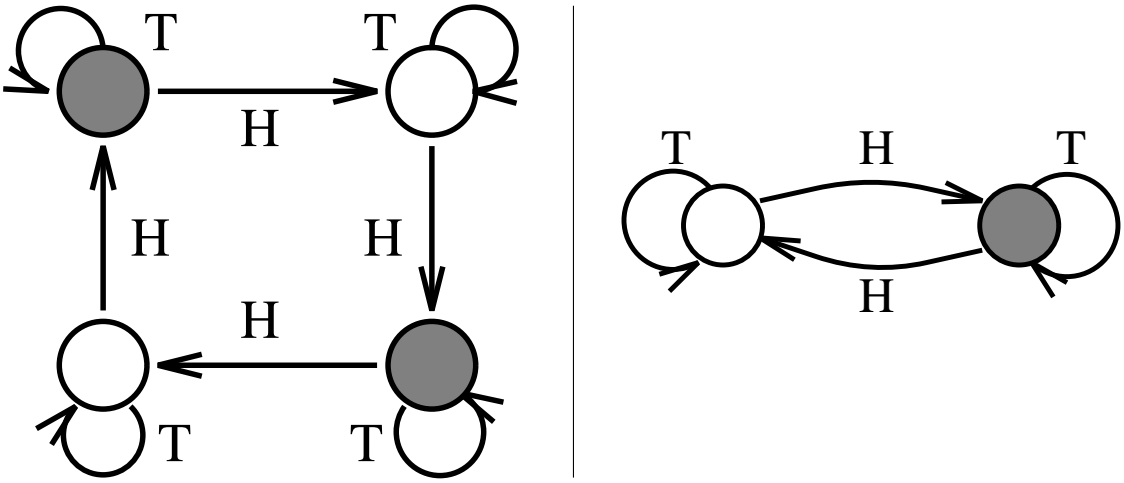


INPUT OUTPUT

## 18.7 Finite State Machine Minimization

**Input description**: A deterministic finite automaton $M$.

**Problem description**: Create the smallest deterministic finite automaton $M'$ such that $M'$ behaves identically to $M$.

**Discussion**: Constructing and minimizing finite state machines arises repeatedly in software and hardware design applications. Finite state machines are very useful for specifying and recognizing patterns. Modern programming languages such as Java and Python provide built-in support for *regular expressions*, a particularly natural way of defining automata. Control systems and compilers often use finite state machines to encode the current state and possible associated actions/transitions. Minimizing the size of these automata reduces both the storage and execution costs of dealing with such machines.

Finite state machines are defined by directed graphs. Each vertex represents a state, and each character-labeled edge defines a transition from one state to another on receipt of the given alphabet symbol. The automata shown analyzes a sequence of coin tosses, with dark states signifying that an even number of heads have been observed. Such automata can be represented using any graph data structure (see Section 12.4 (page 381)), or by an $n \times |\Sigma|$ *transition matrix* where $|\Sigma|$ is the size of the alphabet.

Finite state machines are often used to specify search patterns in the guise of regular expressions, which are patterns formed by and-ing, or-ing, and looping

over smaller regular expressions. For example, the regular expression $a(a+b+c)^*a$ matches any string on $(a, b, c)$ that begins and ends with distinct $a$s. The best way to test whether a string $s$ is recognized by a given regular expression $R$ constructs the finite automaton equivalent to $R$, and then simulates this machine on $S$. See Section 18.3 (page 628) for alternative approaches to string matching.

We consider three different problems on finite automata:

- *Minimizing deterministic finite state machines* – Transition matrices for finite automata become prohibitively large for sophisticated machines, thus fueling the need for tighter encodings. The most direct approach is to eliminate redundant states in the automaton. As the example above illustrates, automata of widely varying sizes can compute the same function.

  Algorithms for minimizing the number of states in a deterministic finite automaton (DFA) appear in any book on automata theory. The basic approach partitions the states into gross equivalence classes and then refines the partition. Initially, the states are partitioned into accepting, rejecting, and other classes. The transitions from each node now branch to a given class on a given symbol. Whenever two states $s$, $t$ from the same class $C$ branch to elements of different classes, the class $C$ must be partitioned into two subclasses, one containing $s$, the other containing $t$.

  This algorithm makes a sweep though all the classes looking for a new partition, and repeats the process from scratch if it finds one. This yields an $O(n^2)$ algorithm, since at most $n-1$ sweeps need ever be performed. The final equivalence classes correspond to the states in the minimum automaton. In fact, a more efficient $O(n \log n)$ algorithm is known. Implementations are cited below.

- *Constructing deterministic machines from nondeterministic machines* – DFAs are simple to work with, because the machine is always in exactly one state at any given time. *Nondeterministic automata* (NFAs) can be in multiple states at a time, so their current "state" represents a subset of all possible machine states.

  In fact, any NFA can be mechanically converted to an equivalent DFA, which can then be minimized as above. However, converting an NFA to a DFA might cause an exponential blowup in the number of states, which perversely might then be eliminated when minimizing the DFA. This exponential blowup makes most NFA minimization problems PSPACE-hard, which is even worse than NP-complete.

  The proofs of equivalence between NFAs, DFAs, and regular expressions are elementary enough to be covered in undergraduate automata theory classes. However, they are surprisingly nasty to actually code. Implementations are discussed below.

- *Constructing machines from regular expressions* – There are two approaches for translating a regular expression to an equivalent finite automaton. The difference is whether the output automaton will be a nondeterministic or deterministic machine. NFAs are easier to construct but less efficient to simulate.

  The nondeterministic construction uses $\epsilon$-moves, which are optional transitions that require no input to fire. On reaching a state with an $\epsilon$-move, we must assume that the machine can be in either state. Using $\epsilon$-moves, it is straightforward to construct an automaton from a depth-first traversal of the parse tree of the regular expression. This machine will have $O(m)$ states, if $m$ is the length of the regular expression. Furthermore, simulating this machine on a string of length $n$ takes $O(mn)$ time, since we need consider each state/prefix pair only once.

  The deterministic construction starts with the parse tree for the regular expression, observing that each leaf represents an alphabet symbol in the pattern. After recognizing a prefix of the text, we can be left in some subset of these possible positions, which would correspond to a state in the finite automaton. The *derivatives* method builds up this automaton state by state as it is needed. Even so, some regular expressions of length $m$ require $O(2^m)$ states in any DFA implementing them, such as $(a+b)^*a(a+b)(a+b)\ldots(a+b)$. There is no way to avoid this exponential space blowup. Fortunately it takes linear time to simulate an input string on any DFA, regardless of the size of the automaton.

**Implementations**: *Grail+* is a C++ package for symbolic computation with finite automata and regular expressions. Grail enables one to convert between different machine representations and to minimize automata. It can handle large machines defined on large alphabets. All code and documentation are accessible from *http://www.csd.uwo.ca/Research/grail*, as well as pointers to a variety of other automaton packages. Commercial use of Grail is not allowed without approval, although it is freely available to students and educators.

The AT&T Finite State Machine Library (FSM) is a set of general-purpose UNIX software tools for building, combining, optimizing, and searching weighted finite-state acceptors and transducers. It supports automata with more than ten million states and transitions. See *http://www.research.att.com/~fsmtools/fsm/*.

*JFLAP* (Java Formal Languages and Automata Package) is a package of graphical tools for learning the basic concepts of automata theory. Included are functions to convert between DFAs, NFAs, and regular expressions, and minimize the resulting automata. High-level automata are also supported, including context-free languages and Turing machines. *JFLAP* is available at *http://www.jflap.org/*. A related book [RF06] is also available.

*FIRE Engine* provides production-quality implementations of finite automata and regular expression algorithms. Several finite automaton

minimization algorithms have been implemented, including Hopcroft's $O(n \lg n)$ algorithm. Both deterministic and nondeterministic automata are supported. It is available at *http://www.fastar.org/* and, with certain enhancements, at *www.eti.pg.gda.pl/∼jandac/minim.html*.

**Notes**: Aho [Aho90] provides a good survey on algorithms for pattern matching, with a particularly clear exposition for the case where the patterns are regular expressions. The technique for regular expression pattern matching with $\epsilon$-moves is due to Thompson [Tho68]. Other expositions on finite automaton pattern matching include [AHU74]. Expositions on finite automata and the theory of computation include [HMU06, Sip05]

The major annual meeting of interest in this field is the *Conference on Implementations and Applications of Automata* (CIAA). Pointers to current and previous meetings with associated software are available at *http://tln.li.univ-tours.fr/ciaa/*.

Hopcroft [Hop71] gave an optimal $O(n \lg n)$ algorithm for minimizing the number of states in DFAs. The derivatives method of constructing a finite state machine from a regular expression is due to Brzozowski [Brz64] and has been expanded upon in [BS86]. Expositions on the derivatives method includes Conway [Con71]. Recent work on incremental construction and optimization of automata includes [Wat03]. The problems of compressing a DFA to a minimum NFA [JR93] and testing the equivalence of two nondeterministic finite state machines [SM73] are both PSPACE-complete.

**Related Problems**: Satisfiability (see page 472). string matching (see page 628).