved (1936) that the Entscheidungsproblem is unsolvable: There is no generalized "effective calculation" (method, algorithm) that can determine whether or not a formula in either the recursive- or λ-calculus is "valid" (more precisely: no method to show that a well formed formula has a "normal form").

Many years later in a letter to Davis (ca 1965), Gödel would confess that "he was, at the time of these [1934] lectures, not at all convinced that his concept of recursion comprised all possible recursions". By 1963-4 Gödel would disavow Herbrand–Gödel recursion and the λ-calculus in favor of the Turing machine as the definition of “algorithm” or “mechanical procedure” or “formal system”.

An hypothesis leading to a natural law?: In late 1936 Alan Turing’s paper (also proving that the Entscheidungsproblem is unsolvable) had not yet appeared. On the other hand, Emil Post's 1936 paper had appeared and was certified independent of Turing's work. Post strongly disagreed with Church’s “identification” of effective computability with the λ-calculus and recursion, stating:

"Actually the work already done by Church and others carries this identification considerably beyond the working hypothesis stage. But to mask this identification under a definition . . . blinds us to the need of its continual verification."

Rather, he regarded the notion of “effective calculability” as merely a "working hypothesis" that might lead by inductive reasoning to a "natural law" rather than by “a definition or an axiom”. This idea was "sharply" criticized by Church.

Thus Post in his 1936 was also discounting Kurt Gödel's suggestion to Church in 1934–5 that the thesis might be expressed as an axiom or set of axioms.

Turing adds another definition, Rosser equates all three: Within just a short time, Turing's 1936–37 paper "On Computable Numbers, with an Application to the Entscheidungsproblem" appeared. In it he asserted another notion of "effective computability" with the introduction of his a-machines (now known as the Turing machine abstract computational model). And in a proof-sketch added as an "Appendix" to his 1936–37 paper, Turing showed that the classes of functions defined by λ-calculus and Turing machines coincided.

In a few years (1939) Turing would propose, like Church and Kleene before him, that *his* formal definition of mechanical computing agent was the correct one. Thus, by 1939, both Church (1934) and Turing (1939), neither having knowledge of the other’s efforts, had individually proposed that their "formal systems" should be *definitions* of "effective calculability"; neither framed their assertions as *theses*.

Rosser (1939) formally identified the three notions-as-definitions:

"All three *definitions* are equivalent, so it does not matter which one is used."

Kleene proposes *Church's Thesis*: This left the overt expression of a "thesis” to Kleene. In his 1943 paper *Recursive Predicates and Quantifiers* Kleene proposed his "THESIS I":

"This heuristic fact [general recursive functions are effectively calculable]...led Church to state the following thesis([22]). The same thesis is implicit in Turing's description of computing machines([23]).
"THESIS I. *Every effectively calculable function (effectively decidable predicate) is general recursive* [Kleene's italics]
"Since a precise mathematical definition of the term effectively calculable (effectively decidable) has been wanting, we can take this thesis ... as a definition of it..."
"([22]) references Church 1936
"([23]) references Turing 1936–7

Kleene goes on to note that:

"...the thesis has the character of an hypothesis—a point emphasized by Post and by Church([24]). If we consider the thesis and its converse as definition, then the hypothesis is an hypothesis about the application of the mathematical theory developed from the definition. For the acceptance of the hypothesis, there are, as we have suggested, quite compelling grounds."

Kleene's Church-Turing Thesis: A few years later (1952) Kleene would overtly name, defend, and express the two "theses" and then "identify" them (show equivalence) by use of his Theorem XXX:

"Heuristic evidence and other considerations led Church 1936 to propose the following thesis.
Thesis I. *Every effectively calculable function (effectively decidable predicate) is general recursive*.
Theorem XXX: "The following classes of partial functions are coextensive, i.e. have the same members: (a) the partial recursive functions, (b) the computable functions. . . ".
Turing's thesis: "Turing's thesis that every function which would naturally be regarded as computable is computable under his definition, i.e. by one of his machines, is equivalent to Church's thesis by Theorem XXX."

Later developments

An attempt to understand the notion of "effective computability" better led Robin Gandy (Turing's student and friend) in 1980 to analyze *machine* computation (as opposed to human-computation acted out by a Turing machine). Gandy's curiosity about, and analysis of, "cellular automata", "Conway's game of life", "parallelism" and "crystalline automata" led him to propose four "principles (or constraints) ... which it is argued, any machine must satisfy." His most-important fourth, "the principle of causality" is based on the "finite velocity of propagation of effects and signals; contemporary physics rejects the possibility of instantaneous action at a distance." From these principles and some additional constraints—(1a) a lower bound on the linear dimensions of any of the parts, (1b) an upper bound on speed of propagation (the velocity of light), (2) discrete progress of the machine, and (3) deterministic behavior—he produces a theorem that "What can be calculated by a device satisfying principles I–IV is computable.".

In the late 1990s Wilfried Sieg analyzed Turing's and Gandy's notions of "effective calculability" with the intent of "sharpening the informal notion, formulating its general features axiomatically, and investigating the axiomatic framework". In his 1997 and 2002 Sieg presents a series of constraints on the behavior of a *computor* -- "a human computing agent who proceeds mechanically"; these constraints reduce to:

- "(B.1) (Boundedness) *There is a fixed bound on the number of symbolic configurations a computor can immediately recognize.*
- "(B.2) (Boundedness) *There is a fixed bound on the number of internal states a computor can be in.*
- "(L.1) (Locality) *A computor can change only elements of an observed symbolic configuration.*
- "(L.2) (Locality) *A computor can shift attention from one symbolic configuration to another one, but the new observed configurations must be within a bounded distance of the immediately previously observed configuration.*
- "(D) (Determinacy) *The immediately recognizable (sub-)configuration determines uniquely the next computation step (and id [instantaneous description])*"; stated

another way: "*A computor's internal state together with the observed configuration fixes uniquely the next computation step and the next internal state.*"

The matter remains in active discussion within the academic community.

Success of the thesis

Other formalisms (besides recursion, the λ-calculus, and the Turing machine) have been proposed for describing effective calculability/computability . Stephen Kleene (1952) adds to the list the functions "*reckonable* in the system S_1" of Kurt Gödel 1936, and Emil Post's (1943, 1946) "*canonical* [also called *normal*] *systems*". In the 1950s Hao Wang and Martin Davis greatly simplified the one-tape Turing-machine model. Marvin Minsky expanded the model to two or more tapes and greatly simplified the tapes into "up-down counters", which Melzak and Lambek further evolved into what is now known as the counter machine model. In the late 1960s and early 1970s researchers expanded the counter machine model into the register machine, a close cousin to the modern notion of the computer. Other models include combinatory logic and Markov algorithms. Gurevich adds the pointer machine model of Kolmogorov and Uspensky (1953, 1958): "...they just wanted to ... convince themselves that there is no way to extend the notion of computable function."

All these contributions involve proofs that the models are computationally equivalent to the Turing machine; such models are said to be Turing complete. Because all these different attempts at formalizing the concept of "effective calculability/computability" have yielded equivalent results, it is now generally assumed that the Church–Turing thesis is correct. In fact, Gödel (1936) proposed something stronger than this; he observed that there was something "absolute" about the concept of "reckonable in S_1":

"It may also be shown that a function which is computable ['reckonable'] in one of the systems S_i, or even in a system of transfinite type, is already computable [reckonable] in S_1. Thus the concept 'computable' ['reckonable'] is in a certain definite sense 'absolute', while practically all other familiar metamathematical concepts (e.g. provable, definable, etc.) depend quite essentially on the system to which they are defined"

Informal Usage in Proofs

Proofs in computability theory often invoke the Church–Turing thesis in an informal way to establish the computability of functions while avoiding the (often very long) details which would be involved in a rigorous, formal proof. To establish that a function is computable by Turing machine, it is usually considered sufficient to give an informal English description of how the function can be effectively computed, and then conclude "By the Church–Turing thesis" that the function is Turing computable (equivalently partial recursive).

Dirk van Dalen (in Gabbay 2001:284) gives the following example for the sake of illustrating this informal use of the Church-Turing thesis:

EXAMPLE: Each infinite RE set contains an infinite recursive set.
Proof: Let A be infinite RE. We list the elements of A effectively, n_0, n_1, n_2, n_3, ...
From this list we extract an increasing sublist: put $m_0=n_0$, after finitely many steps we find an n_k such that $n_k > m_0$, put $m_1=n_k$. We repeat this procedure to find $m_2 > m_1$, etc. this yields an effective listing of the subset $B=\{m_0,m_1,m_2,...\}$ of A, with the property $m_i < m_{i+1}$.
Claim. B is decidable. For, in order to test k in B we must check if $k=m_i$ for some i. Since the sequence of m_i's is increasing we have to produce at most k+1 elements of the list and compare them with k. If none of them is equal to k, then k not in B. Since this test is effective, B is decidable and, **by Church's thesis**, recursive.

(Emphasis added). In order to make the above example completely rigorous, one would have to carefully construct a Turing Machine, or λ-function, or carefully invoke recursion axioms, or at best, cleverly invoke various theorems of computability theory. But because the computability theorist believes that Turing computability correctly captures what can be computed effectively, and because an effective procedure is spelled out in English for deciding the set B, the computability theorist accepts this as proof that the set is indeed recursive.

As a rule of thumb, the Church–Turing thesis should only be invoked to simplify proofs in cases where the writer would be capable of, and expects the readers also to be capable of, easily (but not necessarily without tedium) producing a rigorous proof if one were demanded

Variations

The success of the Church–Turing thesis prompted variations of the thesis to be proposed. For example, the **Physical Church–Turing thesis** (PCTT) states:

"According to Physical CTT, all physically computable functions are Turing-computable"

The Church-Turing thesis says nothing about the efficiency with which one model of computation can simulate another. It has been proved for instance that a (multi-tape) universal Turing machine only suffers a logarithmic slowdown factor in simulating any Turing machine. No such result has been proved in general for an arbitrary but *reasonable* model of computation. A variation of the Church-Turing thesis that addresses this issue is the **Feasibility Thesis** or **(Classical) Complexity-Theoretic Church–Turing Thesis** (SCTT), which is not due to Church or Turing, but rather was realized gradually in the development of complexity theory. It states:

"*A probabilistic Turing machine can efficiently simulate any realistic model of computation.*"

The word 'efficiently' here means up to polynomial-time reductions. This thesis was originally called *Computational Complexity-Theoretic Church–Turing Thesis* by Ethan Bernstein and Umesh Vazirani (1997). The Complexity-Theoretic Church–Turing Thesis, then, posits that all 'reasonable' models of computation yield the same class of problems that can be computed in polynomial time. Assuming the conjecture that probabilistic polynomial time (BPP) equals deterministic polynomial time (P), the word 'probabilistic' is optional in the Complexity-Theoretic Church–Turing Thesis. A similar thesis, called the *Invariant Thesis*, was introduced by Cees F. Slot and Peter van Emde Boas. It states: *"Reasonable" machines can simulate each other within a polynomially bounded overhead in time and a constant-factor overhead in space*. The thesis originally appeared in a paper at STOC'84, which was the first paper to show that polynomial-time overhead and constant-space overhead could be *simultaneously* achieved for a simulation of a Random Access Machine on a Turing machine.

If production-scale quantum computers can be built, they could invalidate the Complexity-Theoretic Church–Turing Thesis, since it is also conjectured that quantum polynomial time (BQP) is larger than BPP. In other words, there are efficient quantum algorithms that perform tasks that are not known to have efficient probabilistic algorithms; for example, factoring integers. They would not however invalidate the original Church–Turing thesis, since a quantum computer can always be simulated by a Turing machine, but they would invalidate the classical Complexity-Theoretic Church-Turing thesis for efficiency reasons. Consequently, the **Quantum Complexity-Theoretic Church-Turing thesis** states:

"A quantum Turing machine can efficiently simulate any realistic model of computation."

Philosophical implications

The Church–Turing thesis has some profound implications for the philosophy of mind; however many of the philosophical interpretations of the Thesis involve basic misunderstandings of the thesis statement. B. Jack Copeland states that it's an open empirical question whether there are actual deterministic physical processes that, in the long run, elude simulation by a Turing machine; furthermore, he states that it is an open empirical question whether any such processes are involved in the working of the human brain. There are also some important open questions which cover the relationship between the Church–Turing thesis and physics, and the possibility of hypercomputation. When applied to physics, the thesis has several possible meanings:

1. The universe is equivalent to a Turing machine; thus, computing non-recursive functions is physically impossible. This has been termed the Strong Church–Turing thesis and is a foundation of digital physics.
2. The universe is not equivalent to a Turing machine (i.e., the laws of physics are not Turing-computable), but incomputable physical events are not "harnessable" for the construction of a hypercomputer. For example, a universe in which physics involves real numbers, as opposed to computable reals, might fall into this category.

3. The universe is a hypercomputer, and it is possible to build physical devices to harness this property and calculate non-recursive functions. For example, it is an open question whether all quantum mechanical events are Turing-computable, although it is known that rigorous models such as quantum Turing machines are equivalent to deterministic Turing machines. (They are not necessarily efficiently equivalent; see above.) John Lucas and, more famously, Roger Penrose have suggested that the human mind might be the result of some kind of quantum-mechanically enhanced, "non-algorithmic" computation, although there is no scientific evidence for this proposal.

There are many other technical possibilities which fall outside or between these three categories, but these serve to illustrate the range of the concept.

Non-computable functions

One can formally define functions that are not computable. A well known example of such a function is the busy beaver function. This function takes an input *n* and returns the largest number of symbols that a Turing machine with *n* states can print before halting, when run with no input. Using particular models of Turing machines, researchers have computed the value of this function for small values of *n*: 0 through 4. Simulations of Turing machines with 5 and 6 states have been performed, but without conclusive results. For higher values, only lower bounds have been given. Finding an upper bound on the busy beaver function is equivalent to solving the halting problem, a problem known to be unsolvable by Turing machines. Since the busy beaver function cannot be computed by Turing machines, the Church–Turing thesis asserts that this function cannot be effectively computed by any method.

Mark Burgin, Eugene Eberbach, Peter Kugel, and other researchers argue that super-recursive algorithms such as inductive Turing machines disprove the Church–Turing thesis. Their argument relies on a definition of algorithm broader than the ordinary one, so that non-computable functions obtained from some inductive Turing machines are called computable. This interpretation of the Church–Turing thesis differs from the interpretation commonly accepted in computability theory, discussed above. The argument that super-recursive algorithms are indeed algorithms in the sense of the Church–Turing thesis has not found broad acceptance within the computability research community.

Chapter 6

Computable Function and Computability Logic

Computable function

Computable functions are the basic objects of study in computability theory. Computable functions are the formalized analogue of the intuitive notion of algorithm. They are used to discuss computability without referring to any concrete model of computation such as Turing machines or register machines. Any definition, however, must make reference to some specific model of computation but all valid definitions yield the same class of functions. Particular models of computability that give rise to the set of computable functions are the Turing-computable functions and the μ-recursive functions.

Before the precise definition of computable function, mathematicians often used the informal term *effectively calculable*. This term has since come to be identified with the computable functions. Note that the effective computability of these functions does not imply that they can be *efficiently* computed (i.e. computed within a reasonable amount of time). In fact, for some effectively calculable functions it can be shown that any algorithm that computes them will be very inefficient in the sense that the running time of the algorithm increases exponentially (or even superexponentially) with the length of the input. The fields of feasible computability and computational complexity study functions that can be computed efficiently.

According to the Church-Turing thesis, computable functions are exactly the functions that can be calculated using a mechanical calculation device given unlimited amounts of time and storage space. Equivalently, this thesis states that any function which has an algorithm is computable. Note that an algorithm in this sense is understood to be a sequence of steps a person with unlimited time and an infinite supply of pen and paper could follow.

The Blum axioms can be used to define an abstract computational complexity theory on the set of computable functions. In computational complexity theory, the problem of determining the complexity of a computable function is known as a function problem.

Definition

The class of computable functions can be defined in many equivalent models, including

- **Turing machines**
- **μ-recursive functions**
- **Lambda calculus**
- **Post machines** (**Post-Turing machines** and **tag machines**).
- **Register machines**

Although those models use different representations for the functions, their inputs and their outputs, translations exist between any two models. In the remainder here, functions from natural numbers to natural numbers are used (as is the case for, e.g., the μ-recursive functions).

Each computable function f takes a fixed, finite number of natural numbers as arguments. Because the functions are partial in general, they may not be defined for every possible choice of input. If a computable function is defined for a certain input, then it returns a single natural number as output (this output can be interpreted as a list of numbers using a pairing function). These functions are also called **partial recursive functions**. In computability theory, the **domain** of a function is taken to be the set of all inputs for which the function is defined.

A function which is defined for all possible arguments is called **total**. If a computable function is total, it is called a **total computable function** or **total recursive function**.

The notation $f(x_1, ..., x_k)\downarrow$ indicates that the partial function f is defined on arguments x_1, ..., x_k, and the notation $f(x_1, ..., x_k) = y$ indicates that f is defined on the arguments x_1, ..., x_k and the value returned is y. The case that a function f is undefined for arguments x_1, ..., x_k is denoted by $f(x_1, ..., x_k)\uparrow$.

Characteristics of computable functions

The basic characteristic of a computable function is that there must be a finite procedure (an algorithm) telling how to compute the function. The models of computation listed above give different interpretations of what a procedure is and how it is used, but these interpretations share many properties. The fact that these models give equivalent classes of computable functions stems from the fact that each model is capable of reading and mimicking a procedure for any of the other models, much as a compiler is able to read instructions in one computer language and emit instructions in another language.

Enderton [1977] gives the following characteristics of a procedure for computing a computable function; similar characterizations have been given by Turing [1936], Rogers [1967], and others.

- "There must be exact instructions (i.e. a program), finite in length, for the procedure."

Thus every computable function must have a finite program that completely describes how the function is to be computed. It is possible to compute the function by just following the instructions; no guessing or special insight is required.

- "If the procedure is given a *k*-tuple **x** in the domain of *f*, then after a finite number of discrete steps the procedure must terminate and produce ***f(x)***."

Intuitively, the procedure proceeds step by step, with a specific rule to cover what to do at each step of the calculation. Only finitely many steps can be carried out before the value of the function is returned.

- "If the procedure is given a *k*-tuple **x** which is not in the domain of *f*, then the procedure might go on forever, never halting. Or it might get stuck at some point, but it must not pretend to produce a value for *f* at **x**."

Thus if a value for ***f(x)*** is ever found, it must be the correct value. It is not necessary for the computing agent to distinguish correct outcomes from incorrect ones because the procedure is always correct when it produces an outcome.

Enderton goes on to list several clarifications of these requirements of the procedure for a computable function:

- The procedure must theoretically work for arbitrarily large arguments. It is not assumed that the arguments are smaller than the number of atoms in the Earth, for example.
- The procedure is required to halt after finitely many steps in order to produce an output, but it may take arbitrarily many steps before halting. No time limitation is assumed.
- Although the procedure may use only a finite amount of storage space during a successful computation, there is no bound on the amount of space that is used. It is assumed that additional storage space can be given to the procedure whenever the procedure asks for it.

The field of computational complexity studies functions with prescribed bounds on the time and/or space allowed in a successful computation.

Computable sets and relations

A set A of natural numbers is called **computable** (synonyms: **recursive**, **decidable**) if there is a computable, total function f such that for any natural number n, $f(n) = 1$ if n is in A and $f(n) = 0$ if n is not in A.

A set of natural numbers is called **computably enumerable** (synonyms: **recursively enumerable**, **semidecidable**) if there is a computable function f such that for each number n, $f(n)$ is defined if and only if n is in the set. Thus a set is computably enumerable if and only if it is the domain of some computable function. The word *enumerable* is used because the following are equivalent for a nonempty subset B of the natural numbers:

- B is the domain of a computable function.
- B is the range of a total computable function. If B is infinite then the function can be assumed to be injective.

If a set B is the range of a function f then the function can be viewed as an enumeration of B, because the list $f(0), f(1), \ldots$ will include every element of B.

Because each finitary relation on the natural numbers can be identified with a corresponding set of finite sequences of natural numbers, the notions of **computable relation** and **computably enumerable relation** can be defined from their analogues for sets.

Formal languages

In computability theory in computer science, it is common to consider formal languages. An **alphabet** is an arbitrary set. A **word** on an alphabet is a finite sequence of symbols from the alphabet; the same symbol may be used more than once. For example, binary strings are exactly the words on the alphabet $\{0, 1\}$. A **language** is a subset of the collection of all words on a fixed alphabet. For example, the collection of all binary strings that contain exactly 3 ones is a language over the binary alphabet.

A key property of a formal language is the level of difficulty required to decide whether a given word is in the language. Some coding system must be developed to allow a computable function to take an arbitrary word in the language as input; this is usually considered routine. A language is called **computable** (synonyms: **recursive**, **decidable**) if there is a computable function f such that for each word w over the alphabet, $f(w) = 1$ if the word is in the language and $f(w) = 0$ if the word is not in the language. Thus a language is computable just in case there is a procedure that is able to correctly tell whether arbitrary words are in the language.

A language is **computably enumerable** (synonyms: **recursively enumerable**, **semidecidable**) if there is a computable function f such that $f(w)$ is defined if and only if the word w is in the language. The term *enumerable* has the same etymology as in computably enumerable sets of natural numbers.

Examples

The following functions are computable:

- Each function with a finite domain; e.g., any finite sequence of natural numbers.
- Each constant function $f : \mathbf{N}^k \to \mathbf{N}, f(n_1,...n_k) := n$.
- Addition $f : \mathbf{N}^2 \to \mathbf{N}, f(n_1,n_2) := n_1 + n_2$
- The function which gives the list of prime factors of a number.
- The greatest common divisor of two numbers is a computable function.
- Bézout's identity, a linear Diophantine equation

If f and g are computable, then so are: $f + g$, $f * g$, $f \circ g$ if f is unary, $\max(f,g)$, $\min(f,g)$, $\arg\max\{y \leq f(x)\}$ and many more combinations.

The following examples illustrate that a function may be computable though it is not known which algorithm computes it.

- The function f such that $f(n) = 1$ if there is a sequence of n consecutive fives in the decimal expansion of π, and $f(n) = 0$ otherwise, is computable. (The function f is either the constant 1 function, which is computable, or else there is a k such that $f(n) = 1$ if $n < k$ and $f(n) = 0$ if $n \geq k$. Every such function is computable. It is not known whether there are arbitrarily long runs of fives in the decimal expansion of π, so we don't know *which* of those functions is f. Nevertheless, we know that the function f must be computable.)
- Each finite segment of an *in*computable sequence of natural numbers (such as the Busy Beaver function Σ) is computable. E.g., for each natural number n, there exists an algorithm that computes the finite sequence $\Sigma(0), \Sigma(1), \Sigma(2), ..., \Sigma(n)$ — in contrast to the fact that there is no algorithm that computes the *entire* Σ-sequence, i.e. $\Sigma(n)$ for all n. Thus, "Print 0, 1, 4, 6, 13" is a trivial algorithm to compute $\Sigma(0), \Sigma(1), \Sigma(2), \Sigma(3), \Sigma(4)$; similarly, for any given value of n, such a trivial algorithm *exists* (even though it may never be *known* or produced by anyone) to compute $\Sigma(0), \Sigma(1), \Sigma(2), ..., \Sigma(n)$.

Church-Turing thesis

The **Church-Turing thesis** states that any function computable from a procedure possessing the three properties listed above is a computable function. Because these three properties are not formally stated, the Church-Turing thesis cannot be proved. The following facts are often taken as evidence for the thesis:

- Many equivalent models of computation are known, and they all give the same definition of computable function (or a weaker version, in some instances).
- No stronger model of computation which is generally considered to be effectively calculable has been proposed.

The Church-Turing thesis is sometimes used in proofs to justify that a particular function is computable by giving a concrete description of a procedure for the computation. This is permitted because it is believed that all such uses of the thesis can be removed by the tedious process of writing a formal procedure for the function in some model of computation.

Incomputable functions and unsolvable problems

Every computable function has a finite procedure giving explicit, unambiguous instructions on how to compute it. Furthermore, this procedure has to be encoded in the finite alphabet used by the computational model, so there are only countably many computable functions. For example, functions may be encoded using a string of bits (the alphabet $\Sigma = \{0, 1\}$).

The real numbers are uncountable so most real numbers are not computable. The set of finitary functions on the natural numbers is uncountable so most are not computable. The Busy beaver function is a concrete example of such a function.

Similarly, most subsets of the natural numbers are not computable. The Halting problem was the first such set to be constructed. The Entscheidungsproblem, proposed by David Hilbert, asked whether there is an effective procedure to determine which mathematical statements (coded as natural numbers) are true. Turing and Church independently showed in the 1930s that this set of natural numbers is not computable. According to the Church-Turing thesis, there is no effective procedure (with an algorithm) which can perform these computations.

Extensions of computability

Relative Computability

The notion of computability of a function can be relativized to an arbitrary set of natural numbers A. A function f is defined to be **computable in A** (equivalently **A-computable** or **computable relative to A**) when it satisfies the definition of a computable function with modifications allowing access to A as an oracle. As with the concept of a computable function relative computability can be given equivalent definitions in many different models of computation. This is commonly accomplished by supplementing the model of computation with an additional primitive operation which asks whether a given integer is a member of A. We can also talk about f being **computable in g** by identifying g with its graph.

Higher Recursion Theory

Hyperarithmetical theory studies those sets that can be computed from a computable ordinal number of iterates of the Turing jump of the empty set. This is equivalent to sets defined by both a universal and existential formula in the language of second order arithmetic and to some models of Hypercomputation. Even more general recursion

theories have been studied, such as **E-recursion theory** in which any set can be used as an argument to an E-recursive function.

Hyper-computation

Although the Church-Turing thesis states that the computable functions include all functions with algorithms, it is possible to consider broader classes of functions that relax the requirements that algorithms must possess. The field of Hypercomputation studies models of computation that go beyond normal Turing computation. These don't violate the Church-Turing thesis since they allow operations that, whether or not they can be implemented in a physical device, couldn't be performed by a human working with pencil and paper.

Computability logic

Introduced by Giorgi Japaridze in 2003, **computability logic** is a research programme and mathematical framework for redeveloping logic as a systematic formal theory of computability, as opposed to classical logic which is a formal theory of truth. In this approach logical formulas represent computational problems (or, equivalently, computational resources), and their validity means being "always computable".

Computational problems and resources are understood in their most general - interactive sense. They are formalized as games played by a machine against its environment, and computability means existence of a machine that wins the game against any possible behavior by the environment. Defining what such game-playing machines mean, computability logic provides a generalization of the Church-Turing thesis to the interactive level.

The classical concept of truth turns out to be a special, zero-interactivity-degree case of computability. This makes classical logic a special fragment of computability logic. Being a conservative extension of the former, computability logic is, at the same time, by an order of magnitude more expressive, constructive and computationally meaningful. Providing a systematic answer to the fundamental question "what (and how) can be computed?", it has a wide range of potential application areas. Those include constructive applied theories, knowledge base systems, systems for planning and action.

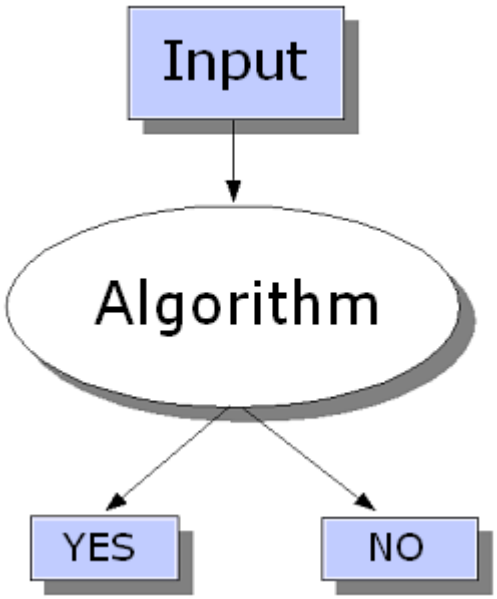
Besides classical logic, linear logic (understood in a relaxed sense) and intuitionistic logic also turn out to be natural fragments of computability logic. Hence meaningful concepts of "intuitionistic truth" and "linear-logic truth" can be derived from the semantics of computability logic.

Being semantically constructed, as yet computability logic does not have a fully developed proof theory. Finding deductive systems for various fragments of it and exploring their syntactic properties is an area of ongoing research.

Chapter 7

Decision Problem and Enumeration

Decision problem



A **decision problem** has only two possible outputs, *yes* or *no* (or alternately 1 or 0) on any input.

In computability theory and computational complexity theory, a **decision problem** is a question in some formal system with a yes-or-no answer, depending on the values of some input parameters. For example, the problem "given two numbers *x* and *y*, does *x* evenly divide *y*?" is a decision problem. The answer can be either 'yes' or 'no', and depends upon the values of *x* and *y*.

Decision problems are closely related to function problems, which can have answers that are more complex than a simple 'yes' or 'no'. A corresponding function problem is "given two numbers *x* and *y*, what is *x* divided by *y*?". They are also related to optimization problems, which are concerned with finding the *best* answer to a particular problem.

A method for solving a decision problem given in the form of an algorithm is called a **decision procedure** for that problem. A decision procedure for the decision problem "given two numbers *x* and *y*, does *x* evenly divide *y*?" would give the steps for determining whether *x* evenly divides *y*, given *x* and *y*. One such algorithm is long division, taught to many school children. If the remainder is zero the answer produced is 'yes', otherwise it is 'no'. A decision problem which can be solved by an algorithm, such as this example, is called **decidable**.

The field of computational complexity categorizes *decidable* decision problems by how difficult they are to solve. "Difficult", in this sense, is described in terms of the computational resources needed by the most efficient algorithm for a certain problem. The field of recursion theory, meanwhile, categorizes *undecidable* decision problems by Turing degree, which is a measure of the noncomputability inherent in any solution.

Research in computability theory has typically focused on decision problems. As explained in the section Equivalence with function problems below, there is no loss of generality.

Definition

A *decision problem* is any arbitrary yes-or-no question on an infinite set of inputs. Because of this, it is traditional to define the decision problem equivalently as: the set of inputs for which the problem returns *yes*.

These inputs can be natural numbers, but also other values of some other kind, such as strings of a formal language. Using some encoding, such as Gödel numberings, the strings can be encoded as natural numbers. Thus, a decision problem informally phrased in terms of a formal language is also equivalent to a set of natural numbers. To keep the formal definition simple, it is phrased in terms of subsets of the natural numbers.

Formally, a **decision problem** is a subset of the natural numbers. The corresponding informal problem is that of deciding whether a given number is in the set.

Examples

A classic example of a decidable decision problem is the set of prime numbers. It is possible to effectively decide whether a given natural number is prime by testing every possible nontrivial factor. Although much more efficient methods of primality testing are known, the existence of any effective method is enough to establish decidability.

Decidability

A decision problem *A* is called **decidable** or **effectively solvable** if *A* is a recursive set. A problem is called **partially decidable**, **semidecidable**, **solvable**, or **provable** if *A* is a recursively enumerable set. Partially decidable problems and any other problems that are not decidable are called **undecidable**.

Complete problems

Decision problems can be ordered according to many-one reducibility and related feasible reductions such as Polynomial-time reductions. A decision problem P is said to be **complete** for a set of decision problems S if P is a member of S and every problem in S can be reduced to P. Complete decision problems are used in computational complexity to characterize complexity classes of decision problems. For example, the Boolean satisfiability problem is complete for the class NP of decision problems under polynomial-time reducibility.

History

The *Entscheidungsproblem*, German for "Decision-problem", is attributed to David Hilbert: "At [the] 1928 conference Hilbert made his questions quite precise. First, was mathematics *complete*... Second, was mathematics *consistent*... And thirdly, was mathematics *decidable*? By this he meant, did there exist a definite method which could, in principle be applied to any assertion, and which was guaranteed to produce a correct decision on whether that assertion was true" (Hodges, p. 91). Hilbert believed that "in mathematics there is no ignorabimus' (Hodges, p. 91ff) meaning 'we will not know'.

Equivalence with function problems

A function problem consists of a partial function f; the informal "problem" is to compute the values of f on the inputs for which it is defined.

Every function problem can be turned into a decision problem; the decision problem is just the graph of the associated function. (The graph of a function f is the set of pairs (x,y) such that $f(x) = y$.) If this decision problem were effectively solvable then the function problem would be as well. This reduction does not respect computational complexity, however. For example, it is possible for the graph of a function to be decidable in polynomial time (in which case running time is computed as a function of the pair (x,y)) when the function is not computable in polynomial time (in which case running time is computed as a function of x alone). The function $f(x) = 2^x$ has this property.

Every decision problem can be converted into the function problem of computing the characteristic function of the set associated to the decision problem. If this function is computable then the associated decision problem is decidable. However, this reduction is more liberal than the standard reduction used in computational complexity (sometimes called polynomial-time many-one reduction); for example, the complexity of the characteristic functions of an NP-complete problem and its co-NP-complete complement is exactly the same even though the underlying decision problems may not be considered equivalent in some typical models of computation.

Practical decision

Having practical decision procedures for classes of logical formulas is of considerable interest for program verification and circuit verification. Pure Boolean logical formulas are usually decided using SAT-solving techniques based on the DPLL algorithm. Conjunctive formulas over linear real or rational arithmetic can be decided using the Simplex algorithm, formulas in linear integer arithmetic (Presburger arithmetic) can be decided using Cooper's algorithm or William Pugh's Omega test. Formulas with negations, conjunctions and disjunctions combine the difficulties of satisfiability testing with that of decision of conjunctions; they are generally decided nowadays using SMT-solving technique, which combine SAT-solving with decision procedures for conjunctions and propagation techniques. Real polynomial arithmetic, also known as the theory of real closed fields, is decidable, for instance using the Cylindrical algebraic decomposition; unfortunately the complexity of that algorithm is excessive for most practical uses.

Enumeration

In mathematics and theoretical computer science, the broadest and most abstract definition of an **enumeration** of a set is an exact listing of all of its elements (perhaps with repetition). The restrictions imposed on the type of list used depend on the branch of mathematics and the context in which one is working. In more specific settings, this notion of enumeration encompasses the two different types of listing: one where there is a natural ordering and one where the ordering is more nebulous. These two different kinds of enumerations correspond to a procedure for listing all members of the set in some definite sequence, or a count of objects of a specified kind, respectively. While the two kinds of enumeration often overlap in most natural situations, they can assume very different meanings in certain contexts.

Enumeration as counting

Formally, the most inclusive definition of an enumeration of a set S is any surjection from an arbitrary index set I onto S. In this broad context, every set S can be trivially enumerated by the identity function from S onto itself. If one does *not* assume the axiom of choice or one of its variants, S need not have any well-ordering. Even if one does assume the axiom of choice, S need not have any natural well-ordering.

This general definition therefore lends itself to a counting notion where we are interested in "how many" rather than "in what order." In practice, this broad meaning of enumeration is often used to compare the relative sizes or cardinalities of different sets. If one works in Zermelo-Fraenkel set theory without the axiom of choice, one may want to impose the additional restriction that an enumeration must also be injective (without

repetition) since in this theory, the existence of a surjection from *I* onto *S* need not imply the existence of an injection from *S* into *I*.

Enumeration as listing

When an enumeration is used in an ordered list context, we impose some sort of ordering structure requirement on the index set. While we can make the requirements on the ordering quite lax in order to allow for great generality, the most natural and common prerequisite is that the index set be well-ordered. According to this characterization, an ordered enumeration is defined to be a surjection with a well-ordered domain. This definition is natural in the sense that a given well-ordering on the index set provides a unique way to list the next element given a partial enumeration.

Enumeration in countable vs. uncountable context

The most common use of enumeration occurs in the context where infinite sets are separated into those that are countable and those that are not. In this case, an enumeration is merely an enumeration with domain ω. This definition can also be stated as follows:

- As a surjective mapping from $\mathbb{N}$(the natural numbers) to *S* (i.e., every element of *S* is the image of at least one natural number). This definition is especially suitable to questions of computability and elementary set theory.

We may also define it differently when working with finite sets. In this case an enumeration may be defined as follows:

- As a bijective mapping from *S* to an initial segment of the natural numbers. This definition is especially suitable to combinatorial questions and finite sets; then the initial segment is {1,2,...,*n*} for some *n* which is the cardinality of *S*.

In the first definition it varies whether the mapping is also required to be injective (i.e., every element of *S* is the image of *exactly one* natural number), and/or allowed to be partial (i.e., the mapping is defined only for some natural numbers). In some applications (especially those concerned with computability of the set *S*), these differences are of little importance, because one is concerned only with the mere existence of some enumeration, and an enumeration according to a liberal definition will generally imply that enumerations satisfying stricter requirements also exist.

Enumeration of finite sets obviously requires that either non-injectivity or partiality is accepted, and in contexts where finite sets may appear one or both of these are inevitably present.

Examples

- The natural numbers are enumerable by the function f(x) = x. In this case $f : \mathbb{N} \to \mathbb{N}$is simply the identity function.

- $\mathbb{Z}$, the set of integers is enumerable by

$$f(x) := \begin{cases} -(x+1)/2, & \text{if } x \text{ is odd} \\ x/2, & \text{if } x \text{ is even.} \end{cases}$$

$f : \mathbb{N} \to \mathbb{Z}$is a bijection since every natural number corresponds to exactly one integer. The following table gives the first few values of this enumeration:

x	0	1	2	3	4	5	6	7	8
$f(x)$	0	−1	1	−2	2	−3	3	−4	4

- All finite sets are enumerable. Let S be a finite set with n elements and let $K = \{1,2,...,n\}$. Select any element s in S and assign $f(n) = s$. Now set $S' = S - \{s\}$ (where − denotes set difference). Select any element $s' \in S'$ and assign $f(n-1) = s$. Continue this process until all elements of the set have been assigned a natural number. Then $f : \{1, 2, \ldots, n\} \to S$is an enumeration of S.

- The real numbers have no countable enumeration as proved by Cantor's diagonalization argument.

Properties

- There exists an enumeration for a set (in this sense) if and only if the set is countable.

- If a set is enumerable it will have an uncountable infinity of different enumerations, except in the degenerate cases of the empty set or (depending on the precise definition) sets with one element. However, if one requires enumerations to be injective *and* allows only a limited form of partiality such that if $f(n)$ is defined then $f(m)$ must be defined for all $m < n$, then a finite set of N elements has exactly $N!$ enumerations.

- An enumeration e of a set S with domain $\mathbb{N}$induces a well-order $\leq$ on that set defined by $s \leq t$ if and only if $\min e^{-1}(s) \leq \min e^{-1}(t)$. Although the order may have little to do with the underlying set, it is useful when some order of the set is necessary.

Computable enumeration

In computability theory one often considers countable enumerations with the added requirement that the mapping from $\mathbb{N}$to the enumerated set must be computable. The set being enumerated is then called recursively enumerable (or computably enumerable in

more contemporary language), referring to the use of recursion theory in formalizations of what it means for the map to be computable.

In this sense, a subset of Natural numbers is computably enumerable if it is the range of a computable function. In this context, enumerable may be used to mean computably enumerable. However, these definitions characterize distinct classes since there are uncountably many subsets of Natural numbers that can be enumerated by an arbitrary function with domain ω and only countably many computable functions. A specific example of a set with an enumeration but not a computable enumeration is the complement of the halting set.

Furthermore, this characterization illustrates a place where the ordering of the listing is important. There exists a computable enumeration of the halting set, but **not** one that lists the elements in an increasing ordering. If there were one, then the halting set would be decidable, which is provably false. In general, being recursively enumerable is a weaker condition than being a decidable set.

Ordinal enumeration

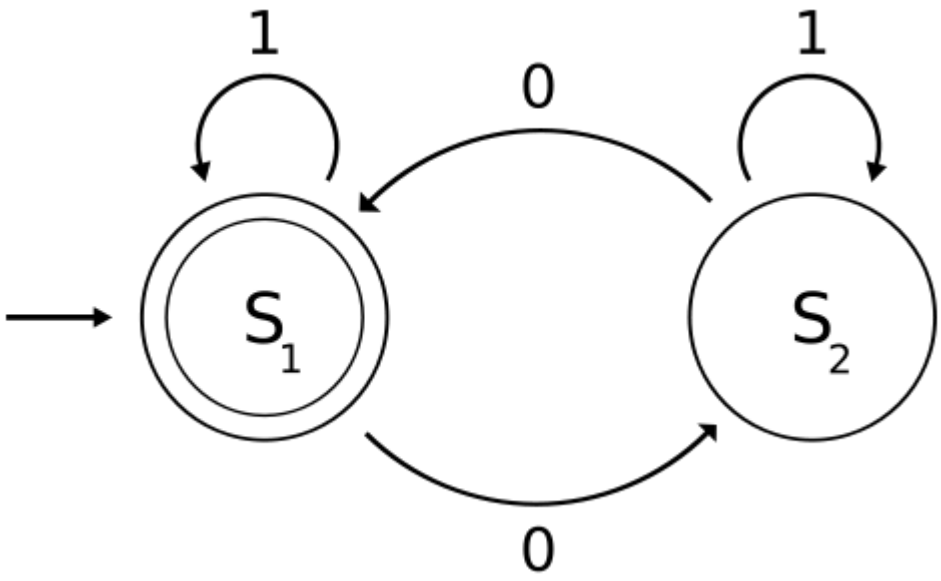
In set theory, there is a more general notion of an enumeration than the characterization requiring the domain of the listing function to be an initial segment of the Natural numbers where the domain of the enumerating function can assume any ordinal. Under this definition, an enumeration of a set *S* is any surjection from an ordinal α onto *S*. The more restrictive version of enumeration mentioned before is the special case where α is a finite ordinal or the first limit ordinal ω. This more generalized version extends the aforementioned definition to encompass transfinite listings.

Under this definition, the first uncountable ordinal ω_1 can be enumerated by the identity function on ω_1 so that these two notions do **not** coincide. More generally, it is a theorem of ZF that any well-ordered set can be enumerated under this characterization so that it coincides up to relabeling with the generalized listing enumeration. If one also assumes the Axiom of Choice, then all sets can be enumerated so that it coincides up to relabeling with the most general form of enumerations.

Since set theorists work with infinite sets of arbitrarily large cardinalities, the default definition among this group of mathematicians of an enumeration of a set tends to be any arbitrary α-sequence exactly listing all of its elements. Indeed, in Jech's book, which is a common reference for set theorists, an enumeration is defined to be exactly this. Therefore, in order to avoid ambiguity, one may use the term finitely enumerable or denumerable to denote one of the corresponding types of distinguished countable enumerations.

Chapter 8

Automata Theory



An example of automata and study of mathematical properties of such automata is automata theory

In theoretical computer science, **automata theory** is the study of abstract machines (or more appropriately, abstract 'mathematical' machines or systems) and the computational problems that can be solved using these machines. These abstract machines are called automata.

The figure at right illustrates a finite state machine, which is one well-known variety of automaton. This automaton consists of states (represented in the figure by circles), and transitions (represented by arrows). As the automaton sees a symbol of input, it makes a *transition* (or *jump*) to another state, according to its *transition function* (which takes the current state and the recent symbol as its inputs).

Automata theory is also closely related to formal language theory, as the automata are often classified by the class of formal languages they are able to recognize. An automaton can be a finite representation of a formal language that may be an infinite set.

Automata play a major role in compiler design and parsing.

Automata

Following is an introductory definition of one type of automata, which attempts to help one grasp the essential concepts involved in automata theory.

Informal description

An automaton is supposed to *run* on some given sequence or string of *inputs* in discrete time steps. At each time step, an automaton gets one input that is picked up from a set of *symbols* or *letters*, which is called an *alphabet*. At any time, the symbols so far fed to the automaton as input form a finite sequence of symbols, which is called a *word*. An automaton contains a finite set of states. At each instance in time of some run, automaton is *in* one of its states. At each time step when the automaton reads a symbol, it *jumps* or *transits* to next state depending on its current state and on the symbol currently read. This function in terms of the current state and input symbol is called *transition function*. The automaton *reads* the input word one symbol after another in the sequence and transits from state to state according to the transition function, until the word is read completely. Once the input word has been read, the automaton is said to have been *stopped* and the state at which automaton has stopped is called *final state*. Depending on the final state, it's said that the automaton either *accepts* or *rejects* an input word. There is a subset of states of the automaton, which is defined as the set of *accepting states*. If the final state is an accepting state, then the automaton *accepts* the word. Otherwise, the word is *rejected*. The set of all the words accepted by an automaton is called the *language recognized by the automaton*.

Formal definition

Automaton

An **automaton** is represented formally by the 5-tuple $\mathbf{(Q,\Sigma,\delta,q_0,A)}$, where:

- Q is a finite set of *states*.
- Σ is a finite set of *symbols*, called the *alphabet* of the automaton.
- δ is the **transition function**, that is, $\delta: Q \times \Sigma \rightarrow Q$.
- q_0 is the *start state*, that is, the state which the automaton is *in* when no input has been processed yet, where $q_0 \in Q$.
- A is a set of states of Q (i.e. $A \subseteq Q$) called **accept states**.

Input word

An automaton reads a finite string of symbols $a_1,a_2,....,a_n$, where $a_i \in \Sigma$, which is called a *input word*. Set of all words is denoted by Σ^*.

Run

A *run* of the automaton on an input word $w = a_1,a_2,....,a_n \in \Sigma^*$, is a sequence of states $q_0,q_1,q_2,....,q_n$, where $q_i \in Q$ such that q_0 is the start state and $q_i = \delta(q_{i-1},a_i)$ for $0 < i \leq n$. In words, at first the automaton is at the start state q_0 and then

automaton reads symbols of the input word in sequence. When automaton reads symbol a_i then it jumps to state $q_i = \delta(q_{i-1},a_i)$. q_n said to be the *final state* of the run.

Accepting word

A word $w \in \Sigma^*$ is accepted by the automaton if $q_n \in A$.

Recognized language

An automaton can recognize a formal language. The recognized language $L \subset \Sigma^*$ by an automaton is the set of all the words that are accepted by the automaton.

Recognizable languages

The recognizable languages is the set of languages that are recognized by some automaton. For above definition of automata the recognizable languages are regular languages. For different definitions of automata, the recognizable languages are different.

Variations in definition of automata

Automata are defined to study useful machines under mathematical formalism. So, the definition of an automaton is open to variations according to the "real world machine", which we want to model using the automaton. People have studied many variations of automata. Above, the most standard variant is described, which is called deterministic finite automaton. The following are some popular variations in the definition of different components of automata.

Input

- *Finite input*: An automaton that accepts only finite sequence of symbols. The above introductory definition only accepts finite words.
- *Infinite input*: An automaton that accepts infinite words (ω-words). Such automata are called *ω-automata*.
- *Tree word input*: The input may be a *tree of symbols* instead of sequence of symbols. In this case after reading each symbol, the automaton *reads* all the successor symbols in the input tree. It is said that the automaton *makes one copy* of itself for each successor and each such copy starts running on one of the successor symbol from the state according to the transition relation of the automaton. Such an automaton is called tree automaton.

States

- *Finite states*: An automaton that contains only a finite number of states. The above introductory definition describes automata with finite numbers of states.
- *Infinite states*: An automaton that may not have a finite number of states, or even a countable number of states. For example, the quantum finite automaton or topological automaton has uncountable infinity of states.
- *Stack memory*: An automaton may also contain some extra memory in the form of a stack in which symbols can be pushed and popped. This kind of automaton is called a *pushdown automaton*

Transition function

- *Deterministic*: For a given current state and an input symbol, if an automaton can only jump to one and only one state then it is a *deterministic automaton*.
- *Nondeterministic*: An automaton that, after reading an input symbol, may jump into any of a number of states, as licensed by its transition relation. Notice that the term transition function is replaced by transition relation: The automaton *non-deterministically* decides to jump into one of the allowed choices. Such automaton are called *nondeterministic automaton*.
- *Alternation*: This idea is quite similar to tree automaton, but orthogonal. The automaton may run its *multiple copies* on the *same* next read symbol. Such automata are called *alternating automaton*. Acceptance condition must satisfy all runs of such *copies* to accept the input.

Acceptance condition

- *Acceptance of finite words*: Same as described in the informal definition above.
- *Acceptance of infinite words*: an *omega automaton* cannot have final states, as infinite words never terminate. Rather, acceptance of the word is decided by looking at the infinite sequence of visited states during the run.
- *Probabilistic acceptance*: An automaton need not strictly accept or reject an input. It may accept the input with some probability between zero and one. For example, quantum finite automaton, geometric automaton and *metric automaton* has probabilistic acceptance.

Different combinations of the above variations produce many variety of automaton.

Automata theory

Automata theory is a subject matter which studies properties of various types of automata. For example, following questions are studied about a given type of automata.

- Which class of formal languages is recognizable by some type of automata? (Recognizable languages)
- Is certain automata *closed* under union, intersection, or complementation of formal languages? (Closure properties)
- How much is a type of automata expressive in terms of recognizing class of formal languages? And, their relative expressive power? (Language Hierarchy)

Automata theory also studies if there exist any effective algorithm or not to solve problems similar to following list.

- Does an automaton accept any input word? (emptiness checking)
- Is it possible to transform a given non-deterministic automaton into deterministic automaton without changing the recognizing language? (Determinization)

- For a given formal language, what is the smallest automaton that recognizes it? (Minimization).

Classes of automata

Automata	Recognizable language
Deterministic finite automata (DFA)	regular languages
Nondeterministic finite automata (NFA)	regular languages
Nondeterministic finite automata with ε-transitions (FND-ε or ε-NFA)	regular languages
Pushdown automata (PDA)	context-free languages
Linear bounded automata (LBA)	context-sensitive language
Turing machines	recursively enumerable languages
Timed automata	
Deterministic Büchi automata	ω-limit languages
Nondeterministic Büchi automata	ω-regular languages
Nondeterministic/Deterministic Rabin automata	ω-regular languages
Nondeterministic/Deterministic Streett automata	ω-regular languages
Nondeterministic/Deterministic parity automata	ω-regular languages
Nondeterministic/Deterministic Muller automata	ω-regular languages

Discrete, continuous, and hybrid automata

Normally automata theory describes the states of abstract machines but there are analog automata or continuous automata or hybrid discrete-continuous automata, using analog data, continuous time, or both.

Applications

Each model in automata theory play varied roles in several applied areas. Finite automata is used in text processing, compilers, and hardware design. Context-free grammar is used in programming languages and artificial intelligence. Originally, CFG were used in the study of the human languages. Cellular automata is used in the field of biology, the most common example being John Conway's Game of Life. Some other examples which could be explained using automata theory in biology include mollusk and pine cones growth and pigmentation patterns. Going further, a theory suggesting that the whole universe is computed by some sort of a discrete automaton, is being advocated by some scientist. The idea originated in the work of Konrad Zuse, most importantly his 1969 book Rechnender Raum and gave rise to Digital physics.

Chapter 9

Powerset Construction

In the theory of computation and Automata theory, the **powerset construction** or **subset construction** is a standard method for converting a nondeterministic finite automaton (NFA) into a deterministic finite automaton (DFA) which recognizes the same formal language. It is important in theory because it establishes that NFAs, despite their additional flexibility, are unable to recognize any language that cannot be recognized by some DFA. It is also important in practice for converting easier-to-construct NFAs into more efficiently executable DFAs. However, if the NFA has n states, the resulting DFA can have up to 2^n states, exponentially more, which sometimes makes the construction impractical for large NFAs.

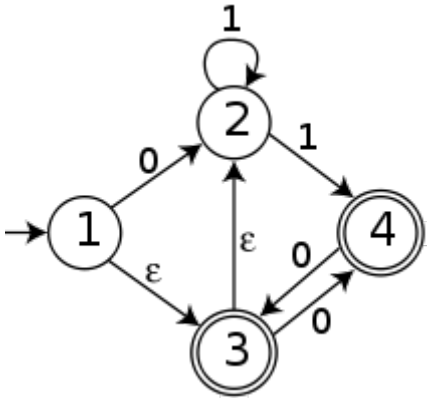
Motivation

Recall that an NFA is like a DFA except that at certain nodes, it can "branch", following a number of paths forward simultaneously. The NFA will accept if one of its paths ends up at an accepting state when the computation completes. If all of its paths fail, it will reject. For example, in the diagram in the example section, if we are at state 2 and the next input symbol is a 1, the machine branches, proceeding to both states 2 and 4.

Notice that no matter how many different paths the NFA might be following, each of them must be in one of the n states. For this reason, we can succinctly sum up the current configuration of the NFA as the set of states it could be in at that moment, given some previous sequence of choices. Moreover, if we know the set of states the NFA is currently in, we can figure out what set of states it will visit next based on the next input token. This is the key to the algorithm.

Example

Consider the following NFA with alphabet {0, 1}:

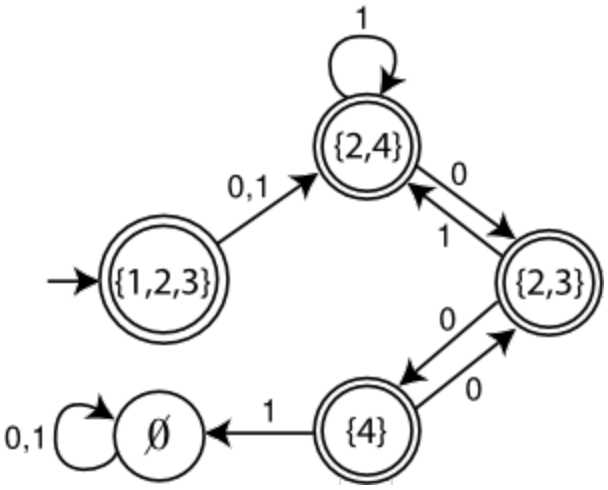


We will construct an equivalent DFA; the final result is shown below. We begin with the start state. The NFA starts in state 1, but it can also, without seeing any input, proceed to states 3 and 2 along ε edges. Therefore we must consider it to be in all these states simultaneously initially. We do this by creating a start state for our DFA and labelling it "{1,2,3}".

Next, suppose we see the input character 0. If we stayed in state 1, we can follow the edge labelled "0" to state 2. If we went to state 3, we can proceed to state 4. If we went to state 2, however, we'd be stuck; there's no "0" edge out of state 2. This means the NFA forgets about the old state 2 path; it is now in states 2 and 4. In the DFA, we express this with a new state labelled "{2,4}" and an edge labelled "0" from the start state to the new state.

Suppose we saw a 1 instead initially. Then both the path in state 1 and the path in state 3 cannot proceed, but the path in state 2 can, and it branches: it both remains in state 2 and proceeds to state 4. Thus, the NFA is now in states 2 and 4 — exactly the same as for the 0 input, but for a different reason. We represent this by adding the label "1" to our edge from the start state to the existing "{2,4}".

Now, suppose we're in the "{2,4}" state and we see a 1. The path in state 4 cannot proceed, but the path in state 2 goes to both 2 or 4, again. We remain in state "{2,4}". If we see a 0, however, we can't proceed from state 2, but we can proceed from state 4 to state 3. Moreover, after we reach state 3, we branch and follow the ε-edge to state 2 as well. The result is that we are in states 2 and 3. We represent this with a new node in the DFA labelled "{2,3}" and an edge labelled "0" from {2,4} to {2,3}.



The final result DFA

If we see a 1 in the state {2,3}, the path in state 3 cannot proceed, but the path in state 2 goes to states 2 and 4, as before, so we return to the node {2,4}. If we see a 0 however, the path in state 2 cannot proceed, while the path in state 3 can reach state 4. Thus, we could only be in state 4. We create a new node labelled "{4}" and an edge labelled "0" from {2,3} to {4}.

Finally, if we're in state {4} and we see a 0 as input, we proceed to states 2 and 3, as before, so there is an edge labelled "0" from {4} to {2,3}. If we see a 1, however, all of our remaining paths are stuck and the machine must reject. How do we ensure this? We create a new node, which we will label "∅", from which there is no escape; all its outgoing edges point to itself, and it is not an accepting state. We then add an edge from {4} to ∅ labelled 1.

We've now considered all possible cases. All we have to decide is which states in our DFA should accept. Since the NFA accepts if any of its paths end up in an accepting state, we can mimic this by setting all the DFA nodes that include an accepting NFA state, namely {1,2,3}, {2,3}, {2,4}, and {4}, as accepting. The resulting DFA accepts exactly the same set of strings as the original NFA. Note that the DFA is larger than the original NFA.

Defining the equivalent DFA

Let's generalize the procedure of the previous section. There are four important questions we must answer to define a DFA:

- What are the states?
- Which of these states are accepting states?
- What state is the start state?
- Where do we place edges and with what labels?

We need a state of the DFA to describe each possible configuration of the NFA. But, in general, the NFA might be at any subset of its states at any given point. The set of subsets of a set S is called its powerset, written $P(S)$, and so we define the set of states in the DFA to be the power set of the set of states in the NFA. This answers the first question.

We already mentioned that if any of the NFA's parallel paths is in an accepting state at the end, then the NFA accepts. The DFA can mimic this by accepting in any state that contains one of the NFA's accepting states. This answers the second question.

Now, for the third question. Suppose that the input string given to the NFA is the empty string. What states can it visit before it must stop? It can't follow any edges that are labelled with an input symbol, but it *can* follow ε edges, which consume no input. Thus, it can visit any state that is reachable from the start state using just ε edges. This set of states is formally called the ε-closure of the start state. Because our DFA can't do anything when given an empty input string except halt immediately, we must be sure its start state includes the possibility of all these states. We do this by setting its start state to the ε-closure of the NFA's start state.

Finally, we will answer the fourth question using similar ideas. Suppose we're in a particular state of the DFA (that is, a particular *set* of states in the NFA) and we see a particular input symbol. We want to know what DFA state to go to next. This will be precisely the set of NFA states this input symbol will allow us to visit from the current set of NFA states. To find this, we look at each one of the current NFA states and ask, 'Given this input symbol, where could it go next from here?' The answer is that it can follow any single edge labelled with that input symbol, as well as any number of ε edges. We search and find all nodes that we can reach in this manner, and add them to the set of nodes we can reach next. When this is done for all current NFA states, we'll have a set of NFA states corresponding to a particular DFA state, and we add an edge from the current DFA state to this state labelled with the input symbol.

Once we've followed this process for all DFA states and all symbols, our DFA will be complete. The resulting machines tracks what set of states the NFA is visiting at each moment in the input string. However, this machine is quite huge: since each set of NFA states may or may not contain any particular NFA state, there are 2^n total such sets, and a DFA node for each of them. If we proceed like we did in the example, however, only creating nodes when we know they are needed, we can often create a much smaller DFA. Nevertheless, there are still cases for which we will need all 2^n states; this is unavoidable.

Formal definition

Let $\mathbf{M} = (S, \Sigma, T, s, A)$ be a nondeterministic finite automaton (with epsilon moves).

Define the 5-tuple $\mathbf{M_d} = (S_d, \Sigma_d, T_d, s_d, A_d)$, where

- $S_d = P(S)$
- $\Sigma_d = \Sigma$

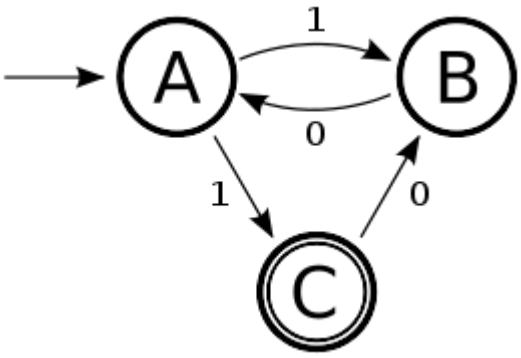
- $s_d = C_\varepsilon(s)$
- $T_d(q, a) = C_\varepsilon(\cup_{\forall r \in q} T(r, a)) \quad \forall q \in S_d, \forall a \in \Sigma$
- $A_d = \{q \mid q \in S_d \wedge q \cap A \neq \varnothing\}$

P(S) is the powerset of S;

$C_\varepsilon(q)$ is the ε-closure of q, i.e. the set of all states reachable from q by one or more ε-transitions.

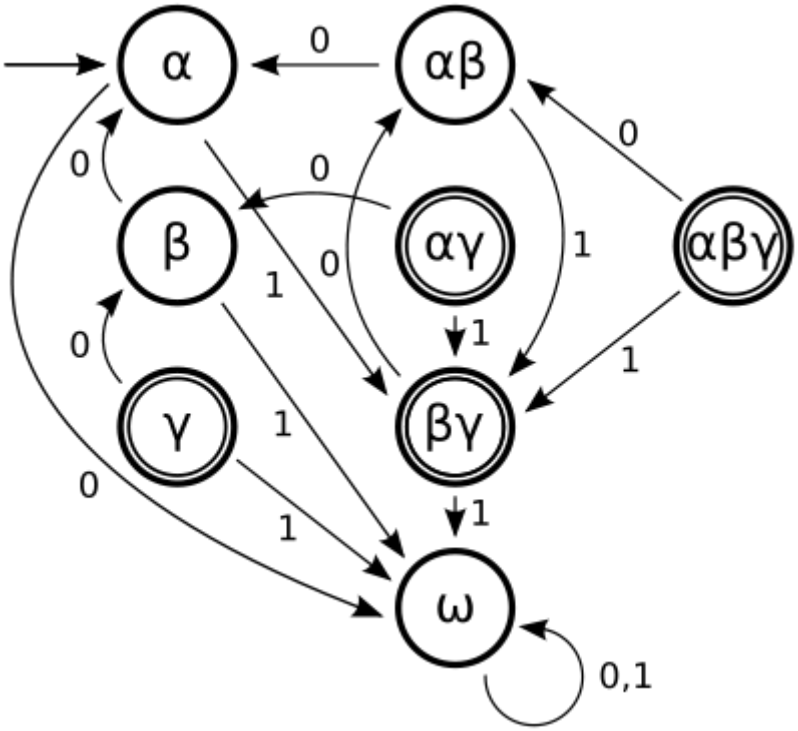
It can be proved mathematically that $\mathbf{M_d}$ is a deterministic finite automaton that accepts the same language as **M**.

Example 2



Consider the following NFA:

- S_n={A,B,C}
- $\sum_n$={0,1}
- s_n=A
- A_n={C}

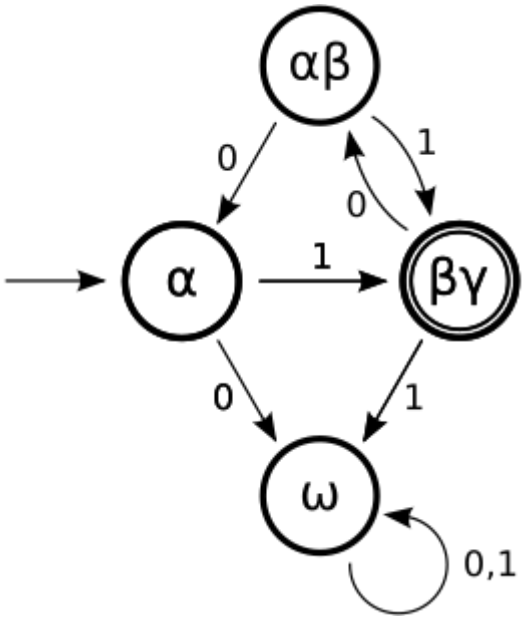


Equivalent DFA has $2^{|S_n|}=2^3=8$ states:

- $S_{d'}=\{\alpha,\beta,\gamma,\alpha\beta,\alpha\gamma,\beta\gamma,\alpha\beta\gamma,\omega\}$

  α corresponds to A, β corresponds to B, γ corresponds to C
  αβ can be reached, for example, when from A exist 2 edges labelled "0": A→A and A→B. $A \times 0 \rightarrow \{A,B\}$

- $\sum_{d'}=\{0,1\}$
- $s_{d'}=\alpha$
- $A_{d'}=\{\gamma,\alpha\gamma,\beta\gamma,\alpha\beta\gamma\}$

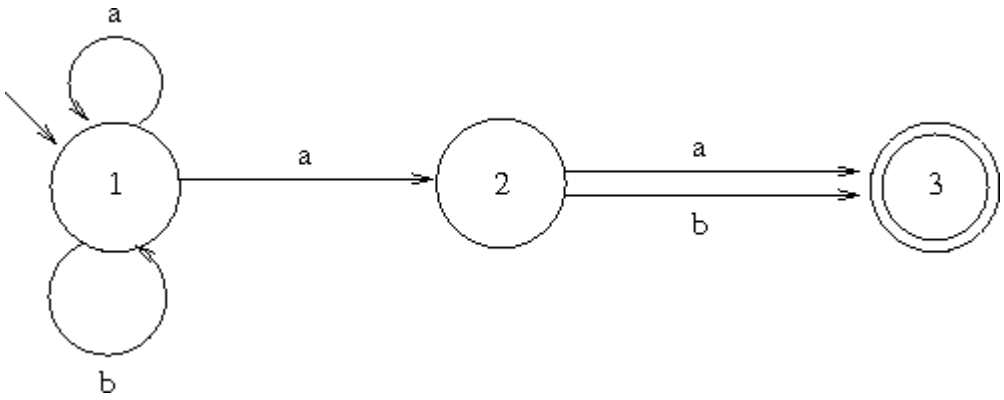


After removing states that cannot be reached from $s_{d'}$ we get final DFA:

- $S_d=\{\alpha,\alpha\beta,\beta\gamma,\omega\}$
- $\sum_d=\{0,1\}$
- $s_d=\alpha$
- $A_d=\{\beta\gamma\}$

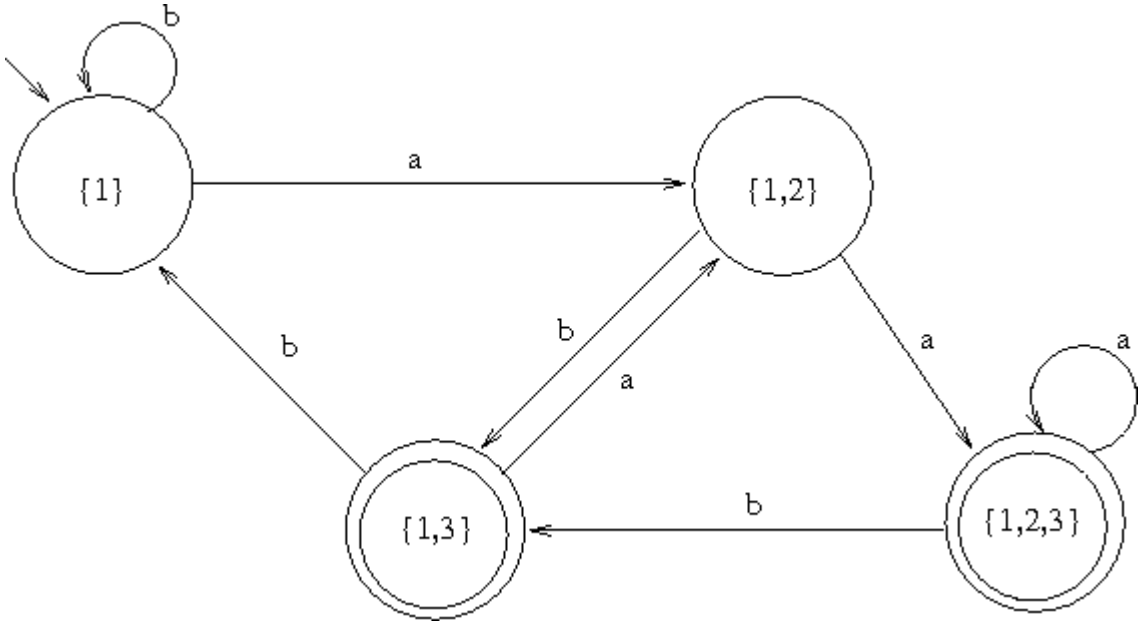
Example 3

Here is another example. Following is a NFA.



Another example NFA

Following is an equivalent DFA of above NFA.



Equivalent DFA

Chapter 10

Probabilistic Automaton and Deterministic Finite-state Machine

Probabilistic automaton

In mathematics and computer science, the **probabilistic automaton (PA)** is a generalization of the non-deterministic finite automaton; it includes the probability of a given transition into the transition function, turning it into a transition matrix or stochastic matrix. Thus, the probabilistic automaton generalizes the concept of a Markov chain or subshift of finite type. The languages recognized by probabilistic automata are called **stochastic languages**; these include the regular languages as a subset. The number of stochastic languages is uncountable.

The concept was introduced by Michael O. Rabin in 1963; a certain special case is sometimes known as the **Rabin automaton**. In recent years, a variant has been formulated in terms of quantum probabilities, the quantum finite automaton.

Definition

The probabilistic automaton may be defined as an extension of a non-deterministic finite automaton (Q,Σ,δ,q_0,F), together with two probabilities: the probability P of a particular state transition taking place, and with the initial state q_0 replaced by a stochastic vector giving the probability of the automaton being in a given initial state.

For the ordinary non-deterministic finite automaton, one has

- a finite set of states Q
- a finite set of input symbols Σ
- a transition function $\delta : Q \times \Sigma \to P(Q)$
- a set of states F distinguished as *accepting* (or *final*) *states* $F \subset Q$.

Here, $P(Q)$ denotes the power set of Q.

By use of currying, the transition function $\delta : Q \times \Sigma \to P(Q)$ of a non-deterministic finite automaton can be written as a membership function

$$\delta : Q \times \Sigma \times Q \to \{0,1\}$$

so that $\delta(q,a,q') = 1$ if $q' \in \delta(q,a)$ and $\delta(q,a,q') = 0$ if $q' \notin \delta(q,a)$. The curried transition function can be understood to be a matrix with matrix entries

$$[\theta_a]_{qq'} = \delta(q,a,q')$$

The matrix θ_a is then a square matrix, whose entries are zero or one, indicating whether a transition $q \xrightarrow{a} q'$ is allowed by the NFA. Such a transition matrix is always defined for a non-deterministic finite automaton.

The probabilistic automaton replaces this matrix by a stochastic matrix *P*, so that the probability of a transition is given by

$$[P_a]_{qq'}$$

A state change from some state to any state must occur with probability one, of course, and so one must have

$$\sum_{q'} [P_a]_{qq'} = 1$$

for all input letters *a* and internal states *q*. The initial state of a probabilistic automaton is given by a row vector *v*, whose components add to unity:

$$\sum_{q} [v]_q = 1$$

The transition matrix acts on the right, so that the state of the probabilistic automaton, after consuming the input string *abc*, would be

$$vP_aP_bP_c$$

In particular, the state of a probabilistic automaton is always a stochastic vector, since the product of any two stochastic matrices is a stochastic matrix, and the product of a stochastic vector and a stochastic matrix is again a stochastic vector. This vector is sometimes called the **distribution of states**, emphasizing that it is a discrete probability distribution.

Formally, the definition of a probabilistic automaton does not require the mechanics of the non-deterministic automaton, which may be dispensed with. Formally, a probabilistic

automaton *PA* is defined as the tuple (*Q*,Σ,*P*,*v*,*F*). A **Rabin automaton** is one for which the initial distribution *v* is a coordinate vector; that is, has zero for all but one entries, and the remaining entry being one.

Stochastic languages

The set of languages recognized by probabilistic automata are called **stochastic languages**. They include the regular languages as a subset.

Let $F = Q_{\mathrm{accept}} \subset Q$ be the set of "accepting" or "final" states of the automaton. By abuse of notation, Q_{accept} can also be understood to be the column vector that is the membership function for Q_{accept}; that is, it has a 1 at the places corresponding to elements in Q_{accept}, and a zero otherwise. This vector may be contracted with the internal state probability, to form a scalar. The language recognized by a specific automaton is then defined as

$$L_\eta = \{s \in \Sigma^* | vP_sQ_{\mathrm{accept}} > \eta\}$$

where Σ^* is the set of all strings in the alphabet Σ (so that * is the Kleene star). The language depends on the value of the **cut-point** η, normally taken to be in the range $0 \leq \eta < 1$.

A language is called **η-stochastic** if and only if there exists some PA that recognizes the language, for fixed η. A language is called **stochastic** if and only if there is some $0 \leq \eta < 1$ for which L_η is η-stochastic.

A cut-point is said to be an **isolated cut-point** if and only if there exists a $\delta > 0$ such that

$$|vP(s)Q_{\mathrm{accept}} - \eta| \geq \delta$$

for all $s \in \Sigma^*$

Properties

Every regular language is stochastic, and more strongly, every regular language is η-stochastic. A weak converse is that every 0-stochastic language is regular; however, the general converse does not hold: there are stochastic languages that are not regular.

Every η-stochastic language is stochastic, for some $0 < \eta < 1$.

Every stochastic language is representable by a Rabin automaton.

If η is an isolated cut-point, then L_η is a regular language.

p-*adic languages*

The *p*-adic languages provide an example of a stochastic language that is not regular, and also show that the number of stochastic languages is uncountable. A *p*-adic language is defined as the set of strings in the letters $0, 1, 2, \ldots, (p-1)$ such that

$$L_\eta(p) = \{0.n_1n_2n_3\ldots | 0 \leq n_k < p \text{ and } 0.n_1n_2n_3\ldots > \eta\}$$

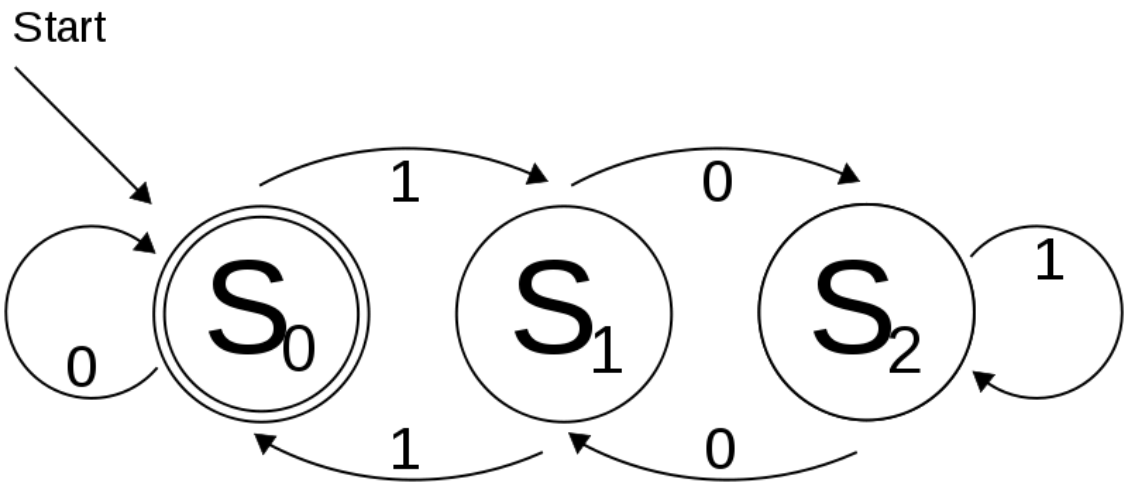
That is, a *p*-adic language is merely the set of real numbers, written in base-*p*, such that they are greater than η. It is straightforward to show that all *p*-adic languages are stochastic. However, a *p*-adic language is regular if and only if η is rational. In particular, this implies that the number of stochastic languages is uncountable.

Generalizations

The probabilistic automaton has a geometric interpretation: the state vector can be understood to be a point that lives on the face of the standard simplex, opposite to the orthogonal corner. The transition matrices form a monoid, acting on the point. This may be generalized by having the point be from some general topological space, while the transition matrices are chosen from a collection of operators acting on the topological space, thus forming a semiautomaton. When the cut-point is suitably generalized, one has a topological automaton.

An example of such a generalization is the quantum finite automaton; here, the automaton state is represented by a point in complex projective space, while the transition matrices are a fixed set chosen from the unitary group. The cut-point is understood as a limit on the maximum value of the quantum angle.

Deterministic finite-state machine



An example of a Deterministic Finite Automaton that accepts only binary numbers that are multiples of 3. The state S_0 is both the start state and an accept state.

In the theory of computation and automata theory, a **deterministic finite state machine**—also known as **deterministic finite automaton** (**DFA**)—is a finite state machine accepting finite strings of symbols. For each state, there is a transition arrow leading out to a next state for each symbol. Upon reading a symbol, a DFA jumps *deterministically* from a state to another by following the transition arrow. Deterministic means that there is only one outcome (i.e. move to next state when the symbol matches (S0 -> S1) or move back to the same state (S0 -> S0)). A DFA has a start state (denoted graphically by an arrow coming in from nowhere) where computations begin, and a set of accept states (denoted graphically by a double circle) which help define when a computation is successful.

DFAs recognize exactly the set of regular languages which are, among other things, useful for doing lexical analysis and pattern matching. A DFA can be used in either an accepting mode to verify that an input string is indeed part of the language it represents, or a generating mode to create a list of all the strings in the language.

DFA is defined as an abstract mathematical concept, but due to the deterministic nature of DFA, it is implementable in hardware and software for solving various specific problems. For example, a software state machine that decides whether or not online user-input such as phone numbers and email addresses are valid. Another example in hardware is the digital logic circuitry that controls whether an automatic door is open or closed, using input from motion sensors or pressure pads to decide whether or not to perform a state transition.

DFAs can be built from nondeterministic finite-state machines through the powerset construction.

Formal definition

A **deterministic finite automaton** M is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, consisting of

- a finite set of states (Q)
- a finite set of input symbols called the alphabet (Σ)
- a transition function ($\delta : Q \times \Sigma \rightarrow Q$)
- a start state ($q_0 \in Q$)
- a set of accept states ($F \subseteq Q$)

Let $w = a_1a_2 \ldots a_n$ be a string over the alphabet Σ. The automaton M accepts the string w if a sequence of states, $r_0, r_1, \ldots, r_n$, exists in Q with the following conditions:

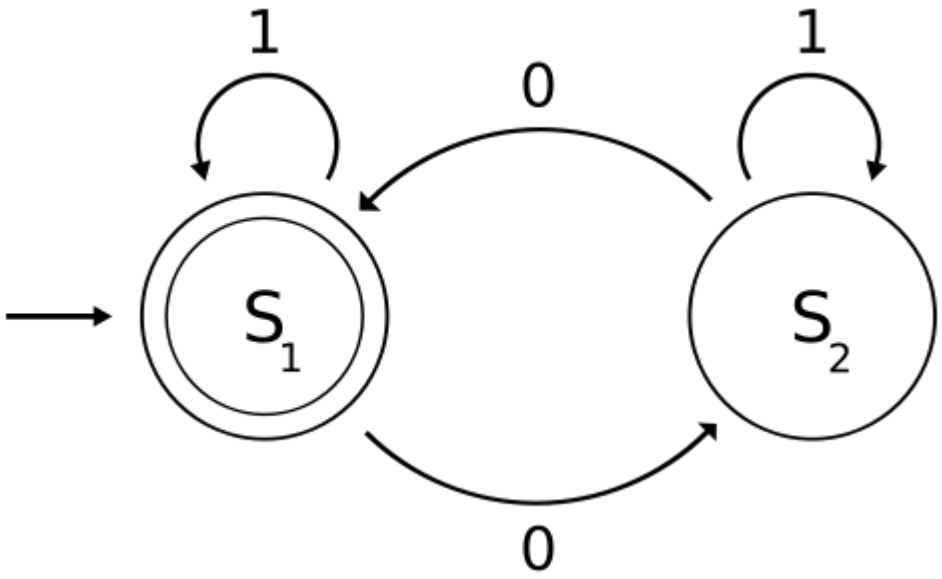
1. $r_0 = q_0$
2. $r_{i+1} = \delta(r_i, a_{i+1})$, for $i = 0, \ldots, n-1$
3. $r_n \in F$.

In words, the first condition says that the machine starts in the start state q_0. The second condition says that given each character of string w, the machine will transition from state to state according to the transition function δ. The last condition says that the machine accepts w if the last input of w causes the machine to halt in one of the accepting states. Otherwise, it is said that the automaton *rejects* the string. The set of strings M accepts is the language *recognized* by M and this language is denoted by $L(M)$.

A deterministic finite automaton without accept states and without a starting state is known as a transition system or semiautomaton.

Example

The following example is of a DFA M, with a binary alphabet, which requires that the input contains an even number of 0s.



The state diagram for M

$M = (Q, \Sigma, \delta, q_0, F)$ where

- $Q = \{S_1, S_2\}$,
- $\Sigma = \{0, 1\}$,
- $q_0 = S_1$,
- $F = \{S_1\}$, and
- δ is defined by the following state transition table:

	0	**1**
$\boldsymbol{S_1}$	S_2	S_1
$\boldsymbol{S_2}$	S_1	S_2

The state S_1 represents that there has been an even number of 0s in the input so far, while S_2 signifies an odd number. A 1 in the input does not change the state of the automaton. When the input ends, the state will show whether the input contained an even number of 0s or not. If the input did contain an even number of 0s, M will finish in state S_1, an accepting state, so the input string will be accepted.

The language recognized by M is the regular language given by the regular expression

$$1^*\big(0(1^*)0(1^*)\big)^*,$$

where "*" is the Kleene star, e.g., 1* denotes any number (possibly zero) of symbols "1".

Accept and Generate modes

A DFA representing a regular language can be used either in an accepting mode to validate that an input string is part of the language, or in a generating mode to generate a list of all the strings in the language.

In the accept mode an input string is provided which the automaton can read in left to right, one symbol at a time. The computation begins at the start state and proceeds by reading the first symbol from the input string and following the state transition corresponding to that symbol. The system continues reading symbols and following transitions until there are no more symbols in the input, which marks the end of the computation. If after all input symbols have been processed the system is in an accept state then we know that the input string was indeed part of the language, and it is said to be accepted, otherwise it is not part of the language and it is not accepted.

The generating mode is similar except that rather than validating an input string its goal is to produce a list of all the strings in the language. Instead of following a single transition out of each state, it follows all of them. In practice this can be accomplished by massive parallelism (having the program branch into two or more processes each time it is faced with a decision) or through recursion. As before, the computation begins at the start state and then proceeds to follow each available transition, keeping track of which branches it took. Every time the automaton finds itself in an accept state it knows that the sequence of branches it took forms a valid string in the language and it adds that string to the list that it is generating. If the language this automaton describes is infinite (ie contains an infinite number or strings, such as "all the binary string with an even number of 0s) then the computation will never halt. Given that regular languages are, in general, infinite, automata in the generating mode tends to be more of a theoretical construct.

DFA as a transition monoid

Alternatively a run can be seen as a sequence of compositions of transition function with itself. Given an input symbol $a \in \Sigma$, one may write the transition function as $\delta_a : Q \to Q$, using the simple trick of currying, that is, writing $\delta(q,a) = \delta_a(q)$ for all $q \in Q$. This way, the transition function can be seen in simpler terms: it's just something that "acts" on a state in Q, yielding another state. One may then consider the result of function composition repeatedly applied to the various functions δ_a, δ_b, and so on. Using this notion we define $\widehat{\delta} : Q \times \Sigma^\star \to Q$. Given a pair of letters $a, b \in \Sigma$, one may define a new function $\widehat{\delta}$, by insisting that $\widehat{\delta}_{ab} = \delta_a \circ \delta_b$, where $\circ$ denotes function composition. Clearly, this process can be recursively continued. So, we have following recursive definition

$\widehat{\delta}(q, \epsilon) = q$ where ε is empty string and
$\widehat{\delta}(q, wa) = \delta_a(\widehat{\delta}(q, w))$ where $w \in \Sigma^*, a \in \Sigma$ and $q \in Q$.

$\widehat{\delta}$ is defined for all words $w \in \Sigma^*$. Repeated function composition forms a monoid. For the transition functions, this monoid is known as the transition monoid, or sometimes the *transformation semigroup*. The construction can also be reversed: given a $\widehat{\delta}$, one can reconstruct a δ, and so the two descriptions are equivalent.

Advantages and disadvantages

DFAs are one of the most practical models of computation, since there is a trivial linear time, constant-space, online algorithm to simulate a DFA on a stream of input. Given two DFAs there are efficient algorithms to find a DFA recognizing:

- the union of the two DFAs
- the intersection of the two DFAs
- complements of the languages the DFAs recognize

Because DFAs can be reduced to a *canonical form* (minimal DFAs), there are also efficient algorithms to determine:

- whether a DFA accepts any strings
- whether a DFA accepts all strings
- whether two DFAs recognize the same language
- the DFA with a minimum number of states for a particular regular language

DFAs are equivalent in computing power to nondeterministic finite automata.

On the other hand, finite state automata are of strictly limited power in the languages they can recognize; many simple languages, including any problem that requires more than constant space to solve, cannot be recognized by a DFA. The classical example of a simply described language that no DFA can recognize is bracket language, that is, language that consists of properly paired brackets, such as (()()). More formally the language consisting of strings of the form $a^n b^n$—some finite number of a's, followed by an equal number of b's. If there is no limit to recursion (i.e., you can always embed another pair of brackets inside) it would require an infinite amount of states to recognize.

Chapter 11

Quantum Finite Automata and Abstract Machine

Quantum finite automata

In quantum computing, **quantum finite automata** or **QFA** are a quantum analog of probabilistic automata. They are related to quantum computers in a similar fashion as finite automata are related to Turing machines. Several types of automata may be defined, including *measure-once* and *measure-many* automata. Quantum finite automata can also be understood as the quantization of subshifts of finite type, or as a quantization of Markov chains. QFA's are, in turn, special cases of **geometric finite automata** or **topological finite automata**.

The automata work by accepting a finite-length string $\sigma = (\sigma_0, \sigma_1, \cdots, \sigma_k)$ of letters σ_i from a finite alphabet $\Sigma \ni \sigma_i$, and assigning to each such string a probability $\Pr(\sigma)$ indicating the probability of the automaton being in an accept state; that is, indicating whether the automaton accepted or rejected the string.

Measure-once automata

Measure-once automata were introduced by Moore and Crutchfield. They may be defined formally as follows.

As with an ordinary finite automaton, the quantum automaton is considered to have *N* possible internal states, represented in this case by an *N*-state qubit $|\psi\rangle$. More precisely, the *N*-state qubit $|\psi\rangle \in \mathbb{C}P^N$ is an element of *N*-dimensional complex projective space, carrying an inner product $\|\cdot\|$ that is the Fubini-Study metric.

The state transitions, transition matrixes or de Bruijn graphs are represented by a collection of $N \times N$ unitary matrixes U_α, with one unitary matrix for each letter $\alpha \in \Sigma$. That is, given an input letter α, the unitary matrix describes the transition of the automaton from its current state $|\psi\rangle$ to its next state $|\psi'\rangle$:

$$|\psi'\rangle = U_\alpha |\psi\rangle$$

Thus, the triple $(\mathbb{C}P^N, \Sigma, \{U_\alpha | \alpha \in \Sigma\})$ form a quantum semiautomaton.

The accept state of the automaton is given by an $N \times N$ projection matrix *P*, so that, given a *N*-dimensional quantum state $|\psi\rangle$, the probability of $|\psi\rangle$ being in the accept state is

$$\langle\psi|P|\psi\rangle = \|P|\psi\rangle\|^2$$

The probability of the state machine accepting a given finite input string $\sigma = (\sigma_0, \sigma_1, \cdots, \sigma_k)$ is given by

$$\Pr(\sigma) = \|PU_{\sigma_k} \cdots U_{\sigma_1} U_{\sigma_0} |\psi\rangle\|^2$$

Here, the vector $|\psi\rangle$ is understood to represent the initial state of the automaton, that is, the state the automaton was in before it stated accepting the string input. The empty string $\varnothing$ is understood to be just the unit matrix, so that

$$\Pr(\varnothing) = \|P|\psi\rangle\|^2$$

is just the probability of the initial state being an accepted state.

Because the left-action of U_α on $|\psi\rangle$ reverses the order of the letters in the string σ, it is not uncommon for QFA's to be defined using a right action on the Hermitian transpose states, simply in order to keep the order of the letters the same.

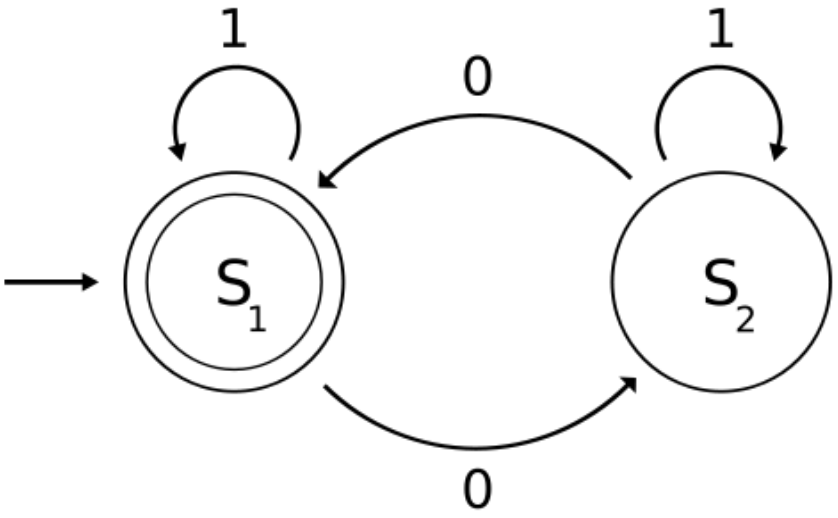
A regular language is accepted with probability *p* by a quantum finite automaton, if, for all sentences σ in the language, (and a given, fixed initial state $|\psi\rangle$), one has $p < \Pr(\sigma)$.

Example

Consider the classical deterministic finite state machine given by the state transition table

State Transition Table		
Input State	**1**	**0**
S_1	S_1	S_2
S_2	S_2	S_1

State Diagram



The quantum state is a vector, in bra-ket notation

$$|\psi\rangle = a_1|S_1\rangle + a_2|S_2\rangle = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

with the complex numbers a_1,a_2 normalized so that

$$\begin{bmatrix} a_1^* & a_2^* \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = a_1^* a_1 + a_2^* a_2 = 1$$

The unitary transition matrices are

$$U_0 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

and

$$U_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Taking S_1 to be the accept state, the projection matrix is

$$P = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

As should be readily apparent, if the initial state is the pure state $|S_1\rangle$ or $|S_2\rangle$, then the result of running the machine will be exactly identical to the classical deterministic finite state machine. In particular, there is a language accepted by this automaton with probability one, for these initial states, and it is identical to the regular language for the classical DFA, and is given by the regular expression:

$$(1^*(01^*0)^*)^*$$

The non-classical behaviour occurs if both a_1 and a_2 are non-zero. More subtle behaviour occurs when the matrices U_0 and U_1 are not so simple; see, for example, the de Rham curve as an example of a quantum finite state machine acting on the set of all possible finite binary strings.

Measure-many automata

Measure-many automata were introduced by Kondacs and Watrous in 1997.. The general framework resembles that of the measure-once automaton, except that instead of there being one projection, at the end, there is a projection, or quantum measurement, performed after each letter is read. A formal definition follows.

The Hilbert space $\mathcal{H}_Q = \mathbb{C}P^N$ is decomposed into three orthogonal subspaces

$$\mathcal{H}_Q = \mathcal{H}_{\text{accept}} \oplus \mathcal{H}_{\text{reject}} \oplus \mathcal{H}_{\text{non-halting}}$$

In the literature, these orthogonal subspaces are usually formulated in terms of the set Q of orthogonal basis vectors for the Hilbert space $\mathcal{H}_Q$. This set of basis vectors is divided up into subsets $Q_{\text{acc}} \subset Q$ and $Q_{\text{rej}} \subset Q$, such that

$$\mathcal{H}_{\text{accept}} = \text{span}\{|q\rangle : |q\rangle \in Q_{\text{acc}}\}$$

is the linear span of the basis vectors in the accept set. The reject space is defined analogously, and the remaining space is designated the *non-halting* subspace. There are three projection matrices, P_{acc}, P_{rej} and P_{non}, each projecting to the respective subspace:

$$P\text{acc} : \mathcal{H}_Q \to \mathcal{H}_{\text{accept}}$$

and so on. The parsing of the input string proceeds as follows. Consider the automaton to be in a state $|\psi\rangle$. After reading an input letter α, the automaton will be in the state

$$|\psi'\rangle = U_\alpha |\psi\rangle$$

At this point, a measurement is performed on the state $|\psi'\rangle$, using the projection operators *P*, at which time its wave-function collapses into one of the three subspaces $\mathcal{H}_{\mathrm{accept}}$ or $\mathcal{H}_{\mathrm{reject}}$ or $\mathcal{H}_{\mathrm{non\text{-}halting}}$. The probability of collapse is given by

$$\Pr_{\mathrm{acc}}(\sigma) = \|P_{\mathrm{acc}}|\psi'\rangle\|^2$$

for the "accept" subspace, and analogously for the other two spaces.

If the wave function has collapsed to either the "accept" or "reject" subspaces, then further processing halts. Otherwise, processing continues, with the next letter read from the input, and applied to what must be an eigenstate of P_{non}. Processing continues until the whole string is read, or the machine halts. Often, additional symbols κ and $ are adjoined to the alphabet, to act as the left and right end-markers for the string.

In the literature, the meaure-many automaton is often denoted by the tuple $(Q;\Sigma;\delta;q_0;Q_{\mathrm{acc}};Q_{\mathrm{rej}})$. Here, *Q*, Σ, *Q*acc and *Q*rej are as defined above. The initial state is denoted by $|\psi\rangle = |q_0\rangle$. The unitary transformations are denoted by the map δ,

$$\delta : Q \times \Sigma \times Q \to \mathbb{C}$$

so that

$$U_\alpha|q_1\rangle = \sum_{q_2 \in Q} \delta(q_1, \alpha, q_2)|q_2\rangle$$

Geometric generalizations

The above constructions indicate how the concept of a quantum finite automaton can be generalized to arbitrary topological spaces. For example, one may take some (*N*-dimensional) Riemann symmetric space to take the place of $\mathbb{C}P^N$. In place of the unitary matrices, one uses the isometries of the Riemannian manifold, or, more generally, some set of open functions appropriate for the given topological space. The initial state may be taken to be a point in the space. The set of accept states can be taken to be some arbitrary subset of the topological space. One then says that a formal language is accepted by this **topological automaton** if the point, after iteration by the homeomorphisms, intersects the accept set. But, of course, this is nothing more than the standard definition of an M-automaton. The behaviour of topological automata is studied in the field of topological dynamics.

The quantum automaton differs from the topological automaton in that, instead of having a binary result (is the iterated point in, or not in, the final set?), one has a probability. The quantum probability is the (square of) the initial state projected onto some final state *P*; that is $\mathbf{Pr} = |\langle P|\psi\rangle|^2$. But this probability amplitude is just a very simple function of the distance between the point $|P\rangle$ and the point $|\psi\rangle$ in $\mathbb{C}P^N$, under the distance metric

given by the Fubini-Study metric. To recap, the quantum probability of a language being accepted can be interpreted as a metric, with the probability of accept being unity, if the metric distance between the initial and final states is zero, and otherwise the probability of accept is less than one, if the metric distance is non-zero. Thus, it follows that the quantum finite automaton is just a special case of a **geometric automaton** or a **metric automaton**, where $\mathbb{C}P^N$ is generalized to some metric space, and the probability measure is replaced by a simple function of the metric on that space.

Abstract machine

An **abstract machine**, also called an **abstract computer**, is a theoretical model of a computer hardware or software system used in automata theory. Abstraction of computing processes is used in both the computer science and computer engineering disciplines and usually assumes discrete time paradigm.

In the theory of computation, abstract machines are often used in thought experiments regarding computability or to analyze the complexity of algorithms. A typical abstract machine consists of a definition in terms of input, output, and the set of allowable operations used to turn the former into the latter. The best-known example is the Turing machine.

More complex definitions create abstract machines with full instruction sets, registers and models of memory. One popular model more similar to real modern machines is the RAM model, which allows random access to indexed memory locations. As the performance difference between different levels of cache memory grows, cache-sensitive models such as the external-memory model and cache-oblivious model are growing in importance.

An abstract machine can also refer to a microprocessor design which has yet to be (or is not intended to be) implemented as hardware. An abstract machine implemented as a software simulation, or for which an interpreter exists, is called a virtual machine.

Through the use of abstract machines it is possible to compute the amount of resources (time, memory, etc.) necessary to perform a particular operation without having to construct an actual system to do it.

Articles concerning Turing-equivalent sequential abstract machine models

An approach is to take a somewhat formal taxonomic approach to classify the Turing equivalent abstract machines. This taxonomy does not include finite automata:

Family: Turing-equivalent (TE) abstract machine:

Subfamilies:

Subfamily (1) Sequential TE abstract machine
Subfamily (2) Parallel TE abstract machine

Subfamily (1)-- *Sequential* TE abstract machine model: There are two classes (genera) of Sequential TE abstract machine models currently in use (cf van Emde Boas, for example):

Genus (1.1) Tape-based Turing machine model
Genus (1.2) Register-based register machine

Genus (1.1) -- Tape-based Turing machine model: This includes the following "species":

{ single tape, Multi-tape Turing machine, deterministic Turing machine, Non-deterministic Turing machine, Wang B-machine, Post-Turing machine, Oracle machine, Universal Turing machine }

Genus (1.2)-- The register machine model: This includes (at least) the following four "species" (others are mentioned by van Emde Boas):

{ (1.2.1) Counter machine, (1.2.2) Random access machine RAM, (1.2.3) Random access stored program machine RASP, (1.2.4) Pointer machine }
Species (1.2.1) -- Counter machine model:
{ abacus machine, Lambek machine, Melzak model, Minsky machine, Shepherdson-Sturgis machine, program machine, etc. }
Species (1.2.2) -- Random access machine (RAM) model:
{ any counter-machine model with additional *indirect addressing*, but with instructions in the state machine in the Harvard architecture; any model with an "accumulator" with additional indirect addressing but instructions in the state machine in the Harvard architecture }
Species (1.2.3) -- Random access stored program machine (RASP) model includes
{ any RAM with program stored in the registers similar to the Universal Turing machine i.e. in the von Neumann architecture }
Species (1.2.4)-- Pointer machine model includes the following:
= { Schönhage Storage Modification Machine SMM, Kolmogorov-Uspensky KU-machine, Knuth linking automaton }

Other abstract machines

- ABC programming language
- Abstract Machine Notation

- ALF programming language
- Categorical Abstract Machine Language
- Context-free grammar
- Finite automata
- Specification and Design Language
- Historycal/Simplicity Abstract Machines for Prolog:
    - 1.Vienna Abstract Machine (VAM Prolog)
    - 2.Warren Abstract Machine (WAM Prolog)
    - 3.Berkeley Abstract Machine (BAM Prolog).
- MMIX
- MikroSim
- Ten15
- TenDRA Distribution Format

Chapter 12

Nondeterministic Finite-state Machine

In the theory of computation, a **nondeterministic finite state machine** or **nondeterministic finite automaton (NFA)** is a finite state machine where for each pair of state and input symbol there may be several possible next states. This distinguishes it from the deterministic finite automaton (DFA), where the next possible state is uniquely determined. Although the DFA and NFA have distinct definitions, it may be shown in the formal theory that they are equivalent, in that, for any given NFA, one may construct an equivalent DFA, and vice-versa: this is the powerset construction. Both types of automata recognize only regular languages. Non-deterministic finite state machines are sometimes studied by the name subshifts of finite type. Non-deterministic finite state machines are generalized by probabilistic automata, which assign a probability to each state transition.

Nondeterministic finite automata were introduced in 1959 by Michael O. Rabin and Dana Scott, who also showed their equivalence to deterministic finite automata.

Intuitive introduction

An NFA, similar to a DFA, consumes a string of input symbols. For each input symbol it transitions to a new state until all input symbols have been consumed.

Unlike a DFA, it is non-deterministic in that, for any input symbol, its next state may be any one of several possible states. Thus, in the formal definition, the next state is an element of the power set of states. This element, itself a set, represents some subset of all possible states to be considered at once.

An extension of the NFA is the **NFA-lambda** (also known as **NFA-epsilon** or the **NFA with epsilon moves**), which allows a transformation to a new state without consuming any input symbols. For example, if it is in state 1, with the next input symbol an *a*, it can move to state 2 without consuming any input symbols, and thus there is an ambiguity: is the system in state 1, or state 2, before consuming the letter *a*? Because of this ambiguity, it is more convenient to talk of the set of possible states the system may be in. Thus, before consuming letter *a*, the NFA-epsilon may be in any one of the states out of the set {1,2}. Equivalently, one may imagine that the NFA is in state 1 and 2 'at the same time': and this gives an informal hint of the powerset construction: the DFA equivalent to an NFA is defined as the one that is in the state q={1,2}. Transformations to new states

without consuming an input symbol are called **lambda transitions** or **epsilon transitions**. They are usually labeled with the Greek letter λ or ε.

The notion of accepting an input is similar to that for the DFA. When the last input symbol is consumed, the NFA accepts if and only if there is *some* set of transitions that will take it to an accepting state. Equivalently, it rejects, if, no matter what transitions are applied, it would not end in an accepting state.

Formal definition

Two similar types of NFAs are commonly defined: the NFA and the *NFA with ε-moves*. The ordinary is defined as a 5-tuple, (Q, Σ, T, q_0, F), consisting of

- a finite set of states Q
- a finite set of input symbols Σ
- a transition function $T : Q \times \Sigma \to P(Q)$.
- an *initial* (or *start*) state $q_0 \in Q$
- a set of states F distinguished as *accepting* (or *final*) *states* $F \subseteq Q$.

Here, $P(Q)$ denotes the power set of Q. The *NFA with ε-moves* (also sometimes called *NFA-epsilon* or *NFA-lambda*) replaces the transition function with one that allows the empty string ε as a possible input, so that one has instead

$$T : Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q).$$

It can be shown that ordinary NFA and NFA with epsilon moves are equivalent, in that, given either one, one can construct the other, which recognizes the same language.

Properties

The machine starts in the specified initial state and reads in a string of symbols from its alphabet. The automaton uses the state transition function T to determine the next state using the current state, and the symbol just read or the empty string. However, "the next state of an NFA depends not only on the current input event, but also on an arbitrary number of subsequent input events. Until these subsequent events occur it is not possible to determine which state the machine is in". If, when the automaton has finished reading, it is in an accepting state, the NFA is said to accept the string, otherwise it is said to reject the string.

The set of all strings accepted by an NFA is the language the NFA accepts. This language is a regular language.

For every NFA a deterministic finite state machine (DFA) can be found that accepts the same language. Therefore it is possible to convert an existing NFA into a DFA for the purpose of implementing a (perhaps) simpler machine. This can be performed using the

powerset construction, which may lead to an exponential rise in the number of necessary states. A formal proof of the powerset construction is given here.

Properties of NFA-ε

For all $p, q \in Q$, one writes $p \xrightarrow{\epsilon} q$ if and only if q can be reached from p by going along zero or more ε arrows. In other words, $p \xrightarrow{\epsilon} q$ if and only if there exists $q_1, q_2, \cdots q_k \in Q$ where $k \geq 0$ such that

$$q_1 \in T(p, \epsilon), q_2 \in T(q_1, \epsilon), \cdots q_k \in T(q_{k-1}, \epsilon), q \in T(q_k, \epsilon).$$

For any $p \in Q$, the set of states that can be reached from p is called the **epsilon-closure** or **ε-closure** of p, and is written as

$$E(\{p\}) = \{q \in Q : p \xrightarrow{\epsilon} q\}.$$

For any subset $P \subset Q$, define the ε-closure of P as

$$E(P) = \bigcup_{p \in P} E(\{p\})$$

.

The epsilon-transitions are transitive, in that it may be shown that, for all $q_0, q_1, q_2 \in Q$ and $P \subset Q$, if $q_1 \in E(\{q_0\})$ and $q_2 \in E(\{q_1\})$, then $q_2 \in E(\{q_0\})$.

Similarly, if $q_1 \in E(P)$ and $q_2 \in E(\{q_1\})$ then $q_2 \in E(P)$

Let x be a string over the alphabet $\Sigma \cup \{\varepsilon\}$. An NFA-ε M accepts the string x if there exist both a representation of x of the form $x_1 x_2 \ldots x_n$, where $x_i \in (\Sigma \cup \{\varepsilon\})$, and a sequence of states $p_0, p_1, \ldots, p_n$, where $p_i \in Q$, meeting the following conditions:

1. $p_0 \in E(\{q_0\})$
2. $p_i \in E(T(p_{i-1}, x_i))$ for $i = 1, \ldots, n$
3. $p_n \in F$.

Implementation

There are many ways to implement a NFA:

- Convert to the equivalent DFA. In some cases this may cause exponential blowup in the size of the automaton and thus auxiliary space proportional to the number

of states in the NFA (as storage of the state value requires at most one bit for every state in the NFA)

- Keep a set data structure of all states which the machine might currently be in. On the consumption of the last input symbol, if one of these states is a final state, the machine accepts the string. In the worst case, this may require auxiliary space proportional to the number of states in the NFA; if the set structure uses one bit per NFA state, then this solution is exactly equivalent to the above.
- Create multiple copies. For each n way decision, the NFA creates up to $n - 1$ copies of the machine. Each will enter a separate state. If, upon consuming the last input symbol, at least one copy of the NFA is in the accepting state, the NFA will accept. (This, too, requires linear storage with respect to the number of NFA states, as there can be one machine for every NFA state.)
- Explicitly propagate tokens through the transition structure of the NFA and match whenever a token reaches the final state. This is sometimes useful when the NFA should encode additional context about the events that triggered the transition.

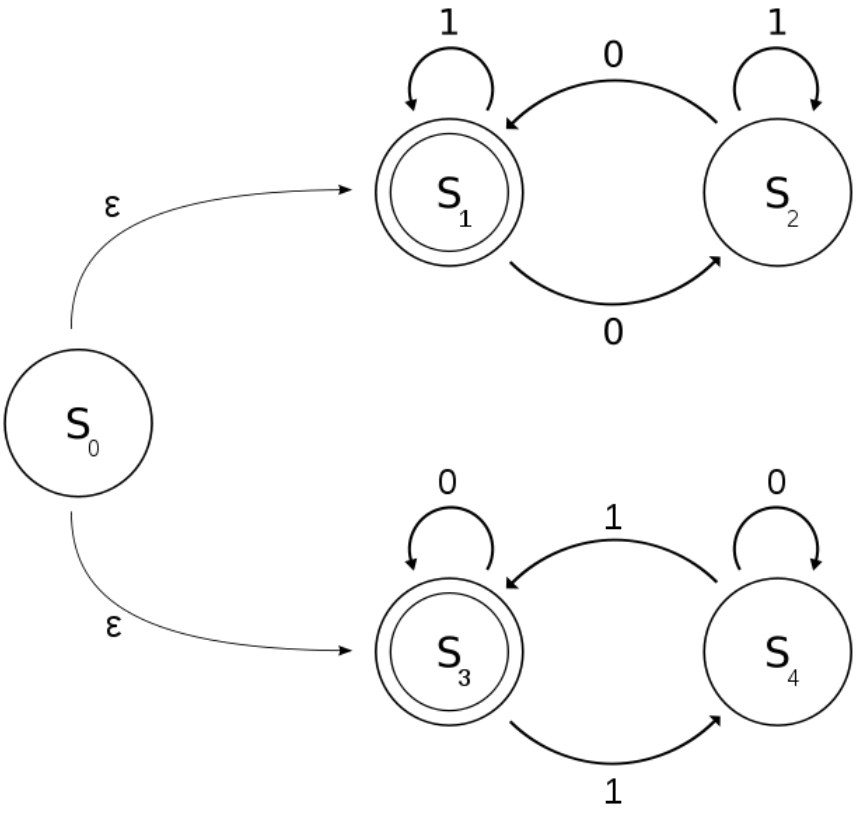
Example

The following example explains a NFA M, with a binary alphabet, which determines if the input contains an even number of 0s or an even number of 1s. (Note that 0 occurrences is an even number of occurrences as well.) Let $M = (Q, \Sigma, T, s_0, F)$ where

- $\Sigma = \{0, 1\}$,
- $Q = \{s_0, s_1, s_2, s_3, s_4\}$,
- $E(\{s_0\}) = \{ s_0, s_1, s_3 \}$
- $F = \{s_1, s_3\}$, and
- The transition function T can be defined by this state transition table:

	0	**1**	**ε**
$\boldsymbol{S_0}$	{}	{}	$\{S_1, S_3\}$
$\boldsymbol{S_1}$	$\{S_2\}$	$\{S_1\}$	{}
$\boldsymbol{S_2}$	$\{S_1\}$	$\{S_2\}$	{}
$\boldsymbol{S_3}$	$\{S_3\}$	$\{S_4\}$	{}
$\boldsymbol{S_4}$	$\{S_4\}$	$\{S_3\}$	{}

The state diagram for M is:



M can be viewed as the union of two DFAs: one with states $\{S_1, S_2\}$ and the other with states $\{S_3, S_4\}$.

The language of *M* can be described by the regular language given by this regular expression:

$$(1^*(01^*01^*)^*) \cup (0^*(10^*10^*)^*)$$

Application of NFA-ε

NFAs and DFAs are equivalent in that if a language is recognized by an NFA, it is also recognized by a DFA and vice versa. The establishment of such equivalence is important and useful. It is useful because constructing an NFA to recognize a given language is sometimes much easier than constructing a DFA for that language. It is important because NFAs can be used to reduce the complexity of the mathematical work required to establish many important properties in the theory of computation. For example, it is much easier to prove the following properties using NFAs than DFAs:

- The union of two regular languages is regular.
- The concatenation of two regular languages is regular.
- The Kleene Closure of a regular language is regular.

Chapter 13

Büchi Automaton and Automatic Sequence

Büchi automaton

In computer science and automata theory, a **Büchi automaton** is a type of ω-automaton, which extends a finite state automaton to infinite inputs. It accepts an infinite input sequence iff there exists a run of the automaton which visits (at least) one of the final states infinitely often. Büchi automata recognize the omega-regular languages, the infinite word version of regular languages. It is named after the Swiss mathematician Julius Richard Büchi who invented this kind of automaton in 1962.

Büchi automata are often used in Model checking as an automata-theoretic version of a formula in linear temporal logic.

Formal definition

Formally, a **deterministic Büchi automaton** is a tuple $A = (Q,\Sigma,\delta,q_0,\mathbf{F})$ that consists of the following components:

- Q is a finite set. The elements of Q are called the *states* of A.
- Σ is a finite set called the *alphabet* of A.
- δ: $Q \times \Sigma \rightarrow Q$ is a function, called the *transition function* of A.
- q_0 is an element of Q, called the initial state.
- $\mathbf{F} \subseteq Q$ is the *acceptance condition*. A accepts exactly those runs in which at least one of the infinitely often occurring states is in **F**.

In a **non-deterministic Büchi automaton**, the transition function δ is replaced with a transition relation Δ that returns a set of states and initial state is q_0 is replaced by a set of initial states Q_0. Generally, Büchi automaton refers to non-deterministic Büchi automaton.

For more comprehensive formalism.

Closure properties

Büchi automata are closed under following operations.

Let A=(Q_A,Σ,Δ_A,I_A,F_A) and B=(Q_B,Σ,Δ_B,I_B,F_B) be Büchi automata and C=(Q_C,Σ,Δ_C,I_C,F_C) be a finite automaton.

- **Union**: *There is a Büchi automaton that recognizes the language L(A) ∪L(B).*

  Proof: If we assume, *w.l.o.g.*, $Q_A \cap Q_B$ is empty then L(A) ∪ L(B) is recognized by the Büchi automaton ($Q_A \cup Q_B$, Σ, $\Delta_A \cup \Delta_B$, $I_A \cup I_B$, $F_A \cup F_B$).

- **Intersection**: *There is a Büchi automaton that recognizes the language L(A)∩L(B).*

  Proof: The Büchi automaton A'=(Q',Σ,Δ',I',F') recognizes L(A)∩L(B), where

  - Q' = $Q_A \times Q_B \times \{1,2\}$
  - $\Delta' = \Delta_1 \cup \Delta_2$
    - Δ_1 = {((q_A,q_B,1), a, (q'_A,q'_B,i)) | (q_A,a,q'_A)∈Δ_A and (q_B,a,q'_B)∈Δ_B and if $q_A \in F_A$ then i=2 else i=1 }
    - Δ_2 = {((q_A,q_B,2), a, (q'_A,q'_B,i)) | (q_A,a,q'_A)∈Δ_A and (q_B,a,q'_B)∈Δ_B and if $q_B \in F_B$ then i=1 else i=2 }
  - I' = $I_A \times I_B \times \{1\}$
  - F' = { (q_A,q_B,2) | $q_B \in F_B$ }

  By construction, r'=(q^0_A,q^0_B,i^0),(q^1_A,q^1_B,i^1),... is a run of automaton A' on input word *w* iff r_A=q^0_A,q^1_A,... is run of A on *w* and r_B=q^0_B,q^1_B,... is run of B on *w*. r_A is accepting and r_B is accepting iff r' is concatenation of an infinite series of finite segments of 1-states(states with third component 1) and 2-states(states with third component 2) alternatively. There is such a series of segments of r' iff r' is accepted by A'.

- **Concatenation**: *There is a Büchi automaton that recognizes the language L(C) ·L(A).*

  Proof: If we assume, *w.l.o.g.*, $Q_C \cap Q_A$ is empty then the Büchi automaton A'=($Q_C \cup Q_A$,Σ,Δ',I',F_A) recognizes L(C) · L(A), where

  - $\Delta' = \Delta_A \cup \Delta_C \cup$ { (q,a,q') | q'∈I_A and ∃ f∈F_C. (q,a,f)∈Δ_C }
  - if $I_C \cap F_C$ is empty then I' = I_C otherwise I' = $I_C \cup I_A$

- **ω-closure**: *If L(C) does not contain empty word then there is a Büchi automaton that recognizes the language L(C)$^{\omega}$.*

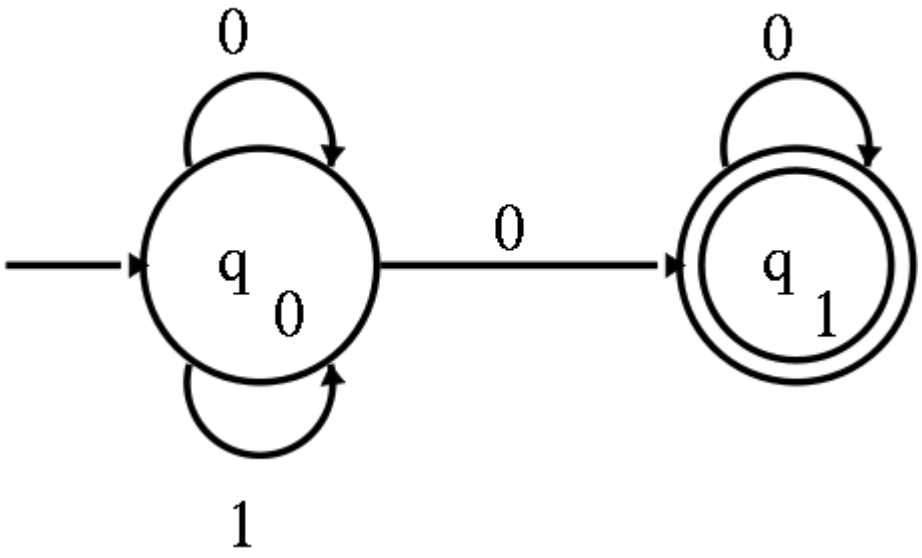
Proof: The Büchi automaton that recognizes $L(C)^{\omega}$ is constructed in two stages. First, we construct an finite automaton A' such that A' also recognizes L(C) but there are no incoming transitions to initial states of A'. So, $A'=(Q_C \cup \{q_{new}\},\Sigma,\Delta',\{q_{new}\},F_C)$, where $\Delta' = \Delta_C \cup \{ (q_{new},a,q') \mid \exists q \in I_C.\ (q,a,q') \in \Delta_C\}$. Note that L(C)=L(A') because L(C) does not contain the empty string. Second, we will construct the Büchi automaton A' ' that recognize $L(C)^{\omega}$ by adding a loop back to the initial state of A'. So, $A'' =(Q_C \cup \{q_{new}\},\Sigma,\Delta'',\{q_{new}\},\{q_{new}\})$, where $\Delta'' = \Delta' \cup \{ (q,a,q_{new}) \mid \exists q' \in F_C.\ (q,a,q') \in \Delta'\}$.

- **Complementation**:*There is a Büchi automaton that recognizes the language Σ*/L(A).*

Recognizable languages

Büchi automata recognize the ω-regular languages. Using the definition of ω-regular language and the above closure properties of Büchi automata, it can be easily shown that a Büchi automaton can be constructed such that it recognizes any given ω-regular language.

Deterministic versus non-deterministic Büchi automata



A non-deterministic büchi automaton that recognizes $(0+1)^*0^{\omega}$

The class of deterministic Büchi automata does not suffice to encompass all omega-regular languages. In particular, there is no deterministic Büchi automaton that recognizes the language $(0+1)^*0^{\omega}$ (Any word that has an infinite suffix consisting of only 0's). We can demonstrate by contradiction that no such deterministic Büchi automaton exists. Let us suppose *A* is a deterministic Büchi automaton that recognize $(0+1)^*0^{\omega}$ with

final state set F. A accepts 0^ω. So, A will visit some state in F after reading some finite prefix of 0^ω, say after the i_0th letter. A also accepts the ω-word $0^{i}_{0}10^\omega$. Therefore, for some i_1, after the prefix $0^{i}_{0}10^{i}_{1}$ the automaton will visit some state in F. Continuing with this construction the ω-word $0^{i}_{0}10^{i}_{1}10^{i}_{2}...$ is generated which causes A to visit some state in F infinitely often and the word is not in $(0+1)^*0^\omega$. Contradiction.

The class of languages recognizable by deterministic Büchi automata is characterized by the following lemma.

Lemma: *An ω-language is recognizable by a deterministic Büchi automaton iff it is the limit language of some regular language.*
Proof: Any deterministic Büchi automaton A can be viewed as a deterministic finite automaton A' and vice-versa, since both types of automaton are defined as 5-tuple of the same components, only the interpretation of acceptance condition is different. We will show that L(A) is the limit language of L(A'). An ω-word is accepted by A iff it will force A to visit final states infinitely often. Iff, infinitely many finite prefixes of this ω-word will be accepted by A'. Hence, L(A) is a limit language of L(A').

Emptiness checking

Model checking problems for finite state processes can often be translated to emptiness checking for Büchi automata.

The language recognized by a Büchi automaton is non-empty if and only if there is a final state that is both reachable from the initial state, and lies on a cycle.

An effective algorithm that can check emptiness of a Büchi automaton:

1. Consider the automaton as a directed graph and decompose it into strongly connected components.
2. Run a search (e.g., the depth-first search) to find which components are reachable from the initial state.
3. Check whether there is a non-trivial strongly connected component that is both reachable and contains a final state.

Each of the steps of this algorithm can be done in time linear in the automaton size, hence the algorithm is clearly optimal.

Variants

- Co-Büchi automaton
- Semi-deterministic Büchi automaton
- Generalized Büchi automaton

Transforming from other models of description to non-deterministic Büchi automata

From generalized Büchi automata(GBA)

Multiple sets of states in acceptance condition can be translated into one set of states by an automata construction, which is known as "counting construction". Lets say $A = (Q,\Sigma,\Delta,q_0,\{F_0,...,F_n\})$ is a GBA, where $F_0,...,F_n$ are sets of accepting states then the equivalent Büchi automaton is $A' = (Q', \Sigma, \Delta',q'_0,F')$, where

- $Q' = Q \times \{1,...,n\}$
- $q'_0 = (q_0,1)$
- $\Delta' = \{ ((q,i), a, (q',j)) \mid (q,a,q') \in \Delta$ and if $q \in F_i$ then $j=(i \bmod n)+1$ else $j=i \}$
- $F'=F_1 \times \{1\}$

From Muller automata

A given Muller automaton can be transformed into an equivalent Büchi automaton with following automata construction. Lets suppose $A = (Q,\Sigma,\Delta,Q_0,\{F_0,...,F_n\})$ is a Muller automaton, where $F_0,...,F_n$ are sets of accepting states. An equivalent Büchi automaton is $A' = (Q', \Sigma, \Delta',Q_0,F')$, where

- $Q' = Q \cup \cup^{n}_{i=0} \{i\} \times F_i \times 2^{F}_{i}$
- $\Delta' = \Delta \cup \Delta_1 \cup \Delta_2$, where
  - $\Delta_1 = \{ (q, a, (i,q',\varnothing)) \mid (q, a, q') \in \Delta$ and $q' \in F_i \}$
  - $\Delta_2 = \{ ((i,q,R), a, (i,q',R')) \mid (q,a,q') \in \Delta$ and $q,q' \in F_i$ and if $R=F_i$ then $R'=\varnothing$ otherwise $R'=R \cup \{q\} \}$
- $F' = \cup^{n}_{i=0} \{i\} \times F_i \times \{F_i\}$

A' keeps original set of states from A and adds extra states on them. The Büchi automaton A' simulates the Muller automaton A as follows: At the beginning of the input word, the execution of A' follows the execution of A, since initial states are same and Δ' contains Δ. At some non-deterministically chosen position in the input word, A' decides of jump into newly added states via a transition in Δ_1. Then, the transitions in Δ_2 try to visit all the states of F_i and keep growing R. Once R becomes equal to F_i then it is reset to the empty set and Δ_2 try to visit all the states of F_i states again and again. So, if the states $R=F_i$ are visited infinitely often then A' accepts corresponding input and so does A. This construction closely follows the first part of the proof of McNaughton's Theorem.

From Kripke structures

Let the given Kripke structure be defined by $M = \langle Q, I, R, L, AP \rangle$ where Q is the set of states, I is the set of initial states, R is a relation between two states also interpreted as an edge, L is the label for the state and AP are the set of atomic propositions that form L.

The Büchi automaton will have the following characteristics:

$Q_{\text{final}} = Q \cup \{\text{init}\}$

$\Sigma = 2^{AP}$

I = {init}
$F = Q \cup \{\text{init}\}$
$\delta = q \xrightarrow{a} p$ if (q, p) belongs to R and $L(p) = a$
and init $\xrightarrow{a} q$ if q belongs to I and $L(q) = a$.
Note however that there is a difference in the interpretation between Kripke structures and Büchi automata. While the former explicitly names every state variable's polarity for every state, the latter just declares the current set of variables holding or not holding true. It says absolutely nothing about the other variables that could be present in the model.

Automatic sequence

An **automatic sequence** (or **k-automatic sequence**) is an infinite sequence of terms characterized by a finite automaton. The n-th term of the sequence is a mapping of the final state of the automaton when its input is the digits of n in some fixed base k. A **k-automatic set** is a set of non-negative integers for which the sequence of values of its characteristic function is an automatic sequence: that is, membership of n in the set can be determined by a finite state automaton on the digits of n in base k.

Automaton point of view

Let q be an integer, and $A = (E, \varphi, e)$ be a deterministic automaton where

- E is the finite set of states
- $\varphi : E \times [0, q-1] \to E$ is the transition function
- $e \in E$ is the initial state

also let A be a finite set, and $\pi : E \to A$ a projection towards A.

For each n, take $m(n) = \pi(\varphi(e, n'))$ where n' is n written in base q. Then the sequence $m = m(1)m(2)m(3)...$ is called a ***q*-automatic sequence**.

Substitution point of view

Let σ be a morphism of the free monoid E^* with $\sigma(E) \subseteq E^q$, and $e \in E$ such that σ(e) begins by e. Let also be A and π as before. Then if m' is a fixpoint of σ, that is to say m' = σ(m'), then $m = \pi(m')$ is a q-automatic sequence over A.

1-automatic sequences

k-automatic sequences are normally only defined for $k \geq 2$. The concept can be extended to $k = 1$ by defining a 1-automatic sequence to be a sequence whose n-th term depends on

the unary notation for n, that is $(1)^n$. Since a finite state automaton must eventually return to a previously visited state, all 1-automatic sequences are eventually periodic.

Properties

For given k and r, a set is k-automatic if and only if it is k^r-automatic. Otherise, if h and k are multiplicatively independent, then a set is both h-automatic and k-automatic if and only if it is 1-automatic, that is, ultimately periodic.

Examples

The following sequences are automatic:

- Thue-Morse sequence: take $E = A = \{0, 1\}$, $e = 0$, $\pi = \mathrm{id}$, and σ such that $\sigma(0) = 01$, $\sigma(1) = 10$; we get the fixpoint 01101001100101101001011001101001... , which is in fact the Thue-Morse word. The n-th term is the parity of the base 2 representation of n and the sequence is thus 2-automatic.
- Rudin–Shapiro sequence
- Baum–Sweet sequence
- Regular paperfolding sequence

Automatic real number

An *automatic real number* is a real number for which the base-b expansion is an automatic sequence. It is conjectured that all such numbers are either rational or transcendental.

Chapter 14

Regular Expression

In computing, a **regular expression**, also referred to as **regex** or **regexp**, provides a concise and flexible means for matching strings of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

The following examples illustrate a few specifications that could be expressed in a regular expression:

- The sequence of characters "car" appearing consecutively in any context, such as in "car", "cartoon", or "bicarbonate"
- The sequence of characters "car" occurring in that order with other characters between them, such as in "Icelander" or "chandler"
- The word "car" when it appears as an isolated word
- The word "car" when preceded by the word "blue" or "red"
- The word "car" when *not* preceded by the word "motor"
- A dollar sign immediately followed by one or more digits, and then optionally a period and exactly two more digits (for example, "$100" or "$245.99").

Regular expressions can be much more complex than these examples.

Regular expressions are used by many text editors, utilities, and programming languages to search and manipulate text based on patterns. Some of these languages, including Perl, Ruby, Awk, and Tcl, have fully integrated regular expressions into the syntax of the core language itself. Others like C, C++, .NET, Java, and Python instead provide access to regular expressions only through libraries. Utilities provided by Unix distributions—including the editor ed and the filter grep—were the first to popularize the concept of regular expressions.

As an example of the syntax, the regular expression `\bex` can be used to search for all instances of the string "*ex*" that occur after "word boundaries" (signified by the `\b`). Thus `\bex` will find the matching string "ex" in two possible locations, (1) at the beginning of words, and (2) between two characters in a string, where one is a word character and the other is not a word character. For instance, in the string "Texts for experts", `\bex` matches

the "*ex*" in "experts" but not in "Texts" (because the "*ex*" occurs inside a word and not immediately after a word boundary).

Many modern computing systems provide wildcard characters in matching filenames from a file system. This is a core capability of many command-line shells and is also known as globbing. Wildcards differ from regular expressions in generally expressing only limited forms of patterns.

Basic concepts

A regular expression, often called a pattern, is an expression that describes a set of strings. They are usually used to give a concise description of a set, without having to list all elements. For example, the set containing the three strings "*Handel*", "*Händel*", and "*Haendel*" can be described by the pattern `H(ä|ae?)ndel` (or alternatively, it is said that the pattern *matches* each of the three strings). In most formalisms, if there is any regex that matches a particular set then there is an infinite number of such expressions. Most formalisms provide the following operations to construct regular expressions.

Boolean "or"
: A vertical bar separates alternatives. For example, `gray|grey` can match "*gray*" or "*grey*".

Grouping
: Parentheses are used to define the scope and precedence of the operators (among other uses). For example, `gray|grey` and `gr(a|e)y` are equivalent patterns which both describe the set of "*gray*" and "*grey*".

Quantification
: A quantifier after a token (such as a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark `?`, the asterisk `*` (derived from the Kleene star), and the plus sign `+` (Kleene cross).

  ? The question mark indicates there is *zero or one* of the preceding element. For example, `colou?r` matches both "*color*" and "*colour*".

  ***** The asterisk indicates there are *zero or more* of the preceding element. For example, `ab*c` matches "*ac*", "*abc*", "*abbc*", "*abbbc*", and so on.

  + The plus sign indicates that there is *one or more* of the preceding element. For example, `ab+c` matches "*abc*", "*abbc*", "*abbbc*", and so on, but not "*ac*".

These constructions can be combined to form arbitrarily complex expressions, much like one can construct arithmetical expressions from numbers and the operations +, −, ×, and ÷. For example, `H(ae?|ä)ndel` and `H(a|ae|ä)ndel` are both valid patterns which match the same strings as the earlier example, `H(ä|ae?)ndel`.

The precise syntax for regular expressions varies among tools and with context; more detail is given in the *Syntax* section.

History

The origins of regular expressions lie in automata theory and formal language theory, both of which are part of theoretical computer science. These fields study models of computation (automata) and ways to describe and classify formal languages. In the 1950s, mathematician Stephen Cole Kleene described these models using his mathematical notation called *regular sets*. The SNOBOL language was an early implementation of pattern matching, but not identical to regular expressions. Ken Thompson built Kleene's notation into the editor QED as a means to match patterns in text files. He later added this capability to the Unix editor ed, which eventually led to the popular search tool grep's use of regular expressions ("grep" is a word derived from the command for regular expression searching in the ed editor: `g/re/p` where *re* stands for regular expression). Since that time, many variations of Thompson's original adaptation of regular expressions have been widely used in Unix and Unix-like utilities including expr, AWK, Emacs, vi, and lex.

Perl and Tcl regular expressions were derived from a regex library written by Henry Spencer, though Perl later expanded on Spencer's library to add many new features. Philip Hazel developed PCRE (Perl Compatible Regular Expressions), which attempts to closely mimic Perl's regular expression functionality and is used by many modern tools including PHP and Apache HTTP Server. Part of the effort in the design of Perl 6 is to improve Perl's regular expression integration, and to increase their scope and capabilities to allow the definition of parsing expression grammars. The result is a mini-language called Perl 6 rules, which are used to define Perl 6 grammar as well as provide a tool to programmers in the language. These rules maintain existing features of Perl 5.x regular expressions, but also allow BNF-style definition of a recursive descent parser via sub-rules.

The use of regular expressions in structured information standards for document and database modeling started in the 1960s and expanded in the 1980s when industry standards like ISO SGML (precursored by ANSI "GCA 101-1983") consolidated. The kernel of the structure specification language standards are regular expressions. Simple use is evident in the DTD element group syntax.

Formal language theory

Definition

Regular expressions describe regular languages in formal language theory. They have thus the same expressive power as regular grammars. Regular expressions consist of constants and operators that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory. Given a finite alphabet Σ, the following constants are defined:

- (*empty set*) ∅ denoting the set ∅.

- (*empty string*) ε denoting the set containing only the "empty" string, which has no characters at all.
- (*literal character*) `a` in Σ denoting the set containing only the character *a*.

The following operations are defined:

- (*concatenation*) *RS* denoting the set { αβ | α in *R* and β in *S* }. For example {"ab", "c"}{"d", "ef"} = {"abd", "abef", "cd", "cef"}.
- (*alternation*) *R* | *S* denoting the set union of *R* and *S*. For example {"ab", "c"}|{"ab", "d", "ef"} = {"ab", "c", "d", "ef"}.
- (*Kleene star*) *R** denoting the smallest superset of *R* that contains ε and is closed under string concatenation. This is the set of all strings that can be made by concatenating any finite number (including zero) of strings from *R*. For example, {"0","1"}* is the set of all finite binary strings (including the empty string), and {"ab", "c"}* = {ε, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "abcab", ... }.

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and then set union. If there is no ambiguity then parentheses may be omitted. For example, `(ab)c` can be written as `abc`, and `a|(b(c*))` can be written as `a|bc*`. Many textbooks use the symbols ∪, +, or ∨ for alternation instead of the vertical bar.

Examples:

- `a|b*` denotes {ε, *a*, *b*, *bb*, *bbb*, ...}
- `(a|b)*` denotes the set of all strings with no symbols other than *a* and *b*, including the empty string: {ε, *a*, *b*, *aa*, *ab*, *ba*, *bb*, *aaa*, ...}
- `ab*(c|ε)` denotes the set of strings starting with *a*, then zero or more *b*s and finally optionally a *c*: {*a*, *ac*, *ab*, *abc*, *abb*, *abbc*, ...}

Expressive power and compactness

The formal definition of regular expressions is purposely parsimonious and avoids defining the redundant quantifiers `?` and `+`, which can be expressed as follows: `a+` = `aa*`, and `a?` = `(a|ε)`. Sometimes the complement operator is added, to give a *generalized regular expression*; here R^c matches all strings over Σ* that do not match *R*. In principle, the complement operator is redundant, as it can always be circumscribed by using the other operators. However, the process for computing such a representation is complex, and the result may require expressions of a size that is double exponentially larger.

Regular expressions in this sense can express the regular languages, exactly the class of languages accepted by deterministic finite automata. There is, however, a significant difference in compactness. Some classes of regular languages can only be described by deterministic finite automata whose size grows exponentially in the size of the shortest equivalent regular expressions. The standard example are here the languages L_k consisting of all strings over the alphabet {*a*,*b*} whose k^{th}-last letter equals *a*. On the one

hand, a regular expression describing L_4 is given by $(a \mid b)^* a(a \mid b)(a \mid b)(a \mid b)$. Generalizing this pattern to L_k gives the expression

$$(a|b)^*a\underbrace{(a|b)(a|b)\cdots(a|b)}_{k-1 \text{ times}}.$$

On the other hand, it is known that every deterministic finite automaton accepting the language L_k must have at least 2^k states. Luckily, there is a simple mapping from regular expressions to the more general nondeterministic finite automata (NFAs) that does not lead to such a blowup in size; for this reason NFAs are often used as alternative representations of regular languages. NFAs are a simple variation of the type-3 grammars of the Chomsky hierarchy.

Finally, it is worth noting that many real-world "regular expression" engines implement features that cannot be described by the regular expressions in the sense of formal language theory.

Deciding equivalence of regular expressions

As the examples show, different regular expressions can express the same language: the formalism is redundant.

It is possible to write an algorithm which for two given regular expressions decides whether the described languages are essentially equal, reduces each expression to a minimal deterministic finite state machine, and determines whether they are isomorphic (equivalent).

To what extent can this redundancy be eliminated? Kleene star and set union are required to find an interesting subset of regular expressions that is still fully expressive, but perhaps their use can be restricted. This is a surprisingly difficult problem. As simple as the regular expressions are, there is no method to systematically rewrite them to some normal form. The lack of axiom in the past led to the star height problem. Recently, Dexter Kozen axiomatized regular expressions with Kleene algebra.

Syntax

A number of special characters or metacharacters are used to denote actions or delimit groups; but it is possible to force these special characters to be interpreted as normal characters by preceding them with a defined escape character, usually the backslash "\". For example, a dot is normally used as a "wild card" metacharacter to denote any character, but if preceded by a backslash it represents the dot character itself. The pattern `c.t` matches "cat", "cot", "cut", and non-words such as "czt" and "c.t"; but `c\.t` matches only "c.t". The backslash also escapes itself, i.e., two backslashes are interpreted as a literal backslash character.

POSIX

POSIX Basic Regular Expressions

Traditional Unix regular expression syntax followed common conventions but often differed from tool to tool. The IEEE POSIX Basic Regular Expressions (BRE) standard (released alongside an alternative flavor called Extended Regular Expressions or ERE) was designed mostly for backward compatibility with the traditional (Simple Regular Expression) syntax but provided a common standard which has since been adopted as the default syntax of many Unix regular expression tools, though there is often some variation or additional features. Many such tools also provide support for ERE syntax with command line arguments.

In the BRE syntax, most characters are treated as literals — they match only themselves (e.g., `a` matches "*a*"). The exceptions, listed below, are called metacharacters or metasequences.

Metacharacter	Description
`.`	Matches any single character (many applications exclude newlines, and exactly which characters are considered newlines is flavor, character encoding, and platform specific, but it is safe to assume that the line feed character is included). Within POSIX bracket expressions, the dot character matches a literal dot. For example, `a.c` matches "*abc*", etc., but `[a.c]` matches only "*a*", ".", or "*c*".
`[ ]`	A bracket expression. Matches a single character that is contained within the brackets. For example, `[abc]` matches "*a*", "*b*", or "*c*". `[a-z]` specifies a range which matches any lowercase letter from "*a*" to "*z*". These forms can be mixed: `[abcx-z]` matches "*a*", "*b*", "*c*", "*x*", "*y*", or "*z*", as does `[a-cx-z]`. The `-` character is treated as a literal character if it is the last or the first (after the `^`) character within the brackets: `[abc-]`, `[-abc]`. Note that backslash escapes are not allowed. The `]` character can be included in a bracket expression if it is the first (after the `^`) character: `[]abc]`.
`[^ ]`	Matches a single character that is not contained within the brackets. For example, `[^abc]` matches any character other than "*a*", "*b*", or "*c*". `[^a-z]` matches any single character that is not a lowercase letter from "*a*" to "*z*". As above, literal characters and ranges can be mixed.
`^`	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
`$`	Matches the ending position of the string or the position just before a string-ending newline. In line-based tools, it matches the ending position of any line.

BRE: `\( \)` ERE: `( )`	Defines a marked subexpression. The string matched within the parentheses can be recalled later. A marked subexpression is also called a block or capturing group.
`\n`	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9. This construct is theoretically **irregular** and was not adopted in the POSIX ERE syntax. Some tools allow referencing more than nine capturing groups.
`*`	Matches the preceding element zero or more times. For example, `ab*c` matches "*ac*", "*abc*", "*abbbc*", etc. `[xyz]*` matches "", "*x*", "*y*", "*z*", "*zx*", "*zyx*", "*xyzzy*", and so on. `\(ab\)*` matches "", "*ab*", "*abab*", "*ababab*", and so on.
BRE: `\{m,n\}` ERE: `{m,n}`	Matches the preceding element at least *m* and not more than *n* times. For example, `a\{3,5\}` matches only "*aaa*", "*aaaa*", and "*aaaaa*". This is not found in a few older instances of regular expressions.

Examples:

- `.at` matches any three-character string ending with "at", including "*hat*", "*cat*", and "*bat*".
- `[hc]at` matches "*hat*" and "*cat*".
- `[^b]at` matches all strings matched by `.at` except "*bat*".
- `^[hc]at` matches "*hat*" and "*cat*", but only at the beginning of the string or line.
- `[hc]at$` matches "*hat*" and "*cat*", but only at the end of the string or line.
- `\[.\]` matches any single character surrounded by "[" and "]" since the brackets are escaped, for example: "*[a]*" and "*[b]*".

POSIX Extended Regular Expressions

The meaning of metacharacters escaped with a backslash is reversed for some characters in the POSIX Extended Regular Expression (ERE) syntax. With this syntax, a backslash causes the metacharacter to be treated as a literal character. So, for example, `\( \)` is now `( )` and `\{ \}` is now `{ }`. Additionally, support is removed for `\n` backreferences and the following metacharacters are added:

Metacharacter	Description
`?`	Matches the preceding element zero or one time. For example, `ba?` matches "*b*" or "*ba*".
`+`	Matches the preceding element one or more times. For example, `ba+` matches "*ba*", "*baa*", "*baaa*", and so on.
`|`	The choice (aka alternation or set union) operator matches either the expression before or the expression after the operator. For example, `abc|def` matches "*abc*" or "*def*".

Examples:

- `[hc]+at` matches "*hat*", "*cat*", "*hhat*", "*chat*", "*hcat*", "*ccchat*", and so on, but not "*at*".
- `[hc]?at` matches "*hat*", "*cat*", and "*at*".
- `[hc]*at` matches "*hat*", "*cat*", "*hhat*", "*chat*", "*hcat*", "*ccchat*", "*at*", and so on.
- `cat|dog` matches "*cat*" or "*dog*".

POSIX Extended Regular Expressions can often be used with modern Unix utilities by including the command line flag *-E*.

POSIX character classes

Since many ranges of characters depend on the chosen locale setting (i.e., in some settings letters are organized as *abc...zABC...Z*, while in some others as *aAbBcC...zZ*), the POSIX standard defines some classes or categories of characters as shown in the following table:

POSIX	Non-standard	Perl	ASCII	Description
`[:alnum:]`			`[A-Za-z0-9]`	Alphanumeric characters
	`[:word:]`	`\w`	`[A-Za-z0-9_]`	Alphanumeric characters plus "_"
		`\W`	`[^A-Za-z0-9_]`	Non-word characters
`[:alpha:]`			`[A-Za-z]`	Alphabetic characters
`[:blank:]`			`[ \t]`	Space and tab
		`\b`	`[(?<=\W)(?=\w)|(?<=\w)(?=\W)]`	Word boundaries
`[:cntrl:]`			`[\x00-\x1F\x7F]`	Control characters
`[:digit:]`		`\d`	`[0-9]`	Digits
		`\D`	`[^0-9]`	Non-digits
`[:graph:]`			`[\x21-\x7E]`	Visible characters
`[:lower:]`			`[a-z]`	Lowercase letters
`[:print:]`			`[\x20-\x7E]`	Visible characters and spaces
`[:punct:]`			`[][!"#$%&'()*+,./:;<=>?@\^_`{|}~-]`	Punctuation characters
`[:space:]`		`\s`	`[ \t\r\n\v\f]`	Whitespace characters

\S	[^ \t\r\n\v\f]	Non-whitespace characters
[:upper:]	[A-Z]	Uppercase letters
[:xdigit:]	[A-Fa-f0-9]	Hexadecimal digits

POSIX character classes can only be used within bracket expressions. For example, [[:upper:]ab] matches the uppercase letters and lowercase "*a*" and "*b*".

In Perl regular expressions, [:print:] matches [:graph:] union [:space:]. An additional non-POSIX class understood by some tools is [:word:], which is usually defined as [:alnum:] plus underscore. This reflects the fact that in many programming languages these are the characters that may be used in identifiers. The editor Vim further distinguishes *word* and *word-head* classes (using the notation \w and \h) since in many programming languages the characters that can begin an identifier are not the same as those that can occur in other positions.

Note that what the POSIX regular expression standards call *character classes* are commonly referred to as *POSIX character classes* in other regular expression flavors which support them. With most other regular expression flavors, the term *character class* is used to describe what POSIX calls *bracket expressions*.

Perl-derivative regular expressions

Perl has a more consistent and richer syntax than the POSIX basic (BRE) and extended (ERE) regular expression standards. An example of its consistency is that \ always escapes a non-alphanumeric character. Other examples of functionality possible with Perl but not POSIX-compliant regular expressions is the concept of lazy quantification, possessive quantifies to control backtracking, named capture groups, and recursive patterns.

Due largely to its expressive power, many other utilities and programming languages have adopted syntax similar to Perl's — for example, Java, JavaScript, PCRE, Python, Ruby, Microsoft's .NET Framework, and the W3C's XML Schema all use regular expression syntax similar to Perl's. Some languages and tools such as Boost and PHP support multiple regular expression flavors. Perl-derivative regular expression implementations are not identical, and all implement no more than a subset of Perl's features, usually those of Perl 5.0, released in 1994. With Perl 5.10, this process has come full circle with Perl incorporating syntactic extensions originally developed in Python, PCRE, and the .NET Framework.

Simple Regular Expressions

Simple Regular Expressions is a syntax that may be used by historical versions of application programs, and may be supported within some applications for the purpose of providing backward compatibility. It is deprecated.

Lazy quantification

The standard quantifiers in regular expressions are greedy, meaning they match as much as they can, only giving back as necessary to match the remainder of the regex. For example, to find the first instance of an item between < and > symbols in this example:

```
Another whale sighting occurred on <January 26>, <2004>.
```

someone new to regexes would likely come up with the pattern `<.*>` or similar. However, instead of the "*<January 26>*" that might be expected, this pattern will actually return "*<January 26>, <2004>*" because the `*` quantifier is greedy — it will consume as many characters as possible from the input, and "*January 26>, <2004*" has more characters than "*January 26*".

Though this problem can be avoided in a number of ways (e.g., by specifying the text that is *not* to be matched: `<[^>]*>`), modern regular expression tools allow a quantifier to be specified as *lazy* (also known as *non-greedy*, *reluctant*, *minimal*, or *ungreedy*) by putting a question mark after the quantifier (e.g., `<.*?>`), or by using a modifier which reverses the greediness of quantifiers (though changing the meaning of the standard quantifiers can be confusing). By using a lazy quantifier, the expression tries the minimal match first. Though in the previous example lazy matching is used to select one of many matching results, in some cases it can also be used to improve performance when greedy matching would require more backtracking.

Patterns for non-regular languages

Many features found in modern regular expression libraries provide an expressive power that far exceeds the regular languages. For example, many implementations allow grouping subexpressions with parentheses and recalling the value they match in the same expression (**backreferences**).

The language of squares is not regular, nor is it context-free. Pattern matching with an unbounded number of back references, as supported by numerous modern tools, is NP-complete.

However, many tools, libraries, and engines that provide such constructions still use the term *regular expression* for their patterns. This has led to a nomenclature where the term regular expression has different meanings in formal language theory and pattern matching. For this reason, some people have taken to using the term *regex* or simply

pattern to describe the latter. Larry Wall, author of the Perl programming language, writes in an essay about the design of Perl 6:

> “ 'Regular expressions' [...] are only marginally related to real regular expressions. Nevertheless, the term has grown with the capabilities of our pattern matching engines, so I'm not going to try to fight linguistic necessity here. I will, however, generally call them "regexes" (or "regexen", when I'm in an Anglo-Saxon mood). ”

Implementations and running times

There are at least three different algorithms that decide if and how a given regular expression matches a string.

The oldest and fastest two rely on a result in formal language theory that allows every nondeterministic finite automaton (NFA) to be transformed into a deterministic finite automaton (DFA). The DFA can be constructed explicitly and then run on the resulting input string one symbol at a time. Constructing the DFA for a regular expression of size m has the time and memory cost of $O(2^m)$, but it can be run on a string of size n in time $O(n)$. An alternative approach is to simulate the NFA directly, essentially building each DFA state on demand and then discarding it at the next step, possibly with caching. This keeps the DFA implicit and avoids the exponential construction cost, but running cost rises to $O(nm)$. The explicit approach is called the DFA algorithm and the implicit approach the NFA algorithm. As both can be seen as different ways of executing the same DFA, they are also often called the DFA algorithm without making a distinction. These algorithms are fast, but using them for recalling grouped subexpressions, lazy quantification, and similar features is tricky.

The third algorithm is to match the pattern against the input string by backtracking. This algorithm is commonly called NFA, but this terminology can be confusing. Its running time can be exponential, which simple implementations exhibit when matching against expressions like `(a|aa)*b` that contain both alternation and unbounded quantification and force the algorithm to consider an exponentially increasing number of sub-cases. This behavior can cause a security problem called Regular expression Denial of Service - ReDoS, which might be used by hackers who want to attack a regular expression engine. More complex implementations will often identify and speed up or abort common cases where they would otherwise run slowly.

Although backtracking implementations only give an exponential guarantee in the worst case, they provide much greater flexibility and expressive power. For example, any implementation which allows the use of backreferences, or implements the various extensions introduced by Perl, must use a backtracking implementation.

Some implementations try to provide the best of both algorithms by first running a fast DFA match to see if the string matches the regular expression at all, and only in that case perform a potentially slower backtracking match.

Unicode

In theoretical terms, any token set can be matched by regular expressions as long as it is pre-defined. In terms of historical implementations, regular expressions were originally written to use ASCII characters as their token set though regular expression libraries have supported numerous other character sets. Many modern regular expression engines offer at least some support for Unicode. In most respects it makes no difference what the character set is, but some issues do arise when extending regular expressions to support Unicode.

- Supported encoding. Some regular expression libraries expect to work on some particular encoding instead of on abstract Unicode characters. Many of these require the UTF-8 encoding, while others might expect UTF-16, or UTF-32. In contrast, Perl and Java are agnostic on encodings, instead operating on decoded characters internally.
- Supported Unicode range. Many regular expression engines support only the Basic Multilingual Plane, that is, the characters which can be encoded with only 16 bits. Currently, only a few regular expression engines (e.g., Perl's and Java's) can handle the full 21-bit Unicode range.
- Extending ASCII-oriented constructs to Unicode. For example, in ASCII-based implementations, character ranges of the form `[x-y]` are valid wherever *x* and *y* are codepoints in the range [0x00,0x7F] and codepoint(x) ≤ codepoint(y). The natural extension of such character ranges to Unicode would simply change the requirement that the endpoints lie in [0x00,0x7F] to the requirement that they lie in [0,0x10FFFF]. However, in practice this is often not the case. Some implementations, such as that of gawk, do not allow character ranges to cross Unicode blocks. A range like [0x61,0x7F] is valid since both endpoints fall within the Basic Latin block, as is [0x0530,0x0560] since both endpoints fall within the Armenian block, but a range like [0x0061,0x0532] is invalid since it includes multiple Unicode blocks. Other engines, such as that of the Vim editor, allow block-crossing but limit the number of characters in a range to 128.
- Case insensitivity. Some case-insensitivity flags affect only the ASCII characters. Other flags affect all characters. Some engines have two different flags, one for ASCII, the other for Unicode. Exactly which characters belong to the POSIX classes also varies.
- Cousins of case insensitivity. As ASCII has case distinction, case insensitivity became a logical feature in text searching. Unicode introduced alphabetic scripts without case like Devanagari. For these, case sensitivity is not applicable. For scripts like Chinese, another distinction seems logical: between traditional and simplified. In Arabic scripts, insensitivity to initial, medial, final, and isolated position may be desired. In Japanese, insensitivity between hiragana and katakana is sometimes useful.

- Normalization. Unicode introduced combining characters. Like old typewriters, plain letters can be followed by one of more non-spacing symbols (usually diacritics like accent marks) to form a single printing character. Consider a letter with both a grave and an acute accent mark. That might be written with the grave appearing before the acute, or vice versa. As a consequence, two different code sequences can result in identical character display.
- New control codes. Unicode introduced amongst others, byte order marks and text direction markers. These codes might have to be dealt with in a special way.
- Introduction of character classes for Unicode blocks, scripts, and numerous other character properties. Block properties are much less useful than script properties, because a block can have code points from several different scripts, and a script can have code points from several different blocks. In Perl and the `java.util.regex` library, properties of the form `\p{InX}` or `\p{Block=X}` match characters in block *X* and `\P{InX}` or `\P{Block=X}` matches code points not in that block. Similarly, `\p{Armenian}`, `\p{IsArmenian}`, or `\p{Script=Armenian}` matches any character in the Armenian script. In general, `\p{X}` matches any character with either the binary property *X* or the general category *X*. For example, `\p{Lu}`, `\p{Uppercase_Letter}`, or `\p{GC=Lu}` matches any upper-case letter. Binary properties that are *not* general categories include `\p{White_Space}`, `\p{Alphabetic}`, `\p{Math}`, and `\p{Dash}`. Examples of non-binary properties are `\p{Bidi_Class=Right_to_Left}`, `\p{Word_Break=A_Letter}`, and `\p{Numeric_Value=10}`.

Uses

Regular expressions are useful in the production of syntax highlighting systems, data validation, and many other tasks.

While regular expressions would be useful on search engines such as Google, processing them across the entire database could consume excessive computer resources depending on the complexity and design of the regex. Although in many cases system administrators can run regex-based queries internally, most search engines do not offer regex support to the public.

Chapter 15

Deterministic Pushdown Automaton and Embedded Pushdown Automaton

Deterministic pushdown automaton

In automata theory, a pushdown automaton is a finite automaton with an additional stack of symbols; its transitions can take the top symbol on the stack and depend on its value, and they can add new top symbols to the stack. A **deterministic pushdown automaton** is effectively a particular type of pushdown automaton, namely ones that have at most one transition for the same combination of input symbol, state, and top stack symbol. Technically however, the notion of determinism for pushdown automata is more complicated than for finite automata as the transition is determined by both state and top stack symbol. This means that if we omit the stack from a deterministic pushdown automaton we usually end up with a nondeterministic finite automaton.

The term "pushdown" refers to the fact that the stack can be regarded as being "pushed down" like a tray dispenser at a cafeteria, since the operations never work on elements other than the top element. A stack automaton, by contrast, does allow operations on other elements, and stack automata can recognize a strictly larger set of languages than pushdown automata.

A deterministic context-free language is a language recognized by some deterministic pushdown automaton. Not all context-free languages are deterministic. This is unlike the situation for deterministic finite automata, which are also a subset of the nondeterministic finite automata but can recognize the same class of languages (as demonstrated by the subset construction).

Formal Definition

A (not necessarily deterministic) PDA *M* can be defined as a 7-tuple:

$$M = (Q, \Sigma, \Gamma, q_0, Z_0, A, \delta)$$

where

- Q is a finite set of states
- Σ is a finite set of input symbols
- Γ is a finite set of stack symbols
- $q_0 \in Q$ is the start state
- $Z_0 \in \Gamma$ is the starting stack symbol
- $A \subseteq Q$, where *A* is the set of accepting states
- δ is a transition function, where

  $\delta : (Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \longrightarrow \mathcal{P}(Q \times \Gamma^*)$

  where Γ^* means "a finite list (maybe empty) of elements of Γ", ε denotes the empty string, and $\mathcal{P}(X)$ is the power set of a set *X*.

M is *deterministic* if it satisfies both the following conditions:

- For any $q \in Q, a \in \Sigma \cup \{\epsilon\}, x \in \Gamma$, the set $\delta(q, a, x)$ has at most one element.
- For any $q \in Q, x \in \Gamma$, if $\delta(q, \epsilon, x) \neq \emptyset$, then $\delta(q, a, x) = \emptyset$ for every $a \in \Sigma$.

There are two possible acceptance criteria: acceptance by *empty stack* and acceptance by *final state*. The two are not equivalent for the deterministic pushdown automaton (although they are for the non-deterministic pushdown automaton). The languages accepted by *empty stack* are the languages that are accepted by *final state*, as well as have no word in the language that is the prefix of another word in the language.

Computation

The formal definition of the computation is the same as that of the pushdown automaton, with the only difference being that there is now only one computation for each input. For an automaton A, L(A) is the set of inputs such that there is a computation from the initial configuration until an accepting one.

Properties

Closure

Closure properties of deterministic context-free languages (accepted by deterministic PDA by final state) are drastically different from the context-free languages. As an example they are (effectively) closed under complementation, but not closed under union. To prove that the complement of a deterministic PDA is again accepted by a deterministic PDA is tricky. In principle one has to avoid infinite computations.

As a consequence of the complementation it is decidable whether a deterministic PDA accepts all words over its input alphabet, by testing its complement for emptiness. This is not possible for context-free grammars (hence not for general PDA).

Equivalence problem

Geraud Senizergues (2001) proved that the equivalence problem for deterministic PDA (i.e. given two deterministic PDA A and B, is L(A)=L(B)?) is decidable. For nondeterministic PDA, equivalence is undecidable.

Embedded pushdown automaton

An **embedded pushdown automaton** or **EPDA** is a computational model for parsing languages generated by tree-adjoining grammars (TAGs). It is similar to the context-free grammar-parsing pushdown automaton, except that instead of using a plain stack to store symbols, it has a stack of iterated stacks that store symbols, giving TAGs a generative capacity between context-free grammars and context-sensitive grammars, or a subset of the mildly context-sensitive grammars.

History and applications

EPDAs were first described by K. Vijay-Shanker in his 1988 doctoral thesis. They have since been applied to more complete descriptions of the class of mildly context-sensitive grammars and have had important roles in extending and refining the Chomsky hierarchy to this class. Various subgrammars, such as the linear indexed grammar, can thus be defined. They are also beginning to play an important role in natural language processing.

While natural languages have traditionally been analyzed using context-free grammars, this model does not work well for languages with crossed dependencies, such as Dutch, situations for which an EPDA is well suited. A detailed linguistic analysis is available in .

Theory

An EPDA is a finite state machine with a set of stacks that can be themselves accessed through the *embedded stack*. Each stack contains elements of the *stack alphabet* Γ, and so we define an element of a stack by $\sigma_i \in \Gamma^*$, where the star is the Kleene closure of the alphabet.

Each stack can then be defined in terms of its elements, so we denote the jth stack in the automaton using a double-dagger symbol: $\Upsilon_j = \ddagger\sigma_j = \{\sigma_{j,k}, \sigma_{j,k-1}, \ldots, \sigma_{j,1}\}$,

where $\sigma_{j,k}$would be the next accessible symbol in the stack. The *embedded stack* of $m$stacks can thus be denoted by $\{\Upsilon_j\} = \{\ddagger\sigma_m, \ddagger\sigma_{m-1}, \ldots, \ddagger\sigma_1\} \in (\ddagger\Gamma^+)^*$.

We define an EPDA by the septuple (7-tuple)

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Q_\mathrm{F}, \sigma_0)_{\text{where}}$$

- $Q$is a finite set of *states*;
- $\Sigma$is the finite set of the *input alphabet*;
- $\Gamma$is the finite *stack alphabet*;
- $q_0 \in Q$is the *start state*;
- $Q_\mathrm{F} \subseteq Q$is the set of *final states*;
- $\sigma_0 \in \Gamma$is the *initial stack symbol*
- $\delta : Q \times \Sigma \times \Gamma \to S$is the *transition function*, where $S$are finite subsets of $Q \times (\ddagger\Gamma^+)^* \times \Gamma^* \times (\ddagger\Gamma^+)^*$.

Thus the transition function takes a state, the next symbol of the input string, and the top symbol of the current stack and generates the next state, the stacks to be pushed and popped onto the *embedded stack*, the pushing and popping of the current stack, and the stacks to be considered the current stacks in the next transition. More conceptually, the *embedded stack* is pushed and popped, the current stack is optionally pushed back onto the *embedded stack*, and any other stacks one would like are pushed on top of that, with the last stack being the one read from in the next iteration. Therefore, stacks can be pushed both above and below the current stack.

A given configuration is defined by

$$C(M) = \{q, \Upsilon_m \ldots \Upsilon_1, x_1, x_2\} \in Q \times (\ddagger\Gamma^+)^* \times \Sigma^* \times \Sigma^*$$

where $q$is the current state, the Υs are the stacks in the *embedded stack*, with $\Upsilon_m$the current stack, and for an input string $x = x_1x_2 \in \Sigma^*$, $x_1$is the portion of the string already processed by the machine and $x_2$is the portion to be processed, with its head being the current symbol read. Note that the empty string $\epsilon \in \Sigma$is implicitly defined as a terminating symbol, where if the machine is at a final state when the empty string is read, the entire input string is *accepted*, and if not it is *rejected*. Such *accepted* strings are elements of the language

$$L(M) = \left\{x | \{q_0, \Upsilon_0, \epsilon, x\} \to^*_M \{q_\mathrm{F}, \Upsilon_m \ldots \Upsilon_1, x, \epsilon\}\right\}$$

where $q_\mathrm{F} \in Q_\mathrm{F}$and $\to^*_M$defines the transition function applied over as many times as necessary to parse the string.

Chapter 16

Quantum Dot Cellular Automaton

Quantum Dot Cellular Automata (sometimes referred to simply as quantum cellular automata, or QCA) are proposed models of quantum computation, which have been devised in analogy to conventional models of cellular automata introduced by von Neumann.

Background

Any device designed to represent data and perform computation, regardless of the physics principles it exploits and materials used to build it, must have two fundamental properties: distinguishability and conditional change of state, the latter implying the former. This means that such a device must have barriers that make it possible to distinguish between states, and that it must have the ability to control these barriers to perform conditional change of state. For example, in a digital electronic system, transistors play the role of such controllable energy barriers, making it extremely practical to perform computing with them.

Cellular automata

A cellular automaton (CA) is a finite state machine consisting of a uniform (finite or infinite) grid of cells. Each one of these cells can only be in one of a finite number of states at a discrete time. The state of each cell in this grid is determined by the state of its adjacent cells, also called the cell's "neighborhood." The most popular example of a cellular automaton was presented by John Horton Conway in 1970, which he named "The Game of Life."

Quantum-dot cells

Origin

Cellular automata are commonly implemented as software programs. However, in 1993, Lent et al. proposed a physical implementation of an automaton using quantum-dot cells. The automaton quickly gained popularity and it was first fabricated in 1997. Lent combined the discrete nature of both cellular automata and quantum mechanics, to create nano-scale devices capable of performing computation at very high switching speeds and consuming extremely small amounts of electrical power.

Modern cells

Today, standard solid state QCA cell design considers the distance between quantum dots to be about 20 nm, and a distance between cells of about 60 nm. Just like any CA, Quantum (-dot) Cellular Automata are based on the simple interaction rules between cells placed on a grid. A QCA cell is constructed from four quantum dots arranged in a square pattern. These quantum dots are sites electrons can occupy by tunneling to them.

Theory behind cell

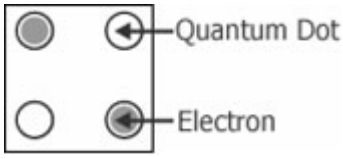


Figure 2 - A simplified diagram of a four-dot QCA cell.

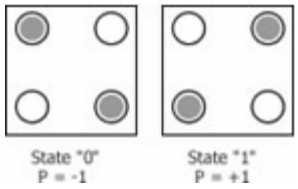


Figure 3 - The two possible states of a four-dot QCA cell.

Figure 2 shows a simplified diagram of a quantum-dot cell. If the cell is charged with two electrons, each free to tunnel to any site in the cell, these electrons will try to occupy the furthest possible site with respect to each other due to mutual electrostatic repulsion. Therefore, two distinguishable cell states exist. Figure 3 shows the two possible minimum energy states of a quantum-dot cell. The state of a QCA represents its polarization, denoted as P. Although arbitrary in meaning, using cell polarization P = -1 to represent logic "0" and P = +1 to represent logic "1" has become standard practice.

Grid arrangements

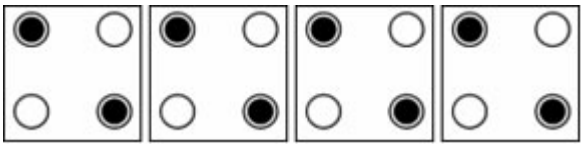


Figure 4 - A wire of quantum-dot cells.

Grid arrangements of quantum-dot cells behave in a ways that allow for computation. The simplest practical cell arrangement is given by placing quantum-dot cells in series, to the side of each other. Figure 4 shows such an arrangement of four quantum-dot cells. The bounding boxes in the figure do not represent physical implementation, but are shown as means to identify individual cells.

If the polarization of any of the cells in the arrangement shown in figure 4 were to be controllable (driver cell), the rest of the cells would immediately synchronize to its polarization due to Coulombic interactions between them; much like an instantaneous chain reaction. In this way, a wire of quantum-dot cells is realizable. Although the ability to realize conductive wires does not alone provide the means to perform computation, a complete set of universal logic gates can be constructed using the same principle.

Logic gates

Majority gate

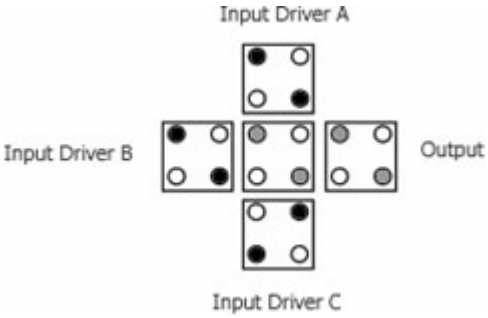


Figure 5 - QCA Majority Gate

The fundamental logic gate in QCA is the majority gate. Figure 5 shows a majority gate with three inputs and one output. Assuming inputs A and B exist in a “binary 0” state and input C exists in a “binary 1” state, the output will exist in a “binary 0” state as the conjunct electrical field effect of inputs A and B is greater than the one of input C. In other words, the majority gate drives the output cell’s state to be equal to that of the majority of the inputs. Now, if the polarization of input C were to be fixed to say, binary 0, the only way the output’s state becomes binary 1, is if input A and B are also 1. Otherwise, the output cell will exhibit a binary 0 state.

Other gates

This conditional behavior is exactly the same as that of an AND gate. Similarly, an OR gate can be constructed using a majority gate with fixed polarization equivalent to binary 1 at one of its inputs. In this way, if any or both of the remaining inputs exist in the binary 1 state, the output will be also in a binary 1 state. Although not certainly based on a majority gate structure, a NOT gate is just as easily realizable. The key principle behind its functionality lies on the fact that placing a cell at 45 degrees with respect of a pair of cells of same polarity, the polarization of the cell will become opposite to that of its driving pair. Figure 6 shows a standard implementation of a NOT logic gate.

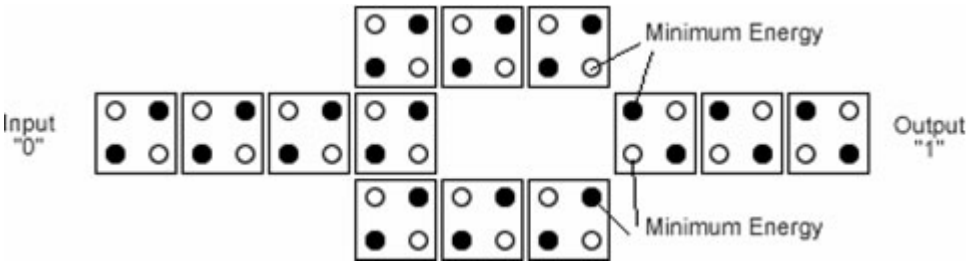


Figure 6 - Standard Implementation of a NOT gate.

State transition

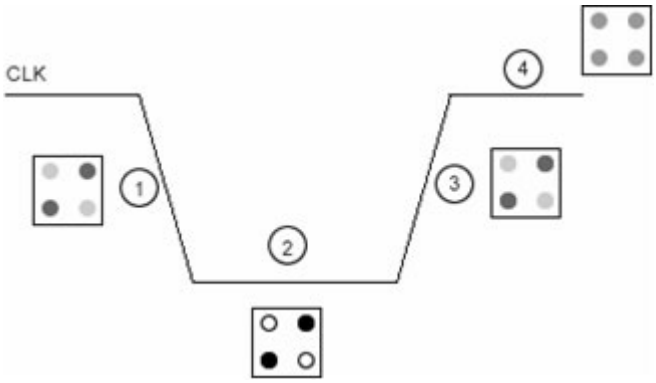


Figure 7 - The QCA clock, its stages and its effects on a cell's energy barriers.

There is a connection between quantum-dot cells and cellular automata. Cells can only be in one of 2 states and the conditional change of state in a cell is dictated by the state of its adjacent neighbors. However, a method to control data flow is necessary to define the direction in which state transition occurs in QCA cells. The clocks of a QCA system serve two purposes: powering the automaton, and controlling data flow direction. QCA clocks are areas of conductive material under the automaton's lattice, modulating the electron tunneling barriers in the QCA cells above it.

Four stages

A QCA clock induces four stages in the tunneling barriers of the cells above it. In the first stage, the tunneling barriers start to rise. The second stage is reached when the tunneling barriers are high enough to prevent electrons from tunneling. The third stage occurs when the high barrier starts to lower. And finally, in the fourth stage, the tunneling barriers allow electrons to freely tunnel again. In simple words, when the clock signal is high, electrons are free to tunnel. When the clock signal is low, the cell becomes latched.

Figure 7 shows a clock signal with its four stages and the effects on a cell at each clock stage. A typical QCA design requires four clocks, each of which is cyclically 90 degrees out of phase with the prior clock. If a horizontal wire consisted of say, 8 cells and each consecutive pair, starting from the left were to be connected to each consecutive clock, data would naturally flow from left to right. The first pair of cells will stay latched until

the second pair of cells gets latched and so forth. In this way, data flow direction is controllable through clock zones.

Wire-crossing

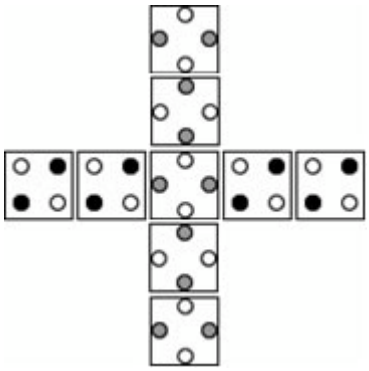


Figure 8 - Basic Wire-Crossing Technique.

Wire-crossing in QCA cells is done by using a "plus-sign" pattern, as shown in figure 8. The distances between a plus-sign pattern and a square pattern are exactly the same, allowing for the same Coulombic interactions between electrons in a cell. Thus, when a wire of square cells crosses a wire of plus-sign cells, they do not interact, thus the signals on each wire are preserved.

Fabrication problem

Although this technique is rather simple, it represents an enormous fabrication problem. A new kind of cell pattern potentially introduces as much as twice the amount of fabrication cost and infrastructure; the number of possible quantum dot locations on an interstitial grid is doubled and an overall increase in geometric design complexity is inevitable. Yet another problem this technique presents is that the additional space between cells of the same orientation decreases the energy barriers between a cells ground state and a cell's first excited state. This degrades the performance of the device in terms of maximum operating temperature, resistance to entropy and switching speed.

Crossbar Network

A different wire-crossing technique, which makes fabrication of QCA devices more practical, was presented by Christopher Graunke, David Wheeler, Douglas Tougaw, and Jeffrey D. Will, in their paper "Implementation of a crossbar network using quantum-dot cellular automata". The paper not only presents a new method of implementing wire-crossings, but it also gives a new perspective on QCA clocking.

Their wire-crossing technique introduces the concept of implementing QCA devices capable of performing computation as a function of synchronization. This implies the ability to modify the device's function through the clocking system without making any physical changes to the device. Thus, the fabrication problem stated earlier is fully

addressed by: a) using only one type of quantum-dot pattern and, b) by the ability to make a universal QCA building block of adequate complexity, which function is determined only by its timing mechanism (i.e. its clocks).

Quasi-adiabatic switching, however, requires that the tunneling barriers of a cell be switched relatively slowly compared to the intrinsic switching speed of a QCA. This prevents ringing and metastable states observed when cells are switched abruptly. Therefore, the switching speed of a QCA is limited not by the time it takes for a cell to change polarization, but by the appropriate quasi-adiabatic switching time of the clocks being used.

Parallel to Serial

When designing a device capable of computing, it is often necessary to convert parallel data lines into a serial data stream. This conversion allows different pieces of data to be reduced to a time-dependent series of values on a single wire. Figure 9 shows such a parallel-to-serial conversion QCA device. The numbers on the shaded areas represent different clocking zones at consecutive 90-degree phases. Notice how all the inputs are on the same clocking zone. If parallel data were to be driven at the inputs A, B, C and D, and then driven no more for at least the remaining 15 serial transmission phases, the output X would present the values of D, C, B and A –in that order, at phases three, seven, eleven and fifteen. If a new clocking region were to be added at the output, it could be clocked to latch a value corresponding to any of the inputs by correctly selecting an appropriate state-locking period.

The new latching clock region would be completely independent from the other four clocking zones illustrated in figure 9. For instance, if the value of interest to the new latching region were to be the value that D presents every 16th phase, the clocking mechanism of the new region would have to be configured to latch a value in the 4th phase and every 16th phase from then on, thus, ignoring all inputs but D.

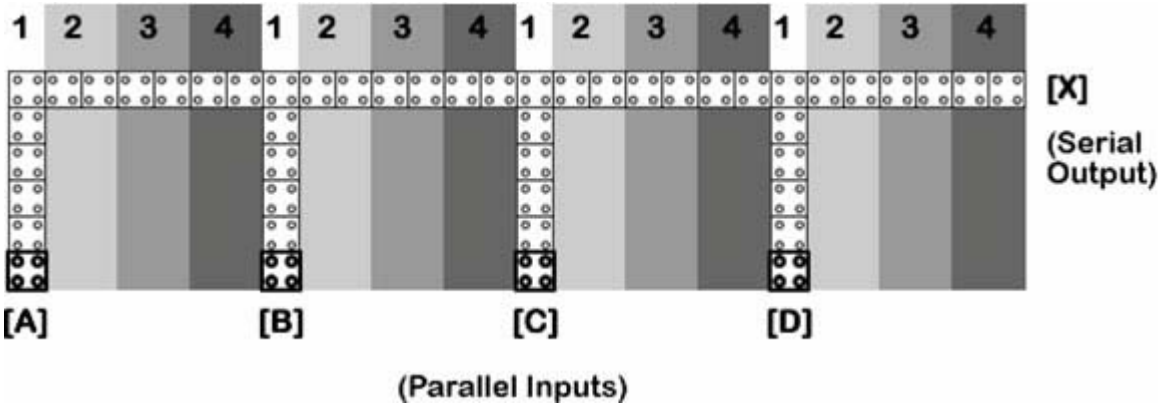


Figure 9 - Parallel to serial conversion.

Additional serial lines

Adding a second serial line to the device, and adding another latching region would allow for the latching of two input values at the two different outputs. To perform computation, a gate that takes as inputs both serial lines at their respective outputs is added. The gate is placed over a new latching region configured to process data only when both latching regions at the end of the serial lines hold the values of interest at the same instant. Figure 10 shows such an arrangement. If correctly configured, latching regions 5 and 6 will each hold input values of interest to latching region 7. At this instant, latching region 7 will let the values latched on regions 5 and 6 through the AND gate, thus the output could be configured to be the AND result of any two inputs (i.e. R and Q) by merely configuring the latching regions 5, 6 and 7.

This represents the flexibility to implement 16 functions, leaving the physical design untouched. Additional serial lines and parallel inputs would obviously increase the number of realizable functions. However, a significant drawback of such devices is that, as the number of realizable functions increases, an increasing number of clocking regions is required. As a consequence, a device exploiting this method of function implementation may perform significantly slower than its traditional counterpart.

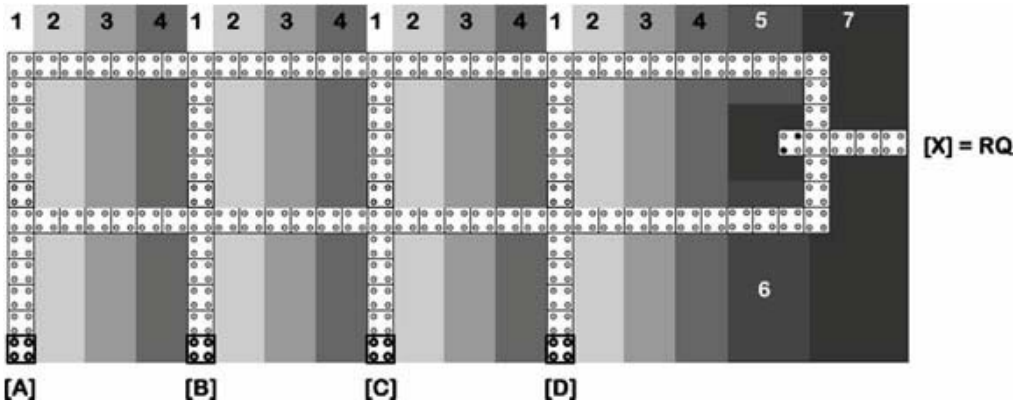


Figure 10 – Multifunction QCA Device.

Fabrication

Generally speaking, there are four different classes of QCA implementations: Metal-Island, Semiconductor, Molecular, and Magnetic.

Metal-Island

The Metal-Island implementation was the first fabrication technology created to demonstrate the concept of QCA. It was not originally intended to compete with current technology in the sense of speed and practicality, as its structural properties are not suitable for scalable designs. The method consists of building quantum dots using aluminum islands. Earlier experiments were implemented with metal islands as big as 1

micrometer in dimension. Because of the relatively large-sized islands, Metal-Island devices had to be kept at extremely low temperatures for quantum effects (electron switching) to be observable.

Semiconductor

Semiconductor (or solid state) QCA implementations could potentially be used to implement QCA devices with the same highly advanced semiconductor fabrication processes used to implement CMOS devices. Cell polarization is encoded as charge position, and quantum-dot interactions rely on electrostatic coupling. However, current semiconductor processes have not yet reached a point where mass production of devices with such small features (~20 nanometers) is possible. Serial lithographic methods, however, make QCA solid state implementation achievable, but not necessarily practical. Serial lithography is slow, expensive and unsuitable for mass-production of solid-state QCA devices. Today, most QCA prototyping experiments are done using this implementation technology.

Molecular

A proposed but not yet implemented method is method consists of building QCA devices out of single molecules. The main advantages of such implementations include: highly symmetric QCA cell structure, very high switching speeds, extremely high device density, operation at room temperature, and even the possibility of mass-producing devices by means of self-assembly. A number of technical challenges, including choice of molecules, the design of proper interfacing mechanisms, and clocking technology remain to be solved before this method can be implemented.

Magnetic

Magnetic QCA –commonly referred to as MQCA (or QCA: M), is based on the interaction between magnetic nanoparticles. The magnetization vector of these nanoparticles is analogous to the polarization vector in all other implementations. In MQCA, the term “Quantum” refers to the quantum-mechanical nature of magnetic exchange interactions and not to the electron-tunneling effects. Devices constructed this way could operate at room temperature.

Improvement over CMOS

Complementary metal-oxide semiconductor (CMOS) technology has been the industry standard for implementing Very Large Scale Integrated (VLSI) devices for the last two decades, mainly due to the consequences of miniaturization of such devices (i.e. increasing switching speeds, increasing complexity and decreasing power consumption). Quantum Cellular Automata (QCA) is only one of the many alternative technologies proposed as a replacement solution to the fundamental limits CMOS technology will impose in the years to come.

Although QCA solves most of the limitations of CMOS technology, it also brings its own. Research suggests that intrinsic switching time of a QCA cell is at best in the order of terahertz. However, the actual speed may be much lower, in the order of megahertz for solid state QCA and gigahertz for molecular QCA, due to the proper quasi-adiabatic clock switching frequency setting. Additionally, solid-state QCA devices cannot operate at room temperature. The only alternative to this temperature limitation is the recently proposed “Molecular QCA” which theoretically has an inter-dot distance of 2 nm and an inter-cell distance of 6 nm. Molecular QCA is also considered to be the only feasible implementation method for mass production of QCA devices.

Chapter 17

Finite-state Machine

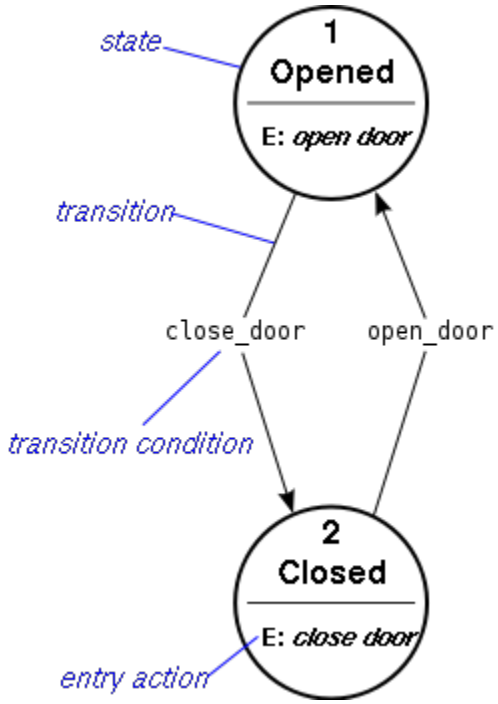


Fig. 1 Example of a simple finite state machine

A **finite-state machine** (**FSM**) or **finite-state automaton** (plural: *automata*), or simply a **state machine**, is a mathematical abstraction sometimes used to design digital logic or computer programs. It is a behavior model composed of a finite number of states, transitions between those states, and actions, similar to a flow graph in which one can inspect the way logic runs when certain conditions are met. It has finite internal memory, an input feature that reads symbols in a sequence, one at a time without going backward; and an output feature, which may be in the form of a user interface, once the model is implemented. The operation of an FSM begins from one of the states (called a *start state*), goes through transitions depending on input to different states and can end in any of those available, however only a certain set of states mark a successful flow of operation (called *accept states*).

Finite-state machines can solve a large number of problems, among which are electronic design automation, communication protocol design, parsing and other engineering applications. In biology and artificial intelligence research, state machines or hierarchies of state machines are sometimes used to describe neurological systems and in linguistics—to describe the grammars of natural languages.

Concepts and vocabulary

A current *state* is determined by past states of the system. As such, it can be said to record information about the past, i.e., it reflects the input changes from the system start to the present moment. The number and names of the states typically depend on the different possible states of the memory, e.g. if the memory is three bits long, there are 8 possible states. A *transition* indicates a state change and is described by a condition that would need to be fulfilled to enable the transition. An *action* is a description of an activity that is to be performed at a given moment. There are several action types:

Entry action
: which is performed *when entering* the state

Exit action
: which is performed *when exiting* the state

Input action
: which is performed depending on present state and input conditions

Transition action
: which is performed when performing a certain transition

An FSM can be represented using a state diagram (or state transition diagram) as in figure 1 above. Besides this, several state transition table types are used. The most common representation is shown below: the combination of current state (e.g. B) and input (e.g. Y) shows the next state (e.g. C). The complete actions information can be added only using footnotes. An FSM definition including the full actions information is possible using state tables.

State transition table

Current state → **Input ↓**	**State A**	**State B**	**State C**
Input X	...	...	...
Input Y	...	State C	...
Input Z	...	...	...

In addition to their use in modeling reactive systems presented here, finite state automata are significant in many different areas, including electrical engineering, linguistics, computer science, philosophy, biology, mathematics, and logic. Finite state machines are a class of automata studied in automata theory and the theory of computation. In computer science, finite state machines are widely used in modeling of application

behavior, design of hardware digital systems, software engineering, compilers, network protocols, and the study of computation and languages.

Classification

There are two different groups of state machines: Acceptors/Recognizers and Transducers.

Acceptors and recognizers

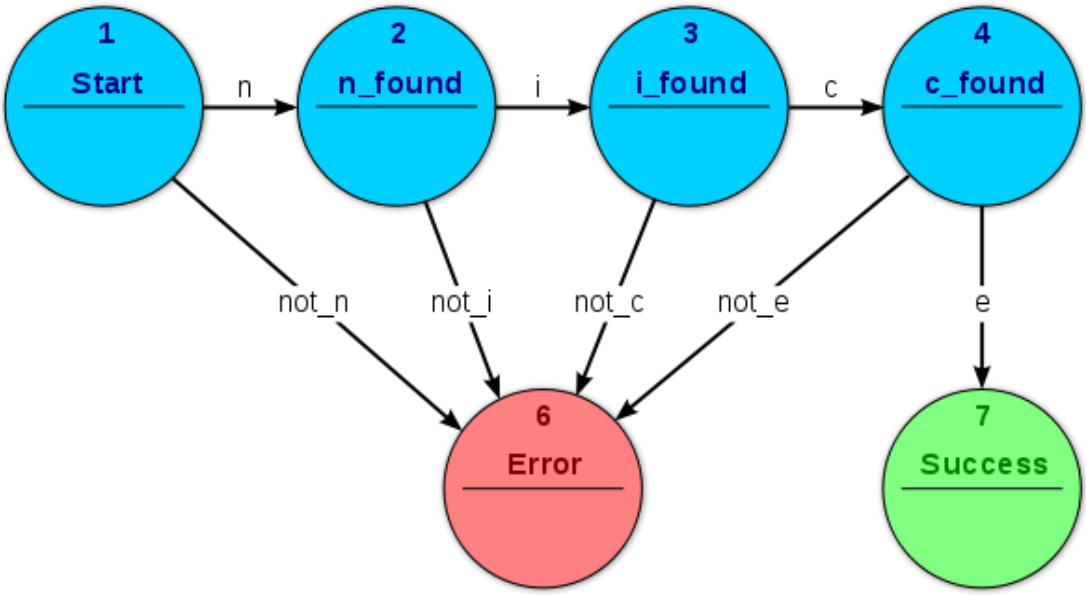


Fig. 2 Acceptor FSM: parsing the word "nice"

Acceptors and **recognizers** (also **sequence detectors**) produce a binary output, saying either *yes* or *no* to answer whether the input is accepted by the machine or not. All states of the FSM are said to be either accepting or not accepting. At the time when all input is processed, if the current state is an accepting state, the input is accepted; otherwise it is rejected. As a rule the input are symbols (characters); actions are not used. The example in figure 2 shows a finite state machine which accepts the word "nice". In this FSM the only accepting state is number 7.

The machine can also be described as defining a language, which would contain every word accepted by the machine but none of the rejected ones; we say then that the language is *accepted* by the machine. By definition, the languages accepted by FSMs are the regular languages—that is, a language is regular if there is some FSM that accepts it.

Start state

The start state is usually shown drawn with an arrow "pointing at it from any where".

Accept (or final) states

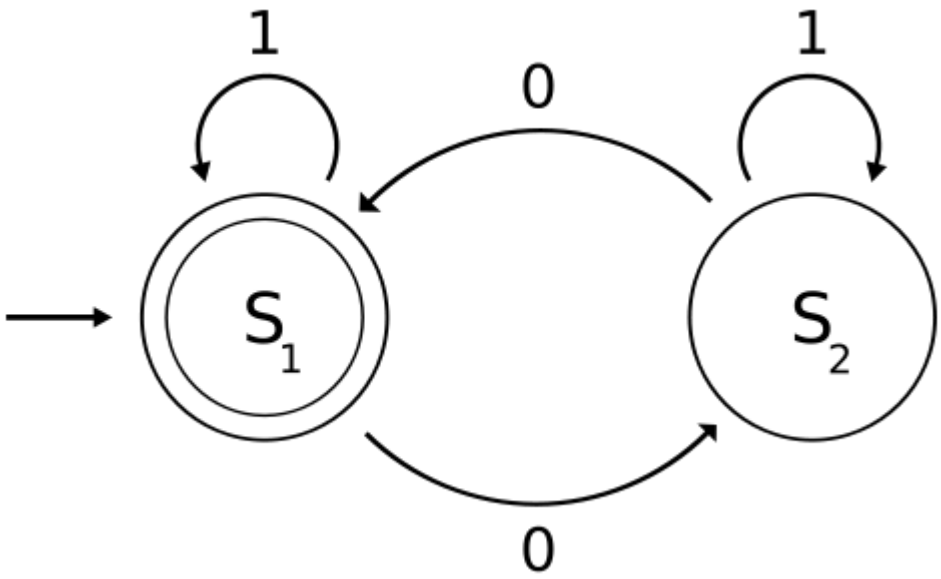


Fig. 3: Representation of a finite-state machine; this example shows one that determines whether a binary number has an odd or even number of 0's, where S_1 is an **accepting state**.

Accept states (also referred to as **accepting** or **final** states) are those at which the machine reports that the input string, as processed so far, is a member of the language it accepts. It is usually represented by a double circle.

An example of an accepting state appears in the diagram to the right: a deterministic finite automaton (DFA) that detects whether the binary input string contains an even number of 0's.

S_1 (which is also the start state) indicates the state at which an even number of 0's has been input. S_1 is therefore an accepting state. This machine will finish in an accept state, if the binary string contains an even number of 0's (including any binary string containing no 0's). Examples of strings accepted by this DFA are epsilon (the empty string), 1, 11, 11..., 00, 010, 1010, 10110, etc...

Transducers

Transducers generate output based on a given input and/or a state using actions. They are used for control applications and in the field of computational linguistics. Here two types are distinguished:

Moore machine

The FSM uses only entry actions, i.e., output depends only on the state. The advantage of the Moore model is a simplification of the behaviour. Consider an elevator door. The state machine recognizes two commands: "command_open" and "command_close" which trigger state changes. The entry action (E:) in state "Opening" starts a motor opening the door, the entry action in state "Closing" starts a motor in the other direction closing the door. States "Opened" and "Closed" stop the motor when fully opened or closed. They signal to the outside world (e.g., to other state machines) the situation: "door is open" or "door is closed".

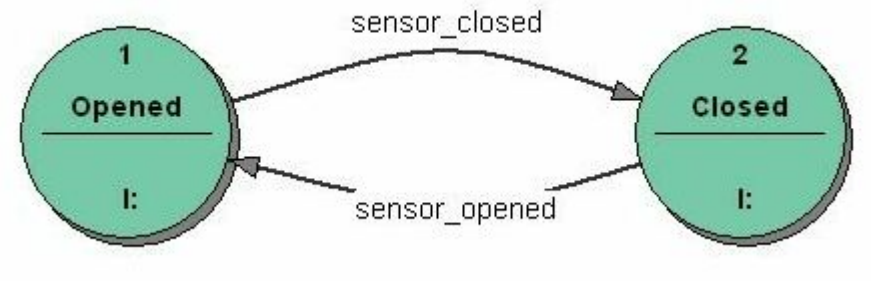


Fig. 4 Transducer FSM: Mealy model example

Mealy machine

The FSM uses only input actions, i.e., output depends on input and state. The use of a Mealy FSM leads often to a reduction of the number of states. The example in figure 4 shows a Mealy FSM implementing the same behaviour as in the Moore example (the behaviour depends on the implemented FSM execution model and will work, e.g., for virtual FSM but not for event driven FSM). There are two input actions (I:): "start motor to close the door if command_close arrives" and "start motor in the other direction to open the door if command_open arrives". The "opening" and "closing" intermediate states are not shown.

In practice mixed models are often used.

More details about the differences and usage of Moore and Mealy models, including an executable example, can be found in the external technical note "Moore or Mealy model?"

Determinism

A further distinction is between **deterministic** (DFA) and **non-deterministic** (NFA, GNFA) automata. In deterministic automata, every state has exactly one transition for each possible input. In non-deterministic automata, an input can lead to one, more than one or no transition for a given state. This distinction is relevant in practice, but not in theory, as there exists an algorithm (the powerset construction) which can transform any NFA into a more complex DFA with identical functionality.

The FSM with only one state is called a combinatorial FSM and uses only input actions. This concept is useful in cases where a number of FSM are required to work together, and where it is convenient to consider a purely combinatorial part as a form of FSM to suit the design tools.

UML state machines

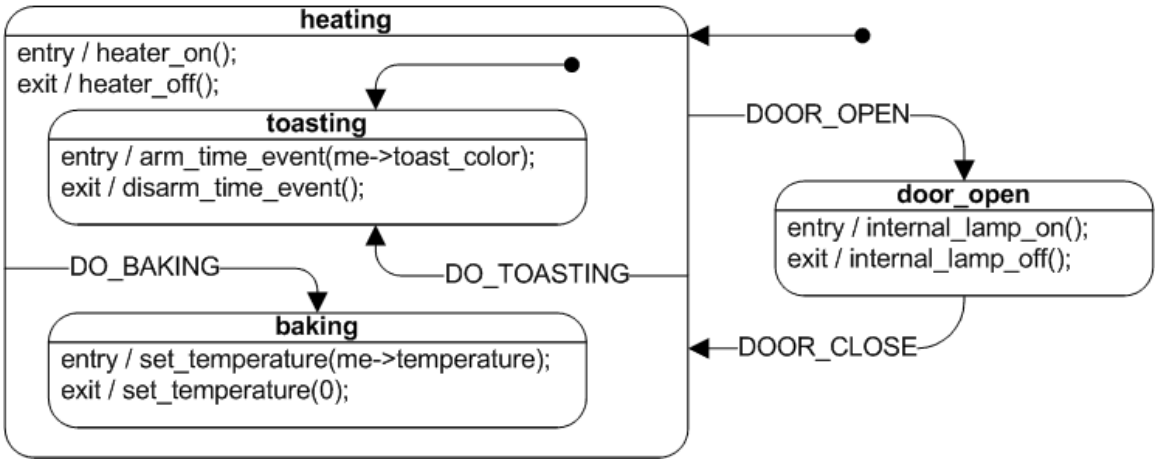


Fig. 5 UML state machine example (a toaster oven)

The Unified Modeling Language has a very rich semantics and notation for describing state machines. UML state machines overcome the limitations of traditional finite state machines while retaining their main benefits. UML state machines introduce the new concepts of hierarchically nested states and orthogonal regions, while extending the notion of actions. UML state machines have the characteristics of both Mealy machines and Moore machines. They support actions that depend on both the state of the system and the triggering event, as in Mealy machines, as well as entry and exit actions, which are associated with states rather than transitions, as in Moore machines.

Alternative semantics

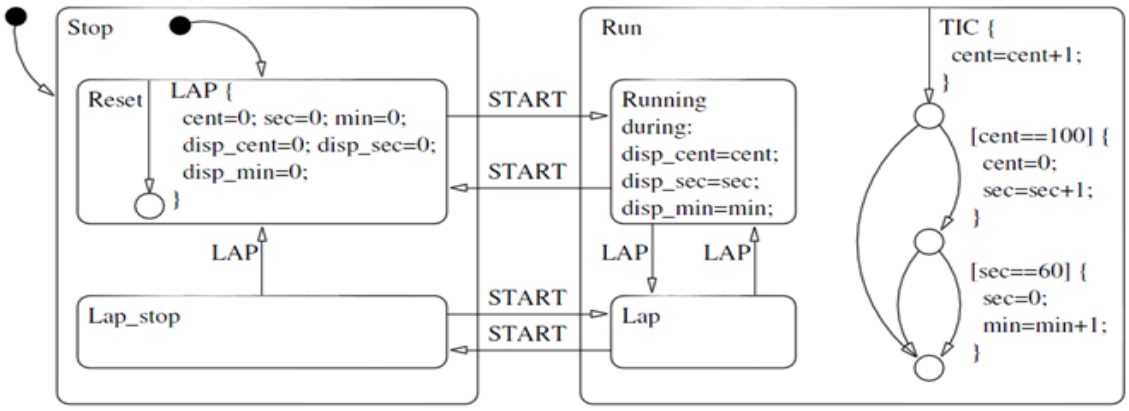


Fig. 6 Model of a simple stopwatch

There are other sets of semantics available to represent state machines. For example, there are tools for modeling and designing logic for embedded controllers. They combine hierarchical state machines, flow graphs, and truth tables into one language, resulting in a different formalism and set of semantics. Figure 6 illustrates this mix of state machines and flow graphs with a set of states to represent the state of a stopwatch and a flow graph to control the ticks of the watch. These charts, like Harel's original state machines, support hierarchically nested states, orthogonal regions, state actions, and transition actions.

FSM logic

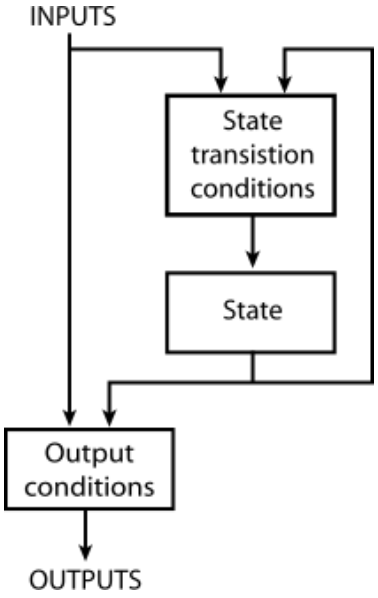


Fig. 7 FSM Logic (Mealy)

The next state and output of an FSM is a function of the input and of the current state. The FSM logic is shown in Figure 7.

Mathematical model

In accordance to the general classification, the following formal definitions are found:

- A *deterministic finite state machine* or *acceptor deterministic finite state machine* is a quintuple $(\Sigma, S, s_0, \delta, F)$, where:
  - Σ is the input alphabet (a finite, non-empty set of symbols).
  - S is a finite, non-empty set of states.
  - s_0 is an initial state, an element of S.
  - δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$ (in a nondeterministic finite state machine it would be $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$, i.e., δ would return a set of states).
  - F is the set of final states, a (possibly empty) subset of S.

For both deterministic and non-deterministic FSMs, it is conventional to allow δ to be a partial function, i.e. δ(*q*,*x*) does not have to be defined for every combination of $q \in S$ and $x \in \Sigma$. If an FSM *M* is in a state *q*, the next symbol is *x* and δ(*q*,*x*) is not defined, then *M* can announce an error (i.e. reject the input).

- A *finite state transducer* is a sextuple (Σ,Γ,*S*,s_0,δ,ω), where:
  - Σ is the input alphabet (a finite non empty set of symbols).
  - Γ is the output alphabet (a finite, non-empty set of symbols).
  - *S* is a finite, non-empty set of states.
  - s_0 is the initial state, an element of *S*. In a nondeterministic finite state machine, s_0 is a set of initial states.
  - δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$.
  - ω is the output function.

If the output function is a function of a state and input alphabet ($\omega : S \times \Sigma \rightarrow \Gamma$) that definition corresponds to the **Mealy model**, and can be modelled as a Mealy machine. If the output function depends only on a state ($\omega : S \rightarrow \Gamma$) that definition corresponds to the **Moore model**, and can be modelled as a Moore machine. A finite-state machine with no output function at all is known as a semiautomaton or transition system.

Optimization

Optimizing an FSM means finding the machine with the minimum number of states that performs the same function. The fastest known algorithm doing this is the Hopcroft minimization algorithm. Other techniques include using an implication table, or the Moore reduction procedure. Additionally, acyclic FSAs can be optimized using a simple bottom up algorithm.

Implementation

Hardware applications

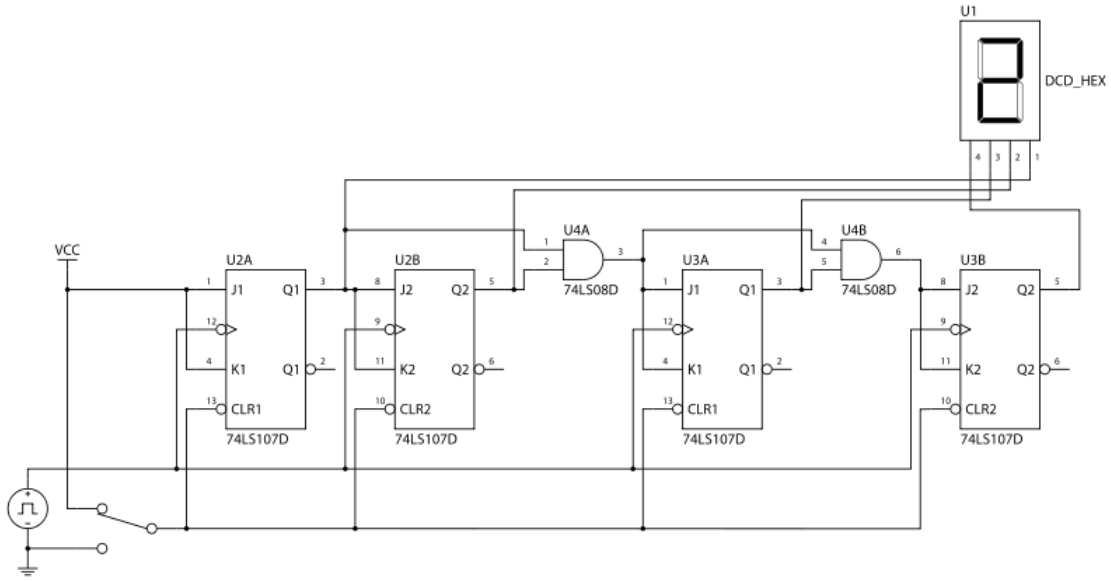


Fig. 8 The circuit diagram for a 4-bit TTL counter, a type of state machine

In a digital circuit, an FSM may be built using a programmable logic device, a programmable logic controller, logic gates and flip flops or relays. More specifically, a hardware implementation requires a register to store state variables, a block of combinational logic which determines the state transition, and a second block of combinational logic that determines the output of an FSM. One of the classic hardware implementations is the Richards controller.

Mealy and Moore machines produce logic with asynchronous output, because there is a propagation delay between the flip-flop and output. This causes slower operating frequencies in FSM. A Mealy or Moore machine can be convertible to a FSM which output is directly from a flip-flop, which makes the FSM run at higher frequencies. This kind of FSM is sometimes called Medvedev FSM. A counter is the simplest form of this kind of FSM.

Software applications

The following concepts are commonly used to build software applications with finite state machines:

- Automata-based programming
- Event driven FSM
- Virtual FSM (VFSM)

Chapter 18

Krohn–Rhodes Theory and McNaughton's Theorem

Krohn–Rhodes theory

In mathematics and computer science, the **Krohn–Rhodes theory** is an approach to the study of finite semigroups and automata that seeks to decompose them in terms of elementary components. These components correspond to finite aperiodic semigroups and finite simple groups that are combined together in a feedback-free manner (called a "wreath product" or "cascade").

Krohn and Rhodes found a general decomposition for finite automata. In doing their research, though, the authors discovered and proved an unexpected major result in finite semigroup theory, revealing a deep connection between finite automata and semigroups.

Definition

A semigroup S that is a homomorphic image of a subsemigroup of T is said to be a *divisor* of T. The **Krohn–Rhodes theorem for semigroups** states that every finite semigroup S is a divisor of a finite alternating wreath product of finite simple groups, each a divisor of S, and finite aperiodic semigroups (which contain no nontrivial subgroups).

In the automata formulation, given a finite automaton A with states Q and input set I, output alphabet U, then one can expand the states to Q' such that the new automaton A' embeds into a cascade of "simple", irreducible automata: In particular, A is emulated by a cascade of automata whose transitions semigroups are either (1) finite simple groups or (2) subsemigroups of the flip-flop. The new automaton A' has the same input and output symbols as A. Here, both the states and inputs of the cascaded automata have a very special hierarchical coordinate form.

Moreover, each simple group (*prime*) or non-group irreducible semigroup (subsemigroup of the flip-flop monoid) that divides the transformation semigroup of A must divide the transition semigroup of some component of the cascade, and only the primes that must occur as divisors of the components are those that divide A 's transition semigroup.

Group complexity

The **Krohn–Rhodes complexity** (also called **group complexity** or just **complexity**) of a finite semigroup S is the least number of groups in a wreath product of finite groups and finite aperiodic semigroups of which S is a divisor.

All finite aperiodic semigroups have complexity 0, while non-trivial finite groups have complexity 1. In fact, there are semigroups of every non-negative integer complexity. For example, for any n greater than 1, the multiplicative semigroup of all $(n+1)\times(n+1)$ upper triangular matrices over any fixed finite field has complexity n (Kambites, 2007).

A major open problem in finite semigroup theory is the *decidability of complexity*: given the multiplication table for a finite semigroup, is there an algorithm that will compute its Krohn-Rhodes complexity? Rhodes has conjectured that the problem is decidable.

History and Applications

At a conference in 1962, Kenneth Krohn and John Rhodes announced a method for decomposing a (deterministic) finite automaton into "simple" components that are themselves finite automata. This joint work, which has implications for philosophy, lead Harvard University to grant a PhD to Krohn, and lead MIT to grant a PhD to Rhodes. Simpler proofs, and generalizations of the theorem to infinite structures, have been published since then.

In the 1965 paper by Krohn and Rhodes, the proof of the theorem on the decomposition of finite automata (or, equivalently sequential machines) made extensive use of the algebraic semigroup structure. Later proofs contained major simplifications using finite wreath products of finite transformation semigroups. The theorem generalizes the Jordan–Hölder decomposition for finite groups (in which the primes are the finite simple groups), to all finite transformation semigroups (for which the primes are again the finite simple groups plus all subsemigroups of the so-called "flip-flop"). Both the group and more general finite automata decomposition require expanding the state-set of the general, but allow for the same number of input symbols. In the general case, these are embedded in a larger structure with a hierarchical "coordinate system".

Some confusion occurs for automata theorists in that Krohn and Rhodes explicitly refer to their theorem as a "prime decomposition theorem" for automata. The components in the decomposition, however, were not prime automata (with *prime* defined in a naïve way); rather, the notion of *prime* is more sophisticated and algebraic: the semigroups and groups associated to the constituent automata of the decomposition are prime (or irreducible) in a strict and natural algebraic sense (Eilenberg, 1976). Also, unlike earlier decomposition theorems, the Krohn–Rhodes decompositions usually require expansion of the state-set, so that the expanded automaton covers (emulates) the one being decomposed. These facts have made the theorem difficult to understand, and challenging to apply in a practical way—until recently, when computational implementations became available.

H.P. Zeiger (1967) proved an important variant called the **Holonomy Decomposition** (Eilenberg 1976). The holonomy method appears to be relatively efficient and has been implemented computationally by A. Egri-Nagy (Egri-Nagy & Nehaniv 2005).

Meyer and Thompson (1969) give a version of Krohn-Rhodes decomposition for finite automata that is equivalent to the decomposition previously developed by Hartmanis and Stearns, but for useful decompositions, the notion of *expanding* the state-set of the original automaton is essential (for the non-permutation automata case).

Many proofs and constructions now exist of Krohn–Rhodes Decompositions (e.g., [Krohn, Rhodes & Tilson 1968], [Ésik 2000]), with the holonomy method the most popular and efficient in general (although not in all cases). Due to the close relation between monoids and categories, a version of the Krohn–Rhodes theorem is applicable to category theory. This observation and a proof of an analogous result were offered by Wells (1980).

The Krohn–Rhodes theorem for semigroups/monoids is an analogue of the Jordan–Hölder theorem for finite groups (for semigroups/monoids rather than groups). As such, the theorem is a deep and important result in semigroup/monoid theory. The theorem was also surprising to many mathematicians, since it had previously been widely believed that the semigroup/monoid axioms were too weak to admit a structure theorem of any strength.

Work by Egri-Nagy and Nehaniv continues to further automate the holonomy version of the Krohn-Rhodes decomposition extended with the related decomposition for finite groups (so-called Frobenius-Lagrange coordinates) using the computer algebra system GAP.

Applications outside of the semigroup and monoid theories are now computationally feasible. They include computations in biology and biochemical systems (e.g. Egri-Nagy & Nehaniv 2008]), artificial intelligence, finite-state physics, psychology, and game theory.

McNaughton's Theorem

In automata theory, **McNaughton's theorem** refers to a theorem which demonstrates that the set of ω-regular languages is identical to the set of languages recognizable by deterministic Muller automata. The theorem also demonstrates that Büchi automata and deterministic Muller automata are equal in expressive power.

Original statement

In McNaughton's original paper, the theorem was stated as:

"An ω-event is regular if and only if it is finite-state."

In modern terminology, *ω-events* are commonly referred to as ω-languages. Following McNaughton's definition, an ω-event is a *finite-state event* if there exists a deterministic Muller automaton that recognizes it.

Constructing an ω-regular language from a deterministic Muller automaton

One direction of the theorem can be proven by showing that any given Muller automaton recognizes an ω-regular language. Suppose $A = (Q,\Sigma,\delta,q_0,\mathbf{F})$ is a deterministic Muller automaton. The union of finitely many ω-regular languages produces an ω-regular language, therefore it can be assumed *w.l.o.g.* that the Muller acceptance condition **F** contains exactly one set of states $\{q_1, ... ,q_n\}$. Let α be the regular language whose elements will take A from q_0 to q_1. For $1 \leq i \leq n$, let β_i be a regular language whose elements take A from q_i to $q_{(i \bmod n)+1}$ without passing through any state outside of $\{q_1, ... ,q_n\}$. It is claimed that $\boldsymbol{\alpha(\beta_1 ... \beta_n)^\omega}$ is the ω-regular language recognized by the Muller automaton A. It is proved as follows.

Suppose w is a word accepted by A. Let ρ be the run which led to the acceptance of w. Create an infinite and strictly increasing sequence of integers $t_1, t_2, ...$, which are the times in the run ρ. Also for each a and b, the time t_{na+b} of ρ contains the state q_b. Such an integer sequence exists because all the states of $\{q_1, ... ,q_n\}$ appear in ρ infinitely often. By the above definitions of α and β's, it can be easily shown that the existence of such an integer sequence implies that w is an element of $\alpha(\beta_1 ... \beta_n)^\omega$.

Suppose $w \in \alpha(\beta_1 ... \beta_n)^\omega$. There can be an initial segment of w that is an element of α and leads A to the state q_1. From there on, the run never assumes a state outside of $\{q_1, ... ,q_n\}$, due to the definitions of β's, and all the states in the set are repeated infinitely often. Therefore, A recognizes the word w.

Constructing a deterministic Muller automaton from a given ω-regular language

The union of finitely many deterministic Muller automata can be easily constructed therefore w.l.o.g. we assume that the given ω-regular language is of the form $\alpha\beta^\omega$. Let's suppose ω-word $w=a_1,a_2,... \in \alpha\beta^\omega$. Let $w(i,j)$ be the finite segment $a_{i+1},...,a_{j-1},a_j$ of w and $w(i)=w(0,i)$. For building a Muller automaton for $\alpha\beta^\omega$, we have to introduce the following two concepts with respect to w.

Favor

A time j *favors* time i if j > i, $w(i) \in \alpha\beta^*$, and $w(i,j) \in \beta^*$.

Equivalence

E(i,j,k), or i is *equivalent* to j as *judged* at time k, if $i,j \leq k$, $w(i) \in \alpha\beta^*$,$w(j) \in \alpha\beta^*$, and for every word x in Σ^*, $w(i,k)x \in \beta^*$ iff $w(j,k)x \in \beta^*$. It is easy to note that if E(i,j,k) then for all k < l, E(i,j,l). In other words, if i and j are ever judged to be equivalent then they will stay equivalent thereafter. And also for the same l, l favors i iff l favors j. Let *E(i,j)* if there exists a k such that E(i,j,k).

Let p be the number of states in the minimum deterministic finite automaton A^{β^*} to recognize language β^*. Now we prove two lemmas about the above two concepts.

Lemma 1

For any time k, among the times i,j < k such that w(i) and w(j) $\in \alpha\beta^*$, the number of equivalence classes induced by E(i,j,k) is bounded by p. Also the number of equivalence classes induced by E(i,j) is bounded by p.

Proof: The finite automaton A^{β^*} is minimum therefore it does not contain equivalent states. Let i and j be such that $w(i)$ and $w(j) \in \alpha\beta^*$ and E(i,j,k). Then, words $w(i,k)$ and $w(j,k)$ will have to take A^{β^*} to the same state starting from the initial state. Hence, first part of lemma is true. The second part is proved by contradiction. Let's suppose there are p+1 times $i_1,...,i_{p+1}$ such that no two of them are equivalent. For $l > \max(i_1,...,i_{p+1})$, we would have, for each m and n, not $E(i_m,i_n,l)$. Therefore, there would be p+1 equivalence classes, as judged at l, contradicting the first part of the lemma.

Lemma 2

$w \in \alpha\beta^{\omega}$ iff there exists a time i such that there are infinitely many times equivalent to i and favoring i.

Proof: Let's suppose $w \in \alpha\beta^{\omega}$ then there exists a strictly increasing sequence of times $i_0,i_1,i_2,...$ such that $w(i_0) \in \alpha$ and $w(i_n,i_{n+1}) \in \beta$. Therefore, for all n > m, $w(i_m,i_n) \in \beta^*$ and i_n favors i_m. So, all the i's are members of one of the finitely many equivalence classes (shown in Lemma 1). So, there must be an infinite subset of all i's which belongs to same class. The smallest member of this subset satisfies the right hand side of this lemma.

Conversely, suppose in w, there are infinitely many times that are equivalent to i and favoring i. From those times, we will construct a strictly increasing and infinite sequence of times $i_0,i_1,i_2,...$ such that $w(i_0) \in \alpha\beta^*$ and $w(i_n,i_{n+1}) \in \beta^*$.Therefore, w would be in $\alpha\beta^{\omega}$. We define this sequence by induction:

Base case: $w(i) \in \alpha\beta^*$ because i is in a equivalence class. So, we set i_0=i. We set i_1 such that i_1 favors i_0 and $E(i_0,i_1)$. So, $w(i_0,i_1) \in \beta^*$.

Induction step: Lets suppose $E(i_0,i_n)$. So, there exists a time i' such that $E(i_0,i_n,i')$. We set i_{n+1} such that $i_{n+1} > i'$, i_{n+1} favors i_0, and $E(i_0,i_{n+1})$. So, $w(i_0,i_{n+1}) \in \beta^*$ and, since $i_{n+1} > i'$ we have by definition of $E(i_0,i_n,i')$, $w(i_n,i_{n+1}) \in \beta^*$.

Muller automaton construction

We have used both the concepts of "favor" and "equivalence" in lemma 2. Now, we are going to use the result of the lemma to construct a Muller automaton for language $\alpha\beta^{\omega}$. The proposed automaton will accept a word iff a time i exists such that it will satisfy the right hand side of lemma 2. The machine below is described informally but note that this machine will be a deterministic Muller automaton.

The machine contains p+2 deterministic finite automaton and a master controller, where p is the size of $A^{\beta*}$. One of the p+2 machine can recognize $\alpha\beta*$ and this machine gets input in every cycle. And, it communicates at any time i to master controller whether or not $w(i) \in \alpha\beta*$. Rest of p+1 machines are copies of $A^{\beta*}$. These machine can be made dormant and activated. If master sets them to be dormant then they remain in initial state and become oblivious to the input. If master activates them then they keep reading the input and move until master makes them dormant and force them back to the initial state. Master can make them active and dormant as many times as it likes. At any time, master keeps following facts about $A^{\beta*}$ machines.

- Current states of $A^{\beta*}$ machines.
- A list of active $A^{\beta*}$ machines in the order of their activation time.
- For each active $A^{\beta*}$ machine M, the set of other active $A^{\beta*}$ machines that were in a accepting state at the time of activation of M. In other words, if a machine is made active at time i and some other machine was last made active at $j < i$ and continue to be active till i then the master keeps the record whether or not i favors j. This record is dropped if the other machine goes dormant before M.

The master may behave 2 different ways depending on α. If α contains empty word then only one of the $A^{\beta*}$ is active otherwise none of the $A^{\beta*}$ machines are active at the start. It at some time i, $w(i) \in \alpha\beta*$ and none of $A^{\beta*}$ machines are in initial state then master actives one of the dormant machines and the just activated $A^{\beta*}$ machine start receiving input from time i+1. If at some time instant two $A^{\beta*}$ machines reaches to same state then master makes the machine which was activated later dormant.

As output, the master also have a pair of red and green lights corresponding to each $A^{\beta*}$ machine. If a $A^{\beta*}$ machine goes active state to dormant state then corresponding red light flashes. The green light for some $A^{\beta*}$ machine M, which was activated at j, flashes at time i in following two situations:

- M is in initial state, thus E(j,i,i) and i favors j (the initial state has to be accepting state).
- For some other active $A^{\beta*}$ machine M' activated at k, where $j< k <i$, k favors j (the master keeps the record of this) and i is the earliest time at which E(j,k,i) (M' goes dormant at time i).

Lemma 3

If there exist a time equivalent to infinitely many times that favor it and i is the earliest such time, then a $A^{\beta*}$ machine M is activated at i, remained active forever (no corresponding red light flash thereafter), and flashes the green light infinitely many times.

Proof: Let's suppose a $A^{\beta*}$ machine was activated at time j such that j < i and this machine goes to initial state at time i. Therefore, if any time is equivalent and favors i then it must be in same relation with j. This contradicts the hypothesis that i is the earliest time such that infinitely many times equivalent to i and favoring i. So at time i, no active machine can be in the initial state. Hence, the master has to activate a new $A^{\beta*}$ machine at time i, which is our M. This machine will never go dormant because if some other machine, which was activated at time l, makes it dormant at time k then E(l,i,k). Again, the same contradiction is implied. By construction and due to infinitely many times are equivalent to i and favor i, the green light will flash infinitely often.

Lemma 4

Conversely, if there is a $A^{\beta*}$ machine M whose green light flashed infinitely often and red light only finitely often then there are infinitely many times equivalent to and favoring the last time at which M became active.

Proof: True by construction.

Lemma 5

"$w \in \alpha\beta^{\omega}$ iff, for some $A^{\beta*}$ machine, the green light flashes infinitely often and the red light flashes only finitely often."

Proof: Due to lemma 2-4.

The above description of a full machine can be viewed as a large deterministic automaton. Now, it is left to define the Muller acceptance condition. In this large automaton, we define μ_n to be the set of states in which the green light flashes and the red light does not flash corresponding to n^{th} $A^{\beta*}$ machine. Let ν_n be the set of states in which the red light does not flash corresponding to n^{th} $A^{\beta*}$ machine. So, Muller acceptance condition $\mathbf{F} = \{ S \mid \exists n\ \mu_n \subseteq S \subseteq \nu_n \}$. This finishes the construction of the desired Muller automaton. Q.E.D.

Other proofs

Since McNaughton's proof, many other proofs have been proposed. The following are some of them.

- ω-regular language can be shown equiv-expressive to Büchi automata. Büchi automata can be shown to equiv-expressive to semi-deterministic Büchi automata. Semi-deterministic Büchi automata can be shown to be equiv-expressive to deterministic Muller automata. This proof follows the same lines of the above proof.
- Safra's construction transforms a non-deterministic Büchi automaton to a Muller automaton. This construction is known to be optimal.
- There is a purely algebraic proof of McNaughton's Theorem.