Chapter 7

Alternating Choices

March 21, 2002

7.1 Computing with Choice

The choice-mode of computation comes in two main flavors. The first is based on probability and briefly discussed in Chapter 1 (Section 6.2). The second is a generalization of nondeterministism called *alternation*. Let us briefly see what an alternating computation looks like. Let δ be the usual Turing transition table that has choice and let $C_0(w)$ denote the initial configuration of δ on input w. For this illustration, assume that every computation path is finite; in particular, this implies that no configuration is repeated in a computation path. The computation tree $T(w) = T_\delta(w)$ is defined in the obvious way: the nodes of $T(w)$ are configurations, with $C_0(w)$ as the root; if configuration C is a node of $T(w)$ and $C \vdash C'$ then C' is a child of C in $T(w)$. Thus the leaves of $T(w)$ are terminal configurations. The description of an alternating machine M amounts to specifying a transition table δ together with an assignment γ of a Boolean function $\gamma(q) \in \{\wedge, \vee\neg\}$ to each state q in δ. This induces a Boolean value on each node of $T(w)$ as follows: the leaves are assigned 1 or 0 depending on whether the configuration is accepting or not. If C is not a leaf, and q is the state in C, then we require that the number of children of C is equal to the arity of $\gamma(q)$. For instance, if C has two children whose assigned values are x and y then C is assigend the value $\gamma(q)(x, y)$. Finally we say M accepts w if the root $C_0(w)$ is assigned value 1.

The reader will see that nondeterministic computation corresponds to the case where $\gamma(q) = \vee$ for all q. Since the introduction of alternating machines by Chandra, Kozen and Stockmeyer[3] in 1978, the concept has proven to be an extremely useful tool in Complexity Theory.

The model of probabilistic machines we study was introduced by Gill[7]. Let us rephrase the description of probabilistic computation in Chapter 1 in terms of assigning values to nodes of a computation tree. A probabilistic machine is formally a transition table δ where each configuration spawns either zero or two children. For any input w, we again have the usual computation tree $T(w)$. The leaves of $T(w)$ are given a value of 0 or 1 as in the alternating case. However, an internal node u of $T(w)$ is assigned the average $(x + y)/2$ of the values x, y of the two children of u. The input w is accepted if the root is assigned a value greater than 1/2. (The reader should be verify that this description is equivalent to the one given in Chapter 1.) The function $f(x, y) = (x + y)/2$ is called the *toss* function because in probabilistic computations, making choices is interpreted as branching according to the outcomes of tossing a fair coin.

Hence, a common feature of probabilistic and alternating modes is their systematic bottom-up method of assigning values to nodes of computation trees. One difference is that, whereas probabilistic nodes are given (rational) values between 0 and 1, the alternating nodes are assigned Boolean values. We modify this view of alternating machines by regarding the Boolean values as the real numbers 0 and 1, and generalizing the Boolean functions $\wedge$, $\vee$ and $\neg$ to the real functions min, max and $f(x) = 1 - x$ (respectively).

With this shift of perspective, we have almost accomplished the transition to a new syncretistic model that we call *probabilistic-alternating machines*. This model was first studied in [23]. A probabilistic-alternating machine M is specified by giving a transition table δ and each state is associated with one of the four real functions

$$\min(x, y), \qquad \max(x, y), \qquad 1 - x, \qquad \frac{x + y}{2}. \tag{1}$$

We require c onfiguration in state q to spawn m children where m is the arity of the function associated with q. This can be enforced by synthetic restrictions on the transition table δ. Given an input w, we construct the tree

$T(w)$ and assign values to its nodes in the usual bottom-up fashion (again, assuming $T(w)$ is a finite tree). We say M accepts the input w if the value at the root of $T(w)$ is $> 1/2$.

Probabilistic and alternating machines in the literature are usually studied independently. In combining these two modes, we extend results known for only one of the modes, or unify distinct results for the separate modes. More importantly, it paves the way towards a general class of machines that we call *choice machines*. Computations by choice machines are characterized by the systematic assignment of 'values' to nodes of computation trees, relative to the functions $\gamma(q)$ which the machine associates to each state q. These functions are similar to those in (1), although an immediate question is what properties should these functions satisfy? This will be answered when the theory is developed. We call any assignment of "values" to the nodes of a computation tree a *valuation*.[1] Intuitively, these values represent probabilities and lies in the unit interval $[0, 1]$. But because of infinite computation trees, we are forced to take as "values" any subinterval $[a, b]$ of the unit interval $[0, 1]$. Such intervals represent uncertainty ranges in the probabilities. This leads to the use of a simple interval algebra. The present chapter develops the valuation mechanism needed for our theory of choice machines. We will specifically focus on alternation machines, leaving stochastic machines to the next chapter.

Other choice modes. Other authors independently proposed a variety of computational modes that turn out to be special cases of our probabilistic-alternating mode: **interactive proof systems** (Goldwasser, Micali and Rackoff [8]), **Arthur-Merlin games** (Babai [2]), **stochastic Turing machines** (Papadimitriou [16]), **probabilistic-nondeterministic machines** (Goldwasser and Sipser [9]). In general, communication protocols and game playing models can be translated as choice machines. In particular, this holds for the **probabilistic game automata** (Condon and Ladner [4]) which generalize interactive proof systems and stochastic machines[2]. Alternating machines are generalized to **logical type machines** (Hong [11]) where machine states can now be associated with any of the 16 Boolean functions on two variables. Some modes bearing little resemblance to choice machines can nevertheless be viewed as choice machines: for example, in Chapter 9 we describe a choice mode that generalizes nondeterminism in a different direction than alternation. (This gives rise to the so-called *Boolean Hierarchy*.) These examples suggests that the theory of valuation gives a proper foundation for choice modes of computation. The literature can avoid our systematic development only by restrictions such as requiring constructible time-bounds.

7.2 Interval Algebra

The above introduction to probabilistic-alternating machines explicitly avoided infinite computation trees $T(x)$. Infinite trees cannot be avoided in general; such is the case with space-bounded computations or with probabilistic choices. In particular, a computation using finite amount of space may have infinite computation paths. To see why infinite trees are problematic, recall that we want to systematically assign a value in $[0, 1]$ to each node of $T(x)$, in a bottom-up fashion. But if a node u of $T(x)$ lies on an infinite path, it is not obvious what value to assign to u.

Our solution [23] lies in assigning to u the smallest 'confidence' interval $I(u) \subseteq [0, 1]$ guaranteed to contain the 'true' value of u. This leads us to the following development of an *interval algebra*[3].

In the following, u, v, x, y, etc., denote real numbers in the unit interval [0, 1]. Let

$$INT := \{[u, v] : 0 \le u \le v \le 1\}$$

denote the set of closed subintervals of $[0, 1]$. An interval $[u, v]$ is *exact* if $u = v$, and we identify the exact interval $[u, u]$ with the real number u. We call u and v (respectively) the *upper* and *lower bounds* of the interval $[u, v]$. The unit interval $[0, 1]$ is also called *bottom* and denoted $\perp$.

By an *interval function* we mean a function $f : INT^n \to INT$, where $n \ge 0$ denotes the arity of the function. We are interested in six interval functions. The first is the unary function of *negation* ($\neg$), defined as follows:

$$\neg[x, y] = [1 - y, 1 - x].$$

The remaining five are binary functions:

[1] The term 'valuation' in algebra refers to a real function on a ring that satisfies certain axioms. Despite some concern, we will expropriate this terminology, seeing little danger of a context in which both senses of the term might be gainfully employed.

[2] This game model incorporates 'partially-hidden information'. It will be clear that we could add partially-hidden information to choice machines too.

[3] *Interval arithmetic*, a subject in numerical analysis, is related to our algebra but serves a rather different purpose. We refer the reader to, for example, Moore [15].

minimum ($\wedge$), *maximum* ($\vee$),
toss ($\text{\textcircled{T}}$),
probabilistic-and ($\otimes$), *probabilistic-or* ($\oplus$).

It is convenient to first define them as real functions. The real functions of minimum and maximum are obvious. The toss function is defined by

$$x \,\text{\textcircled{T}}\, y := \frac{x+y}{2}.$$

We saw in our introduction how this function arises from probabilistic (coin-tossing) algorithms. The last two functions are defined as follows:

$$\begin{array}{rcl} x \otimes y & := & xy \\ x \oplus y & := & x + y - xy \end{array}$$

Thus $\otimes$ is ordinary multiplication of numbers but we give it a new name to signify the interpretation of the numbers as probabilities. If E is the event that *both* E_1 *and* E_2 occur, then the probability $\Pr(E)$ of E occurring is given by

$$\Pr(E) = \Pr(E_1) \otimes \Pr(E_2).$$

We assume that E_1, E_2 are independent events. Similarly $\oplus$ has a probabilistic interpretation: if E is the event that *either* E_1 *or* E_2 occurs, then

$$\Pr(E) = \Pr(E_1) \oplus \Pr(E_2).$$

To see this, simply note that $x \oplus y$ can also be expressed as $1 - (1-x)(1-y)$. For brevity, we suggest reading $\otimes$ and $\oplus$ as 'prand' and 'pror', respectively.

We note that these 5 real functions can also be regarded as functions on $[0,1]$ (*i.e.*, if their arguments are in $[0,1]$ then their values remain in $[0,1]$). We may then extend them to the subintervals *INT* of the unit interval as follows. If $\circ$ is any of these 5 functions, then we define

$$[x,y] \circ [u,v] := [(x \circ u), (y \circ v)].$$

For instance, $[x,y] \otimes [u,v] = [xu, yv]$ and $[x,y] \wedge [u,v] = [\min(x,u), \min(y,v)]$. Alternatively, for any continuous function $f : [0,1] \to [0,1]$, we extend the range and domain of f from $[0,1]$ to *INT* by the definition $f(I) = \{f(x) : x \in I\}$. If f is also monotonic, this is equivalent to the above.

One easily verifies:

LEMMA 1 *All five binary functions are commutative. With the exception of* $\text{\textcircled{T}}$*, they are also associative.*

The set *INT* forms a lattice with $\wedge$ and $\vee$ as the join and meet functions, respectively[4]. It is well-known that we can define a partial order $\le$ in any lattice by:

$$[x,y] \le [u,v] \iff ([x,y] \wedge [u,v]) = [x,y]. \qquad (2)$$

Note that (2) is equivalent to:

$$[x,y] \le [u,v] \iff x \le u \text{ and } y \le v.$$

When we restrict this partial ordering to exact intervals, we get the usual ordering of real numbers. For reference, we will call $\le$ the *lattice-theoretic ordering* on *INT*.

The negation function is not a complementation function (in the sense of Boolean algebra [5]) since neither $I \wedge \neg I = 0$ nor[5] $I \vee \neg I = 1$ holds for all $I \in \mathit{INT}$. However it is idempotent, $\neg\neg I = I$. Probabilistic-and and probabilistic-or can be recovered from each other in the presence of negation. For example,

$$I \otimes J = \neg(\neg I \oplus \neg J).$$

It easy to verify the following forms of de Morgan's law:

[4]A lattice X has two binary functions, join and meet, satisfying certain axioms (essentially all the properties we expect from max and min). Lattice-theoretic notations can be found, for instance, in [5]. The lattice-theoretic properties are not essential for the development of our results.

[5]We assume that $\neg$ has higher precedence than the binary operators so we may omit parenthesis when convenient.

Lemma 2

$$\begin{aligned}
\neg(I \wedge J) &= \neg I \vee \neg J \\
\neg(I \vee J) &= \neg I \wedge \neg J \\
\neg(I \mathbin{\bigcirc\!\!\!\!\!\top} J) &= \neg I \mathbin{\bigcirc\!\!\!\!\!\top} \neg J \\
\neg(I \otimes J) &= \neg I \oplus \neg J \\
\neg(I \oplus J) &= \neg I \otimes \neg J
\end{aligned}$$

where $I, J \in INT$.

In view of these laws, we say that the functions $\wedge$ and $\vee$ are *duals of each other* (with respect to negation); similarly for the pair $\otimes$ and $\oplus$. However, $\mathbin{\bigcirc\!\!\!\!\!\top}$ is self-dual.

We verify the distributivity of $\wedge$ and $\vee$ with respect to each other:

$$I \vee (J_1 \wedge J_2) = (I \vee J_1) \wedge (I \vee J_2)$$
$$I \wedge (J_1 \vee J_2) = (I \wedge J_1) \vee (I \wedge J_2).$$

Furthermore, $\mathbin{\bigcirc\!\!\!\!\!\top}$, $\otimes$ and $\oplus$ each distributes over both $\wedge$ and $\vee$:

$$\begin{aligned}
I \mathbin{\bigcirc\!\!\!\!\!\top} (J_1 \wedge J_2) = (I \mathbin{\bigcirc\!\!\!\!\!\top} J_1) \wedge (I \mathbin{\bigcirc\!\!\!\!\!\top} J_2), &\quad I \mathbin{\bigcirc\!\!\!\!\!\top} (J_1 \vee J_2) = (I \mathbin{\bigcirc\!\!\!\!\!\top} J_1) \vee (I \mathbin{\bigcirc\!\!\!\!\!\top} J_2) \\
I \otimes (J_1 \wedge J_2) = (I \otimes J_1) \wedge (I \otimes J_2), &\quad I \otimes (J_1 \vee J_2) = (I \otimes J_1) \vee (I \otimes J_2) \\
I \oplus (J_1 \wedge J_2) = (I \oplus J_1) \wedge (I \oplus J_2), &\quad I \oplus (J_1 \vee J_2) = (I \oplus J_1) \vee (I \oplus J_2)
\end{aligned}$$

However $\otimes$ and $\oplus$ do not distribute with respect to each other (we only have $x \otimes (y \oplus z) \le (x \otimes y) \oplus (x \otimes z)$). And neither $\wedge$ nor $\vee$ distributes over $\mathbin{\bigcirc\!\!\!\!\!\top}$, $\otimes$ or $\oplus$.

Another Partial Order. For our applications, it turns out that a more useful partial order on INT is $\sqsubseteq$, defined by:

$$[x, y] \sqsubseteq [u, v] \iff x \le u \text{ and } v \le y.$$

Clearly $\sqsubseteq$ is the reverse of the set inclusion relation between intervals: $I \sqsubseteq J \iff J \supseteq I$ as sets. With respect to the $\sqsubseteq$-ordering, all exact intervals are maximal and pairwise incomparable[6]. In view of our interpretation of intervals as 'intervals of confidence', if $I \sqsubseteq J$ then we say J has 'at least as much information' as I. For this reason, we call $\sqsubseteq$ the *information-ordering*. In contrast to the lattice-theoretic $\le$-ordering, $\sqsubseteq$ only gives rise to a lower semi-lattice with the meet function $\sqcap$ defined by

$$[x, y] \sqcap [u, v] = [\min(x, u), \max(y, v)].$$

(The following suggestion for defining the join $\sqcup$ fails: $[x, y] \sqcup [u, v] = [\max(x, u), \min(y, v)]$.) Note that bottom $\perp$ is the least element ("no information") in the information-ordering.

Example 1 The strong 3-valued algebra described[7] by Chandra, Kozen and Stockmeyer [3] is a subalgebra of our interval algebra, obtained by restricting values to $\{0, 1, \perp\}$. See figure 7.1 for its operation tables. They only

$\wedge$	0	1	$\perp$
0	0	0	0
1	0	1	$\perp$
$\perp$	0	$\perp$	$\perp$

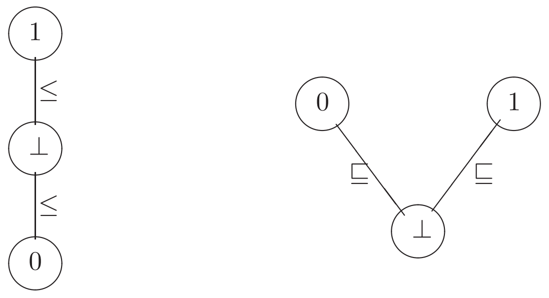
$\vee$	0	1	$\perp$
0	0	1	$\perp$
1	1	1	1
$\perp$	$\perp$	1	$\perp$

Figure 7.1: The strong 3-valued algebra.

[6] I and J are $\sqsubseteq$-*comparable* if $I \sqsubseteq J$ or $J \sqsubseteq I$, otherwise they are $\sqsubseteq$-*incomparable*.
[7] Attributed to Kleene.

were interested in the functions $\wedge$, $\vee$, $\neg$. Thus our interval algebra gives a model (interpretation) for this 3-valued algebra.

The contrast between $\leq$ and $\sqsubseteq$ is best exemplified by the respective partial orders restricted to this 3-valued algebra, put graphically:



■

This information ordering gives rise to some important properties of interval functions. Given a sequence of $\sqsubseteq$-increasing intervals

$$I_1 \sqsubseteq I_2 \sqsubseteq \cdots,$$

we define its **limit** in the natural way: $\lim_{j\geq 1} I_j$ is just $\cap_{j\geq 1} = I_j$.

Definition 1
(i) An n-ary function

$$f : \mathit{INT}^n \to \mathit{INT}$$

is *monotonic* if for all intervals $J_1, \ldots, J_n, J'_1, \ldots, J'_n$:

$$J_1 \sqsubseteq J'_1, \ldots, J_n \sqsubseteq J'_n \;\Rightarrow\; f(J_1, \ldots, J_n) \sqsubseteq f(J'_1, \ldots, J'_n).$$

(ii) f is *continuous* if it is monotonic and for all non-decreasing sequences

$$J_i^{(1)} \sqsubseteq J_i^{(2)} \sqsubseteq J_i^{(3)} \sqsubseteq \cdots$$

$(i = 1, \ldots, n)$, we have that

$$f(\lim_j \{J_1^{(j)}\}, \ldots, \lim_j \{J_n^{(j)}\}) = \lim_j f(J_1^{(j)}, \ldots, J_n^{(j)}). \tag{3}$$

■

Note that a continuous function is assumed monotonic. This ensures that the limit on the right-hand side of (3) is meaningful because monotonicity of f implies

$$f(J_1^{(1)}, \ldots, J_n^{(1)}) \sqsubseteq f(J_1^{(2)}, \ldots, J_n^{(2)}) \sqsubseteq f(J_1^{(3)}, \ldots, J_n^{(3)}) \sqsubseteq \cdots.$$

Lemma 3 *The six functions $\wedge$, $\vee$, $\mathbin{\bigcirc\kern-0.75em\top}$, $\otimes$, $\oplus$ and $\neg$ are continuous.*

We leave the proof as an exercise. Continuity of these functions comes from continuity of their real counterpart.

Example 2 The *cut-off function* $\delta_{\frac{1}{2}}(x)$ is defined to be 1 if $x > \frac{1}{2}$ and 0 otherwise. We extend this function to intervals in the natural way: $\delta_{\frac{1}{2}}([u, v]) = [\delta_{\frac{1}{2}}(u), \delta_{\frac{1}{2}}(v)]$. This function is monotonic but not continuous. To see this, let $I_i = [0, \frac{1}{2} + \frac{1}{i}]$. Then $\lim_i I_i = [0, \frac{1}{2}]$ but

$$0 = \delta_{\frac{1}{2}}([0, \frac{1}{2}]) \neq \lim_i \delta_{\frac{1}{2}}(I_i) = \bot.$$

■

The following simple observation is useful for obtaining monotonic and continuous functions.

Lemma 4
(i) Any composition of monotonic functions is monotonic.
(ii) Any composition of continuous functions is continuous.

7.3 Theory of Valuations

To give a precise definition for the choice mode of computation, and its associated complexity measures, we introduced the theory of valuations. We also signal a new emphasis through a terminology change:

- **Global Decisions.** In Chapter 0, a non-accepting computation is classified as either rejecting or looping. Now, the disposition of an acceptor with respect to any input is one of the following: *acceptance*, *rejection* or *indecision*. These three dispositions are called "global decisions" because they are based on the entire computation tree. We will define them using valuations. Intuitively, looping has been replaced by indecision but the two concepts are not identical.

- **Local Answers.** Each computation path leads to a "local answer". If the path is finite, the local answer resides in the terminal configuration C of the path. Previously, the local answers are either accept or not-accept. We now have three local answers from C: **YES**, **NO** or **YO**. If the path is infinite, the local answer is **YO**. These answers are analogues of the three global decisions. In particular "YO" is neither Yes or No, i.e., indecision.

To implement the local answers, we introduce two distinguished states,

$$q_Y, q_N \in Q_\infty.$$

called the *YES-state* and *NO-state*, respectively, These are replacements for our previous accept and reject states (q_a and q_r). We can further arrange transition tables so configurations with these states are terminal. A terminal configuration C is called a *YES-configuration*, *NO-configuration* or a *YO-configuration*, depending on whether its state is q_Y, q_N or some other state. A computation path is called a *YES-path*, *NO-path* or *YO-path*, depending on whether it terminates in a YES, NO or otherwise. Thus a YO-path either terminates in a YO-configuration or is non-terminating.

Naturally, the global decision is some generalized average of the local answers. This global/local terminology anticipates the quantitative study of errors in a computation (see chapter 8). For now, it suffices to say that all error concepts are based on the discrepancies between global decisions and local answers. The seemingly innocuous introduction of YO-answers[8] is actually critical in our treatment of errors.

Let f be a function on *INT*, i.e., there is an $n \geq 0$ (n is the arity of f) such that

$$f : INT^n \to INT.$$

The 0-ary functions are necessarily *constant functions*, and the *identity function* $\iota(I) = I$ has arity 1.

Definition 2 A set B of functions on *INT* is called a *basis set* if the functions in B are continuous and B contains the identity function ι.

For any basis set B, a *B-acceptor* is a triple $M = (\delta, \gamma, \overset{M}{<})$ where δ is a Turing transition table whose state set Q_M is ordered by a total ordering $\overset{M}{<}$, and γ associates a basis function $\gamma(q)$ to each state $q \in Q_M$,

$$\gamma : Q \to B.$$

Moreover, δ has the property that if C is a configuration of δ in state q and $\gamma(q)$ has arity n, then C either is a terminal configuration or has exactly n immediate successors $C_1, \ldots, C_n$ such that the C_i's have distinct states. ■

We will see that, without loss of generality, the 0-ary functions $0, 1$ and $\bot$ can also be assumed to be in B. We simply call M a *choice acceptor (or machine)* if B is understood.

Explanation. We describe how choice machines operate. If the immediate successors of a configuration C are $C_1, \ldots, C_n$ such that the state of C_i is less than the state of C_{i+1} (under the ordering $\overset{M}{<}$) for each $i = 1, \ldots, n-1$, then we indicate this by writing[9]

$$C \vdash (C_1, \ldots, C_n).$$

If q is the state of C, we also write $\gamma(C)$ or γ_C instead of $\gamma(q)$. We require that $C_1, \ldots, C_n$ be distinct states because the value of the node (labeled by) C in the computation tree is given by $\gamma_C(v_1, \ldots, v_n)$ where v_i is the

[8]We owe the YO-terminology to the unknown street comedian in Washington Square Park whose response to an ongoing public campaign called "Just say NO to drugs" was: "we say YO to drugs". Needless to say, this local answer is in grave error.

[9]This notation could cause some confusion because we do not want to abandon the original meaning of "$C \vdash C'$", that C' is a successor of C. Hence "$C \vdash C'$" does not mean that C has only one successor; to indicate this, we need to write "$C \vdash (C')$".

Name	Basis B	Mode Symbol
deterministic	$\emptyset$	D
nondeterministic	$\{\vee\}$	N
probabilistic	$\{\bigcirc\!\!\!\!\!\!\top\}$	Pr
alternating	$\{\wedge, \vee, \neg\}$	A
interactive proofs	$\{\bigcirc\!\!\!\!\!\!\top, \vee\}$	Ip
probabilistic-alternating	$\{\bigcirc\!\!\!\!\!\!\top, \wedge, \vee, \neg\}$	PrA
stochastic	$\{\bigcirc\!\!\!\!\!\!\top, \otimes, \oplus, \neg\}$	St
stochastic-alternating	$\{\bigcirc\!\!\!\!\!\!\top, \otimes, \oplus, \wedge, \vee, \neg\}$	StA

Figure 7.2: Some Choice Modes and their Symbols.

value of the node (labeled by) C_i. Without an ordering such as $\overset{M}{<}$ on the children of C, we have no definite way to assign the v_i's as arguments to the function γ_C. But for basis sets that we study, the functions are symmetric in their arguments and so we will not bother to mention the ordering $\overset{M}{<}$. ■

The table in Figure 7.3 collects some common classes of B-choice machines. Since every basis set contains the identity function ι, it is omitted when writing out B.

Each basis set B gives rise to a new computational mode. The symbols for these modes are in the third column of this table. We shall say a B-machine makes *B-choices*. Thus, nondeterministic machines makes nondeterministic choices and alternating machines makes alternating choices. MIN- and MAX-choices are also called *universal choices* and *existential choices*; Coin-tossing choices are also called random choices or probabilistic choices.

From Figure 7.3, it is evident that we differentiate between the words 'probabilistic' and 'stochastic': the adjective 'probabilistic' applies only to coin-tossing concepts – a usage that conforms to the literature. The adjective 'stochastic' is more general and includes coin-tossing concepts. We abbreviate a probabilistic-alternating machine to 'PAM', and a stochastic-alternating machine to 'SAM'.

If $\gamma(q) = \wedge$ (respectively, $\vee, \bigcirc\!\!\!\!\!\!\top, \otimes, \oplus, \neg$) then q is called an *MIN-state* (respectively, *MAX-, TOSS-, PrAND-, PrOR-, NOT-state*). If the state of C is an MIN-state (MAX-state, etc.), then C is an *MIN-configuration* (*MAX-configuration*, etc.).

Example 3 This is an instructive exercise if you want to have some understanding of alternating machines. Recall the palindrome language

$$L_{pal} = \{w : w \in \{0,1\}^*, w = w^R\}$$

. We will show that an alternating machine M can accept L_{pal} in linear time, using only logarithmic space.

To understand what this tells us about the power of alternation, recall from Chapter 2,

$$L_{pal} \in \textit{D-TIME-SPACE}(O(n), O(n))$$

and

$$L_{pal} \in \textit{D-TIME-SPACE}(O(n^2), \log n).$$

Furthermore, if

$$L_{pal} \in \textit{N-TIME-SPACE}(t(n), s(n))$$

then $s(n) \cdot t(n) = \Omega(n^2)$. Hence this example shows that the alternating mode is strictly more powerful than the fundamental mode. Of course, the formal definition of what it means for a choice machine to accept a word, and the notion of space and time complexity is yet to come. But if our machine halts on all paths, then these concepts are intuitively obvious: we rely on such intuitions for the reader to understand the example. The idea of the construction is that M accepts an input w provided for all $i = 1, \ldots, n$, $w[i] = w[n+1-i]$, provided $|w| = n$.

In phase 1, the machine M on input w of length n marks out some $m \geq 0$ cells using existential choice. It is not hard to show that the simple procedure P that repeatedly increments a binary counter from 0 to 2^m, taking $O(2^m)$ steps overall. In phase 2, M deterministically checks if $2^{m-1} < n \leq 2^m$ using this procedure P, answering NO otherwise. Hence phases 1 and 2 take linear time. In phase 3, M universally guesses a bit for each of the marked cells. This takes $O(\log n)$ steps. At the end of phase 3, M is armed with a binary number i between 0 and 2^{m+1}. Then it deterministically tests if $w[i] = w[n-i]$. If $i > n$ then this test, by definition, passes. In any case this test takes a linear number of steps, again using procedure P. This completes our description of M. It is clear that M accepts L_{pal} and uses $O(\log n)$ space. ■

We now want to define acceptance by choice machines. Basically we need to assign intervals $[u, v] \in \mathit{INT}$ to nodes in computation trees. The technical tool we employ is the concept of a 'valuation'.

Definition 3 Let M= (δ, γ) be a choice machine. The set of configurations of δ is denoted $\Delta(M)$. A *valuation* of M is a function

$$V \ : \ \Delta(M) \to \mathit{INT}.$$

A partial ordering on valuations is induced from the $\sqsubseteq$ -ordering on INT as follows: for valuations V_1 and V_2, define $V_1 \sqsubseteq V_2$ if

$$V_1(C) \sqsubseteq V_2(C)$$

for all $C \in \Delta(M)$. The *bottom valuation*, denoted $V_\perp$, is the valuation that always yield $\perp$. Clearly $V_\perp \sqsubseteq V$ for any valuation V. ■

Definition 4 Let $\Delta \subseteq \Delta(M)$. We define the following operator τ_Δ on valuations. If V is a valuation, then $\tau_\Delta(V)$ is the valuation V' defined by:

$$V'(C) = \begin{cases} \perp & \text{if} \quad C \notin \Delta \text{ or } C \text{ is YO-configuration,} \\ 1 & \text{else if} \quad C \text{ is a YES-configuration,} \\ 0 & \text{else if} \quad C \text{ is a NO-configuration,} \\ \gamma_C(V(C_1), \ldots, V(C_n)) & \text{else if} \quad C \vdash (C_1, \ldots, C_n). \end{cases}$$

■

For instance, we may choose Δ to be the set of all configurations of M that uses at most space h (for some h).

Lemma 5 (Monotonicity) *$\Delta_1 \subseteq \Delta_2$ and $V_1 \sqsubseteq V_2$ implies $\tau_{\Delta_1}(V_1) \sqsubseteq \tau_{\Delta_2}(V_2)$.*

Proof. We must show $\tau_{\Delta_1}(V_1)(C) \sqsubseteq \tau_{\Delta_2}(V_2)(C)$ for all $C \in \Delta(M)$. If $C \notin \Delta_1$, then this is true since the left-hand side is equal to $\perp$. So assume $C \in \Delta_1$. If C is terminal, then $\tau_{\Delta_1}(V_1)(C) = \tau_{\Delta_2}(V_2)(C)$ $(= 0, 1 \text{ or } \perp)$. Otherwise, $C \vdash (C_1, \ldots, C_n)$ where n is the arity of γ_C. Then

$$\begin{aligned} \tau_{\Delta_1}(V_1)(C) &= \gamma_C(V_1(C_1), \ldots, V_1(C_n)) \\ &\sqsubseteq \gamma_C(V_2(C_1), \ldots, V_2(C_n)) \\ &= \tau_{\Delta_2}(V_2)(C). \end{aligned}$$

where the $\sqsubseteq$ follows from the monotonicity of γ_C. **Q.E.D.**

For any $\Delta \subseteq \Delta(M)$ and $i \geq 0$, let τ^i_Δ denote operator obtained by the i-fold application of τ_Δ, i.e.,

$$\tau^0_\Delta(V) = V, \quad \tau^{i+1}_\Delta(V) = \tau_\Delta(\tau^i_\Delta(V)).$$

As corollary, we get

$$\tau^i_\Delta(V_\perp) \ \sqsubseteq \ \tau^{i+1}_\Delta(V_\perp)$$

for all $i \geq 0$. To see this, use induction on i and the monotonicity lemma.

Definition 5 From the compactness of the interval $[0, 1]$, we see that there exists a unique least upper bound Val_Δ defined by

$$\mathit{Val}_\Delta(C) = \lim\{\tau^i_\Delta(V_\perp)(C) \ : \ i \geq 0\},$$

for all $C \in \Delta$. If $\Delta = \Delta(M)$, then we denote the operator τ_Δ by τ_M, and the valuation Val_Δ by Val_M. ■

A simple consequence of the monotonicity lemma is the following:

$$\Delta_1 \subseteq \Delta_2 \quad \Rightarrow \quad \mathit{Val}_{\Delta_1} \ \sqsubseteq \ \mathit{Val}_{\Delta_2}.$$

To see this, it is enough to note that for all $i \geq 0$, $\tau^i_{\Delta_1}(V_\perp) \ \sqsubseteq \ \tau^i_{\Delta_2}(V_\perp)$.

For any operator τ and valuation V, we say V is a *fixed point* of τ if $\tau(V) = V$.

Lemma 6 *Val_Δ is the least fixed point of τ_Δ, i.e.,*

(i) *It is a fixed point: $\tau_\Delta(\mathit{Val}_\Delta) = \mathit{Val}_\Delta$*

(ii) *It is the least such: for all valuations V, if $\tau_\Delta(V) = V$ then $Val_\Delta \sqsubseteq V$.*

Proof.

(i) If C is terminal then it is easy to see that $\tau_\Delta(Val_\Delta)(C) = Val_\Delta(C)$. For non-terminal C, if $C \vdash (C_1, \ldots, C_n)$ then

$$\begin{aligned}
\tau_\Delta(Val_\Delta)(C) &= \gamma_C(Val_\Delta(C_1), \ldots, Val_\Delta(C_n)) \\
&= \gamma_C(\lim_i\{\tau_\Delta^i(V_\perp)(C_1)\}, \ldots, \lim_i\{\tau_\Delta^i(V_\perp)(C_n)\}) \\
&= \lim_i\{\gamma_C(\tau_\Delta^i(V_\perp)(C_1), \ldots, \tau_\Delta^i(V_\perp)(C_n))\} \quad \text{(by continuity)} \\
&= \lim_i\{\tau_\Delta^{i+1}(V_\perp)(C)\} \\
&= Val_\Delta(C).
\end{aligned}$$

(ii) $V_\perp \sqsubseteq V$, so $\tau_\Delta^i(V_\perp) \sqsubseteq \tau_\Delta^i(V) = V$ for all $i \geq 0$. Hence $Val_\Delta \sqsubseteq V$.

Q.E.D.

Example 4 To see that a fixed point of τ_Δ need not be unique, consider a binary computation tree in which all paths, with a single exception, terminate at accepting configurations. The exception is the infinite path π that always branches to the right. We could make sure that each node in this tree has a distinct configuration. Assuming that all nodes are MIN-configurations, a fixed point valuation V_1 of the computation tree is where all nodes have value 1. Another fixed point valuation V_2 assigns each nodes in π to 0 but the rest has value 1. But the least fixed point valuation V_0 assigns to the value $\perp$ to each node on the path π and the value 1 to the rest. ■

Definition 6 An interval $I \subseteq [0, 1]$ is a *accepting* if $I \subseteq (\frac{1}{2}, 1]$. It is *rejecting* if $I \subseteq [0, \frac{1}{2})$; it is *undecided* if it is neither accepting nor rejecting. ■

Note that I is accepting/rejecting iff each $v \in I$ is greater/less than $\frac{1}{2}$. Similarly I is undecided iff $\frac{1}{2} \in I$.

Definition 7 (Acceptance rule for choice machines)
(i) Let w be a word in the input alphabet of a choice acceptor M, and Δ a set of configurations of M. The Δ-*value of* w, denoted $Val_\Delta(w)$, refers to $Val_\Delta(C_0(w))$ where $C_0(w)$ is the initial configuration of M on w. If $\Delta = \Delta(M)$, the set of all configurations of M, we write $Val_M(w)$ instead of $Val_{\Delta(M)}(w)$.
(ii) We say M *accepts*, *rejects* or *is undecided on* w according as $Val_M(w)$ is accepting, rejecting or undecided.
(iii) A machine is said to be *decisive* if every input word is either accepted or rejected; otherwise it is *indecisive*
(iv) The *language accepted by* M is denoted $L(M)$. The *language rejected by* M is denote $\overline{L}(M)$. Thus, M is decisive iff $\overline{L}(M) = \text{co-}L(M)$. ■

Convention. In the course of this section, we will introduce other types of fixed point valuations. It is helpful to realize that we use '*Val*' (with various subscripts) only to denote valuations that are least fixed points of the appropriate operators.

7.3.1 Tree Valuations and Complexity

To discuss complexity in general, we need an alternative approach to valuations, called 'tree valuations'. To emphasize the difference, the previous notion is also called 'configuration valuations'.

Configuration valuations allows us to define the notion of acceptance or rejection. They can also define space complexity: thus, we say that M accepts input w in space h if $Val_\Delta(w)$ is accepting with Δ comprising all those configurations of M that uses space $\leq h$, Unfortunately, configuration valuations are not suited for time complexity. To see why, note that they are unable to distinguish between different occurrences of the same configuration C in computation trees. Suppose C occurs at two nodes (say, u_1 and u_2) of a computation tree. Assume the depth of u_1 is less than the depth of u_2. In a time-limited computation, we are interested in computation trees with bounded depths. and want "valuations" V of such trees which may distinguish between u_1 and u_2. For instance, if u_2 is a descendent of u_1 then u_1 typically have "more information" than u_2, and we want $V(u_2) \sqsubseteq V(u_1)$.

The following treatment is abbreviated since it imitates the preceding development.

Definition 8 Fix a choice machine M and any input w.
(i) The *complete computation tree* $T_M(w)$ of M on w is an ordered tree whose nodes are labeled by configurations from Δ_M such that the root is labeled with the initial configuration $C_0(w)$, and whenever a node u is labeled by some C and $C \vdash (C_1, \ldots, C_n)$ then u has n children $u_1, \ldots, u_n$ which are ordered so that u_i is labeled by C_i. We write $u \vdash (u_1, \ldots, u_n)$ in this case. By abuse of terminology, we sometimes identify a node u with its label C.
(ii) A tree T' is a *prefix* of another tree T if T' is obtained from T by pruning[10] some subset of nodes of T. In particular, if T' is non-empty then the root of T' is the root of T. If T is labeled, then T' has the induced labeling.
(iii) A *computation tree* T of w is a prefix of the complete computation tree $T_M(w)$; also, $T_M(w)$ is the *completion* of T.
(iv) A *(tree) valuation* on a computation tree T is a function V that assigns a value $V(u) \in \mathit{INT}$ for each node u in the completion $T_M(w)$ of T, with the property that nodes not in T are assigned $\perp$. We also call V *a tree valuation of* w. If V, V' are valuations of w then we define

$$V \sqsubseteq V'$$

if $V(u) \sqsubseteq V'(u)$ for all $u \in T_M(w)$.
(v) The *bottom valuation*, denoted $V_\perp$, assigns each node of $T_M(w)$ to $\perp$. Clearly $V_\perp$ is the $\sqsubseteq$-minimum tree valuation, for any given w.
(vi) The operator τ_T transforms a valuation V on T to a new valuation $\tau_T(V)$ on T as follows: for each node $u \in T_M(w)$,

$$\tau_T(V)(u) = \begin{cases} \perp & \text{if } u \text{ is a YO-node, or } u \notin T, \\ 1 & \text{else if } u \text{ is a YES-node,} \\ 0 & \text{else if } u \text{ is a NO-node,} \\ \gamma_u(V(u_1), \ldots, V(u_n)) & \text{else if } u \vdash (u_1, \ldots, u_n). \end{cases}$$

Let the least fixed point of τ_T be denoted by Val_T. In fact, $\tau_T^i(V_\perp) \sqsubseteq \tau_T^{i+1}(V_\perp)$ for all $i \geq 0$ and we have

$$\mathit{Val}_T = \lim\{\tau_T^i(V_\perp) \; : \; i \geq 0\}.$$

(vii) A computation tree T is *accepting/rejecting/undecided* if $\mathit{Val}_T(u_0)$ is accepting/rejecting/undecided where u_0 is the root of T. ■

We claim that Val_T is the least fixed point without proof because it is proved exactly as for configuration valuations; furthermore, the next section gives another approach.

Let us say a word w is accepted or rejected by M in the 'new sense' if there is an accepting/rejecting tree for w. We next show the new sense is the same as the old. First we introduce a notation: if $\Delta \subseteq \Delta(M)$ then let

$$T_\Delta(w)$$

denote the largest computation tree T of w all of whose nodes are labeled by elements of Δ. It is not hard to see that this tree is uniquely defined, and is non-empty if and only if the initial configuration $C_0(w)$ is in Δ. Equivalence of the two senses of acceptance amounts to the following.

Lemma 7 *Fix M and input w.*
(a) If T is an accepting/rejecting computation tree of w then $\mathit{Val}_\Delta(w)$ is also accepting/rejecting, where Δ is the set of labels in T.
(b) Conversely, for any $\Delta \subseteq \Delta(M)$, if $\mathit{Val}_\Delta(w)$ is accepting/rejecting then $T_\Delta(w)$ is also accepting/rejecting.

Its proof is left as an exercise. It follows that a word cannot be both accepted *and* rejected in the tree sense. For, if T is accepting/rejecting tree for w, and Δ are the labels of T, then $\mathit{Val}_\Delta(w)$ is accepting/rejecting. Hence $\mathit{Val}_M(w)$ is accepting/rejecting.

Definition 9 (Acceptance Complexity) Let r be any extended real number.
(i) We say that M *accepts x in time r* if there is an accepting tree T on input x whose nodes are at level at most r (the root is level 0).
(ii) We say M *accepts x in space r* if there is an accepting tree T on input x whose nodes each uses space at most

[10]To *prune* a node u from T means to remove from T the node u and all the descendents of u. Thus, if we prune the root of T, we an empty tree.

r.
(iii) We say M *accepts x in reversal r* if there is an accepting tree T on input x such that each path in the tree makes at most r reversals.
(iv) A computation path $C_1 \vdash C_2 \vdash \cdots \vdash C_m$ makes *(at least) r alternations* if there are $k = \lfloor r \rfloor + 1$ configurations

$$C_{i(1)}, C_{i(2)}, \ldots, C_{i(k)}$$

$1 \le i(1) < i(2) < \cdots < i(k) \le m$ such that each $C_{i(j)}$ $(j = 1, \ldots, k)$ is either a MIN- or a MAX-configuration, and if $k(j) < i(j)$ is the largest index such that $C_{k(j)}$ is either a MIN- or MAX-configuration then $C_{k(j)}$ and $C_{i(j)}$ makes different choices (one makes a MIN- and the other a MAX-choice) if and only if there is an even number of NOT-configurations between them along the path. We say M *accepts x in r alternations* if there is an accepting tree T on input x such that no path in the tree makes $1 + r$ alternations.
(v) Let r_1, r_2, r_3 be extended real numbers. We say M *accepts x in simultaneous time-space-reversal* (r_1, r_2, r_3) if there is an accepting tree T that satisfies the requirements associated with each of the bounds r_i $(i = 1, \ldots, 3)$ for the respective resources.
(vi) For complexity functions f_1, f_2, f_3, we say that M *accepts in simultaneous time-space-reversal* (f_1, f_2, f_3) if for each $x \in L(M)$, M accepts x in simultaneous time-space-reversal $(f_1(|x|), f_2(|x|), f_3(|x|))$. This definition extends to other simultaneous bounds. ■

We have just introduced a new resource 'alternation'. Unlike time, space and reversals, this resource is mode-dependent. For example, the machine in the palindrome example above has one alternation and nondeterministic machines has no alternations. We have a similar monotonicity property for tree valuations: if T is a prefix of T' then

$$Val_T \sqsubseteq Val_{T'}.$$

In consequence, we have:

Corollary 8 *If M accepts an input in time r then it accepts the same input in time r' for any $r' > r$. Similarly for the other resources.*

Our definition of accepting in time r is phrased so that the accepting tree T need not include all nodes at levels up to $\lfloor r \rfloor$. Because of monotonicity, it may be more convenient to include all nodes up to level $\lfloor r \rfloor$. But when other resource bounds are also being considered, we may no longer be free to do this.

The following result is fundamental:

Theorem 9 (Compactness) *If a choice machine M accepts a word x then it has a finite accepting tree on input x. Similarly, if M rejects a word, then there is a finite rejecting tree.*

The proof will be deferred to the next section. Thus, if an input is accepted, then it is accepted in finite amounts of time, space, etc. This result implies that the complexity measures such as time, space, reversals or alternation are Blum measures (chapter 6, section 8).

On rejection and running complexity. The above definitions of complexity is concerned with accepted inputs only, and no assumptions on the computation of M are made if $w \notin L(\mathrm{M})$. In other words, we have been discussing *acceptance complexity*. We now introduce *running complexity* whose general idea is that complexity bounds apply to rejected as well as accepted words. Should running complexity allow indecision on any input? Our definition disallows this.

Definition 10 (Running time complexity) Fix a choice machine M.
(i) We say M *rejects* an input w in k steps if there is a rejecting tree of M on w whose nodes have level at most k.
(ii) For any complexity function $t(n)$, we say M *rejects in time* $t(n)$ if for all rejected inputs w, M rejects w in time $t(|w|)$.
(iii) M *runs in time* (t, t') if each input of length n is either accepted in time $t(n)$ or rejected in time $t'(n)$. If $t = t'$, we simply say M runs in time t. ■

This definition extends naturally to other resources. Note that if M has a running time that is finite, *i.e.*, $t(n) < \infty$ for all n, then it is decisive. Thus, we can alternatively say that M is *halting* if it is decisive.

Complexity classes. We are ready to define complexity classes for choice modes. Our previous convention for naming complexity classes extends in a natural way: First note that our notation for complexity classes such as $NTIME(F)$ or $D\text{-}TIME\text{-}REVERSAL(F, F')$ has the general format

$$Mode\text{-}Resources\ (\ Bounds\ )$$

where *Mode* is either N or D, *Resources* is a sublist of *time*, *space*, *reversal* and *Bounds* is a list of (families of) complexity functions. The complexity class defined by choice machines can be named using the same format: we only have to add symbols for the new modes and resources. The new mode symbols (see the last column of Figure 7.3) are

$$Pr, \quad A, \quad Ip, \quad PrA, \quad St, \quad StA$$

denoting (respectively) the probabilistic, alternating, interactive proof, probabilistic-alternating, stochastic, stochastic-alternating modes. We have one new resource, with symbols[11]

$$\mathit{ALTERNATION} \text{ or } \mathit{ALT}.$$

Example 5
(i) Thus $\mathit{PrTIME}(n^{O(1)})$ denotes the class of languages accepted in polynomial time by probabilistic machines. This class is usually denoted PP.
(ii) The class $\mathit{IpTIME}(n^{O(1)})$ contains the class usually denoted IP in the literature. If we introduce (see next chapter) the notion of bounded-error decision, indicated by the subscript 'b', then we have

$$IP = \mathit{IpTIME}_b(n^{O(1)}).$$

(iii) If F, F' are families of complexity functions, $\mathit{PrA\text{-}TIME\text{-}SPACE}\,(F, F')$ denotes the class of languages that can be accepted by PAMs in simultaneous time-space (t, s) for some $t \in F, s \in F'$.
(iv) We will write $\mathit{A\text{-}TIME\text{-}ALT}(n^{O(1)}, O(1))$ for the class of languages accepted by alternating machines in polynomial time in some arbitrary but constant number of alternations. This class is denoted PH and contains precisely the languages in the polynomial-time hierarchy (chapter 9). ■

Example 6 Note that $\{\otimes\}$-machines (respectively, $\{\oplus\}$-machines) are equivalent to $\{\wedge\}$-machines ($\{\vee\}$-machines). A more interesting observation is that the probabilistic mode is at least as powerful as nondeterministic mode:

$$\mathit{N\text{-}TIME\text{-}SPACE\text{-}REVERSAL}(t, s, r) \subseteq \mathit{Pr\text{-}TIME\text{-}SPACE\text{-}REVERSAL}(t+1, s, r)$$

for any complexity functions t, s, r. To see this, let N be any nondeterministic machine that accepts in time-space (t, s). Let M be the following probabilistic machine: on input w, first toss a coin. If tail, answer YES; otherwise simulate N and answer YES iff N answers YES. The jargon 'toss a coin and if tail then do X, else do Y' formally means that the machine enters a TOSS-state from which there are two next configurations: in one configuration it does X and in the other choice it does Y. The reader may verify that M accepts $L(N)$ in time-space-reversal $(t+1, s, r)$. ■

7.4 Basic Results

In the last section, the least fixed point tree valuation Val_T for a computation tree T is obtained by repeated application of the operator τ_T to the bottom valuation $Val_\perp$. We now obtain Val_T in an alternative, top-down way.

For each integer $m \geq 0$, let T_m denote the prefix of T obtained by pruning away all nodes at level $m+1$. Thus T_0 consists of just the root of T. By monotonicity,

$$Val_{T_m} \sqsubseteq Val_{T_{m+1}}.$$

Lemma 10
(i) For any finite computation tree T, the fixed point of τ_T is unique (and, a fortiori*, equal to the least fixed point Val_T). Moreover, this fixed point is easily computed 'bottom-up' (e.g., by a postorder traversal of T).*
(ii) For any computation tree T (finite or not), the valuation

$$V_T^* := \lim\{\, Val_{T_m} \ : \ m \geq 0\}$$

is a fixed point of τ_T.
(iii) V_T^* is equal to the least fixed point, Val_T.
(iv) A computation tree T is accepting/rejecting if and only if it has a finite prefix that is accepting/rejecting.

[11]Since "alternation" is the name of a mode as well as of a resource, awkward notations such as $A\text{-}ALT(f(n))$ arise.

Proof. (i) This is seen by a bottom-up examination of the fixed point values at each node.
(ii) If u is a leaf then clearly $\tau_T(V_T^*)(u) = V_T^*(u)$. Otherwise, let $u \vdash (u_1, \ldots, u_n)$.

$$\begin{aligned}
\tau_T(V_T^*)(u) &= \gamma_u(V_T^*(u_1), \ldots, V_T^*(u_n)) \\
&= \gamma_u(\lim_{m\geq 0}\{Val_{T_m}(u_1)\}, \ldots, \lim_{m\geq 0}\{Val_{T_m}(u_n)\}) \\
&= \lim_{m\geq 0}\{\gamma_u(Val_{T_m}(u_1), \ldots, Val_{T_m}(u_n))\} \\
&= \lim_{m\geq 0}\{Val_{T_m}(u)\} \\
&= V_T^*(u)
\end{aligned}$$

This proves that V_T^* is a fixed point of τ_T.
(iii) Since Val_T is the least fixed point, it suffices to show that $V_T^* \sqsubseteq Val_T$. This easily follows from the fact that $V_{T_m} \sqsubseteq Val_T$.
(iv) If a prefix of T is accepting/rejecting then by monotonicity, T is accepting/rejecting. Conversely, suppose T is accepting/rejecting. Then the lower bound of $Val_T(u_0)$ is greater/less than $\frac{1}{2}$, where u_0 is the root of T. By the characterization of Val_T in part (iii), we know that the lower/upper bound of $Val_{T_m}(u_0)$ is monotonically non-decreasing/non-increasing and is greater/less than $\frac{1}{2}$ in the limit as m goes to infinity. Then there must be a first value of m when this lower/upper bound is greater/less than $\frac{1}{2}$. This m gives us the desired finite prefix T_m of T. **Q.E.D.**

This lemma has a two-fold significance: First, part (iv) proves the compactness theorem in the last section. Second, part (iii) shows us an constructive way to compute Val_T, by approximating it from below by Val_{T_m} with increasing m. This method is constructive (in contrast to the $\tau_T^m(V_\perp)$ approximation) because T_m is finite for each m, and the proof of part (i) tells us how to compute Val_{T_m}.

The following lemma is useful for stochastic-alternating computations:

Lemma 11 *Let T be a computation tree of a choice machine M on w, and $i \geq 0$.*
(i) If M is a probabilistic-alternating machine then $\tau_T^{i+1}(V_\perp)(u) = [x, y]$ implies $2^i x$ and $2^i y$ are integers.
(ii) If M is a stochastic-alternating machine then $2^{2^i} x$ and $2^{2^i} y$ are integers.

We leave the proof as an exercise.

We now have the machinery to show that the language accepted by a B-choice machine M is recursively enumerable provided each function in B is computable. To be precise, assume a suitable subset X of $[0,1]$ consisting of all the 'representable' numbers, and call an interval representable if its endpoints are representable. We assume $0, 1 \in X$. We require X to be dense in $[0,1]$ (for example, $X \subseteq [0,1]$ is the set of rational numbers or X is the set of "binary rationals" which are rationals with finite binary expansion). We say $f \in B$ is computable (relative to the representation of X) if it returns a representable value when given representable arguments; moreover this value is computable by some recursive transducer.

Theorem 12 *Let B be a basis set each of whose functions are computable.*
a) The class of languages accepted by B-machines is precisely the class RE of recursively enumerable languages.
b) The class of languages accepted by decisive B-machines is precisely the class REC.

Proof. a) Languages in RE are accepted by B-choice machines since B-choice machines are generalizations of ordinary Turing machines. Conversely, let M be a B-choice machine and w an input word. To show that $L(\mathrm{M})$ is in RE, it is sufficient to give a deterministic procedure for checking if $w \in L(\mathrm{M})$, where the procedure is required to halt only if w is in $L(\mathrm{M})$. The procedure computes, for successive values of $m \geq 0$, the value $Val_{T_m}(u_0)$ where T_m is the truncation of $T_M(w)$ below level m and u_0 the root. If T_m is accepting for any m, the procedure answers YES. If T_m is non-accepting for all m, the procedure loops. Lemma 10 not only justifies this procedure, but it also shows how to carry it out: the values $Val_{T_m}(u)$ of each node $u \in T_m$ is computed in a bottom-up fashion. The computability of the basis functions B ensures this is possible.

b) We leave this as an exercise. **Q.E.D.**

One can view the theorem as yet another confirmation of Church's thesis. Our next result shows that negation $\neg$ can be avoided in stochastic-alternating machines at the cost of an increase in the number of states. The following generalizes a result for alternation machines in [3].

Theorem 13 *For any stochastic-alternating acceptor* M, *there is a stochastic-alternating acceptor* N *such that* N *has no NOT-states, $L(\mathrm{M}) = L(\mathrm{N})$, and for all w and $t, s, r, a \geq 0$:* M *accepts w in (time, space, reversal, alternation) (t, s, r, a) iff* N *accepts w in (time, space, reversal, alternation) (t, s, r, a).*

Proof. The idea is to use de Morgan's law to move negation to the leaves of the computation tree. Let M be any SAM. We construct a SAM N satisfying the requirements of the theorem. For each state q of M, there are two states q^+ and q^- for N. For any configuration C of M, let C^+ (resp., C^-) denote the corresponding configuration of N where q^+ (resp., q^-) is substituted for q. In N, we regard q_0^+, q_Y^+ and q_N^+ (respectively) as the initial, YES and NO states. However, we identify q_Y^-, q_N^- (respectively) with the NO, YES states (note the role reversal). Of course, the technical restriction does not permit two YES- or two NO-states, so we will identify them ($q_Y^+ = q_N^-, q_N^+ = q_Y^-$). The functions $\gamma(q^+), \gamma(q^-)$ assigned to the states in N are defined from $\gamma(q)$ in M as follows:

$$\gamma(q^+) = \begin{cases} \gamma(q) & \text{if } \gamma(q) \neq \neg \\ \iota & \text{if } \gamma(q) = \neg \end{cases}$$

$$\gamma(q^-) = \begin{cases} \iota & \text{if } \gamma(q) = \neg \\ \wedge & \text{if } \gamma(q) = \vee \\ \vee & \text{if } \gamma(q) = \wedge \\ \oplus & \text{if } \gamma(q) = \oplus \\ \oplus & \text{if } \gamma(q) = \otimes \\ \otimes & \text{if } \gamma(q) = \oplus \end{cases}$$

Hence N has no NOT-states. We now state the requirements on transitions of N (this easily translates into an explicit description of the transition table of N). Suppose that $C_1 \vdash C_2$. If C_1 is not a NOT-configuration then

$$C_1^+ \vdash C_2^+ \text{ and } C_1^- \vdash C_2^-.$$

If C_1 is a NOT-configuration then

$$C_1^+ \vdash C_2^- \text{ and } C_1^- \vdash C_2^+.$$

Our description of N is complete: there are no transitions besides those listed above.

Let T be an accepting computation tree of N for an input word w; it is easy to see that there is a corresponding computation tree $\widehat{T}$ for N with exactly the same time, space, reversal and alternation complexity. In fact there is a bijection between the nodes of T and $\widehat{T}$ such a node labeled C in T corresponds to one labeled C^+ or C^- in $\widehat{T}$. The fact that T and $\widehat{T}$ have identical alternating complexity comes from the carefully-crafted definition of N.

Our theorem is proved if we show that $\widehat{T}$ is accepting. Let T_m be the truncation of the tree T at levels below $m \geq 0$; $\widehat{T}_m$ is similarly defined with respect to $\widehat{T}$. For $h \geq 0$, let V_m^h denote the valuations on T_m given by h-fold applications of the operator τ_{T_m} to $\perp$:

$$V_m^h = \tau_{T_m}^h(V_\perp)$$

and similarly define $\widehat{V}_m^h = \tau_{\widehat{T}_m}^h(V_\perp)$. We now claim that for all m, h and $C \in T_m$,

$$V_m^h(C) = \begin{cases} \widehat{V}_m^h(C^+) & \text{if } C^+ \in \widehat{T} \\ \neg\widehat{V}_m^h(C^-) & \text{if } C^- \in \widehat{T} \end{cases}$$

Here, we have abused notation by identifying the configuration C with the node of T_m that it labels. But this should be harmless except for making the proof more transparent. If $h = 0$ then our claim is true

$$\perp = V_m^h(C) = \widehat{V}_m^h(C^+) = \neg\widehat{V}_m^h(C^-)$$

since $\neg\perp = \perp$. So assume $h > 0$. If C is a leaf of T, it is also easy to verify our claim. Hence assume C is not a leaf. Suppose $C^- \vdash (C_1^-, C_2^-)$ occurs in $\widehat{T}$. Then

$$\begin{aligned} V_m^h(C) &= \gamma(C)(V_m^{h-1}(C_1), V_m^{h-1}(C_2)) \\ &= \gamma(C)(\neg\widehat{V}_m^{h-1}(C_1^-), \neg\widehat{V}_m^{h-1}(C_2^-)) \quad \text{(by induction)} \\ &= \neg\gamma(C^-)(\widehat{V}_m^{h-1}(C_1^-), \widehat{V}_m^{h-1}(C_2^-)) \quad \text{(de Morgan's law for } \gamma(C)) \\ &= \neg\widehat{V}_m^h(C^-). \end{aligned}$$

Similarly, we can show $V_m^h(C) = \widehat{V}_m^h(C^+)$ if $C^+ \vdash (C_1^+, C_2^+)$ occurs in $\widehat{T}$. We omit the demonstration in case C is a NOT-configuration. Finally, noting that $V_m^{m+1} = Val_{T_m}$ and $\widehat{V}_m^{m+1} = Val_{\widehat{T}_m}$, we conclude that $Val_{T_m} = Val_{\widehat{T}_m}$. It follows $\widehat{T}$ is accepting. **Q.E.D.**

Consequence of eliminating negation. This proof also shows that negation can be eliminated in alternating machines and in PAMs. With respect to SAMs without negation, the use of intervals in valuations can be replaced by ordinary numbers in $[0,1]$ *provided we restrict attention to acceptance complexity.* A valuation is now a mapping from $\Delta \subseteq \Delta(\mathrm{M})$ into $[0,1]$. Likewise, a tree valuation assigns a real value in $[0,1]$ to each node in a complete computation tree. We now let $V_\perp$ denote the valuation that assigns the value 0 to each configuration or node, as the case may be. The operator τ_Δ or τ_T on valuations is defined as before. Their least fixed point is denoted Val_Δ or Val_T as before. The connection between the old valuation V and the new valuation V' is simply that $V'(C)$ or $V'(u)$ (for any configuration C or node u) is equal to the lower bound of the interval $V(C)$ or $V(u)$. When we discuss running complexity, we need to consider the upper bounds of intervals in order to reject an input; so we are essentially back to the entire interval.

Convention for this chapter. In this chapter, we only consider alternating machines, PAMs and SAMs with no NOT-states. We are mainly interested in *acceptance complexity.* In this case, we may restrict valuations take values in $[0,1]$ instead of in *INT* (we call these real values *probabilities*). With this convention, the acceptance rule becomes:

M accepts a word w iff the probability $Val_M(w)$ is greater than $\frac{1}{2}$.

■

It is sometimes convenient to construct SAMs *with* NOT-states, knowing that they can be removed by an application of the preceding theorem.

Suppose we generalize SAMs by allowing the k-ary versions of the alternating-stochastic functions:

$$B_k := \{\max_k, \min_k, \underset{k}{\oplus\!\!\!\!\!\bigcirc}, \otimes_k, \oplus_k\},$$

for each $k \geq 2$. For example,

$$\begin{aligned} \underset{3}{\oplus\!\!\!\!\!\bigcirc}(x,y,z) &= (x+y+z)/3, \\ \oplus_3(x,y,z) &= 1-(1-x)(1-y)(1-z). \end{aligned}$$

Consider generalized SAMs whose basis set is $\cup_{k\geq 2} B_k$. Note that even though the basis set is infinite, each generalized SAM uses only a finite subset of these functions. It is easily seen that with this generalization for the alternating choices ($\max_k$, $\min_k$ and $\otimes_k$), the time complexity is reduced by at most a constant factor. It is a little harder (Exercise) to show the same for the stochastic choices ($\oplus\!\!\!\!\!\bigcirc_k, \oplus_k, \otimes_k$).

A useful technical result is tape reduction for alternating machines. The following is from Paul, Praus and Reischuk [18].

Theorem 14 *For any k-tape alternating machine M accepting in time-alternation $(t(n), a(n))$, there is a simple alternating machine N accepting the same language and time-alternation $(O(t(n)), a(n)+O(1))$. Here N is 'simple' as in 'simple Turing machines', with only one work-tape and no input tape.*

This leads, in the usual fashion, to a hierarchy theorem for alternating time:

Theorem 15 *Let $t(n)$ be constructible and $t'(n) = o(t(n))$. Then*

$$ATIME(t) - ATIME(t') \neq \emptyset.$$

We leave both proofs as exercises.

Theorem 16 (Space compression) *Let B be any basis set. Then the B-choice machines have the space compression property. More precisely, if M is a B-choice machine accepting in space $s(n)$ then there is another N which accepts the same language in space $s(n)/2$. Furthermore, N has only one work-tape.*

Proof. We only sketch the proof, emphasizing those aspects that are not present in the proof of original space compression theorem in chapter 2. As before, we compress 2 symbols from each of the k work-tapes of M into one *composite symbol* (with k tracks) of N. We show how N simulates one step of M: suppose M is in some configuration C and $C \vdash (C_1, \ldots, C_m)$. Assume that the tape head of N is positioned at the leftmost non-blank cell of its tape. By deterministically making a rightward sweep across the non-blank part of its work-tape, N can remember in its finite state control the two composite symbols adjacent to the currently scanned cell *in each track*: the cells of M corresponding to these remembered symbols constitute the *current neighborhood.* In the original proof, N makes a leftward sweep back to its starting position, updating the contents of the current neighborhood. The new twist is that there are now m ways to do the updating. N can use choice to ensure that each of these possibilities are covered. More precisely, before making the return sweep, N enters a state q such that $\gamma(q) = \gamma(C)$ and then N branches into m different states, each corresponding to a distinct way to update the current neighborhood. Then N can make a deterministic return sweep on each of the m branches. By making some adjustments in the finite state of N, we may ensure that N uses space $s(n)/2$. Further, N is also a B-machine by construction. **Q.E.D.**

For any family of functions B over INT, let B^* denote the *closure* of B (under function composition): B^* is the smallest class of functions containing B and closed under function composition.[12] For example, the function $h(x, y, z) = x \oplus (y \oplus z) = x/2 + y/4 + z/4$ is in the closure of the basis $\{\oplus\}$. The closure of a basis set is also a basis set (in fact, an infinite set).

THEOREM 17 (LINEAR SPEEDUP) *Let B be an admissible family which is closed under function composition, $B = B^*$. Then the B-choice machines have the linear speedup property. More precisely, if M is a B-choice machine accepting in time $t(n) > n$ then there is another N which accepts the same language in time $n + t(n)/2$.*

Proof. The proof is similar to that for ordinary Turing machines in chapter 2, so we only emphasize the new aspects. The new machine N has $k+1$ work-tapes if M has k. Each tape cell of N encodes up to $d > 1$ (for some d to be determined) of the original symbols. N spends the first n steps making a compressed copy of the input. Thereafter, N uses 8 steps to simulates d steps of M. In general, suppose that M is in some configuration C and d moves after C, there are m successor configurations $C_1, \ldots, C_m$ (clearly m is bounded by a function of M and d only). Suppose that the complete computation tree in question is T. The value $Val_T(C)$ is given by

$$f(Val_T(C_1), \ldots, Val_T(C_m))$$

where $f \in B$ since B is closed under function composition. We show how N can determine this f: first N takes 4 steps to determine the contents of the 'current neighborhood' (defined as in the original proof). From its finite state constrol, N now knows f and each C_i ($i = 1, \ldots, m$). So at the end of the fourth step, N could enters a state q where $\gamma(q) = f$ and such that q has m successors, corresponding to the C_i's. In 4 more steps, N deterministically updates its current neighborhood according to each C_i. It is clear that by choosing $d = 16$, N accepts in time $n + t(n)/2$. One minor difference from the original proof: previously the updated tapes represent the configuration at the first time some tape head leaves the current neighborhood, representing at least d steps of M. Now we simply simulate *exactly* d steps and so it is possible that a tape head remain in the current neighborhood after updating. **Q.E.D.**

As corollary, if we generalize alternating machines by replacing the usual basis set $B = \{\wedge, \vee, \neg\}$ by its closure B^*, then the generalized alternating time classes enjoy the time speedup property. A similar remark holds for the other modes.

7.5 Alternating Time versus Deterministic Space

We begin the study of alternating machines. This section points out strong similarities between alternating time and deterministic space. This motivates a variation of choice machines called the addressable-input model.

Example: Let PH denote the class of languages accepted by polynomial time alternating machines that makes a constant number of alternations. Consider the following problem MIN-FORMULA of recognizing if a given Boolean formula F is minimal. That is, for all F', if $|F'| < |F|$ then $F' \not\equiv F$. It is not obvious that this problem is in $NP \cup$ co-NP. But we can easily show that MIN-FORMULA$\in PH$: on input $F(x_1, \ldots, x_n)$, we universally guess another formula $F'(x_1, \ldots, x_n)$ such that $|F'| < |F|$ and then existentially guess an assignment $a_i \mapsto x_i$ and accepts if $F'(a_1, \ldots, a_n) \neq F(a_1, \ldots, a_n)$.

We first prove the following result of Chandra, Kozen and Stockmeyer:

THEOREM 18 *For all t, $ATIME(t) \subseteq DSPACE(t)$.*

Proof. Let M be an alternating machine accepting in time t. We describe a deterministic N that simulates M in space t. Let w be the input and N computes in successive stages. In the mth stage ($m = 1, 2, 3, \ldots$), N computes Val_{T_m} where T_m is the truncation of the complete computation tree $T_M(w)$ at levels below m. For brevity, write Val_m for Val_{T_m}. If T_m is accepting then N accepts, otherwise it proceeds to the next stage. So if w is not in $L(M)$ then N will not halt.

[12]More precisely, if $f \in B^*$ is a function of arity k and g_i ($i = 1, \ldots, k$) are functions of arity m_i in B^* then $f(g_1(\bar{x}_1), \ldots, g_k(\bar{x}_2))$ is a function in B^* of arity $p \leq \sum_{i=1}^{k} m_i$ where $\bar{x}_i$ is a sequence of m_i variables and p is the number of distinct variables in $\bar{x}_1\bar{x}_2\cdots\bar{x}_k$.

To complete the proof, we show that the mth stage can be carried out using $O(m)$ space. We describe a procedure to compute the values $Val_m(C)$ of the nodes C in T_m in a post-order manner. The structure of this search is standard. We inductively assume that the tapes of N contain three pieces of information when we visit a configuration C:

(a) The configuration C. This requires $O(m)$ space. In particular, the input head position can be recorded in $O(\log m)$ space (rather than $O(\log |w|)$ space).

(b) A representation of the path $\pi(C)$ from the root of T_m to C. If the path has length k, we store a sequence of k tuples from the transition table of M where each tuple represents a transition on the path $\pi(C)$. The space for storing the k tuples is $O(m)$ since $k \leq m$. These tuples allow us to "backup" from C to any of its predecessors C' on the path (this simply means we reconstruct C' from C).

(c) All previously computed values $Val_m(C')$ where C' is the child of some node in the path $\pi(C)$. The space is $O(m)$, using the fact that $Val_m(C')$ is 0 or 1.

We maintain this information at each "step". A step (at current configuration C) either involves descending to a child of C or backing up from C to its parent. We descending to a child of C provided C is not a leaf and at least one child of C has not yet been visited. Maintaining (a)-(c) is easy in this case. So suppose we want to backup from C to its parent C'. We claim that $Val_m(C)$ can be determined at this moment. This is true if C is a leaf of T_m. Otherwise, the reason we are backing up to C' is because we had visited both children C_1, C_2 of C. But this meant we had just backed up from (say) C_2 to C, and inductively by our claim, we have determined $Val_m(C_2)$. From (c), we also know $Val_m(C_1)$. Thus we can determine $Val_m(C)$, as claimed. Eventually we determine the value of Val_m at the root. **Q.E.D.**

Discussion. (A) This theorem shows that *deterministic space is at least as powerful as alternating time.* This suggests new results as follows: take a known simulation by deterministic space and ask if it can be improved to an alternating time simulation. This methodology has proven fruitful and has resulted in a deeper understanding of the space resource. Thus, in section 8, a known inclusion $DTIME(t) \subseteq DSPACE(t/\log t)$ was sharped to $DTIME(t) \subseteq ATIME(t/\log t)$. This strengthening apparently lead to a simplification (!) of the original proof. This paradox is explained by the fact that the control mechanism in alternating computation is "in-built"; an alternating simulation (unlike the original space simulation) need not explicitly describe this mechanism.

(B) In fact there is evidence to suggest that alternating time and deterministic space are very similar. For instance, we prove (§7.7) a generalization of Savitch's result, which yields the corollary

$$NSPACE(s) \subseteq ATIME(s^2).$$

This motivates another class of new results: given a known result about deterministic space, try to prove the analogue for alternating time, or vice-versa. For instance, the last section shows a tape-reduction and a hierarchy theorem for alternating-time; the motivation for these results is that we have similar results for deterministic space. We now give another illustration. In chapter 2, we show that $DSPACE_r(s)$ is closed under complementation for all s finite (*i.e.*, $s(x) < \infty$ whenever defined). We ask for a corresponding result for $ATIME_r(t)$. (Note that the subscript 'r' indicates running complexity.) As it turns out, this result[13] is rather easy for alternating time:

THEOREM 19 *For all complexity function $t(n)$,*

$$ATIME_r(t) = \text{co-}ATIME_r(t).$$

Similarly, the following time classes are closed under complementation

$$PrTIME_r(t), StTIME_r(t), PrA\text{-}TIME_r(t), StA\text{-}TIME_r(t).$$

Proof. Recall the construction in theorem 13 of a machine N without negation from another machine M that may have negation. Now let M be an alternating machine. Suppose that we make q_0^- (instead of q_0^+) the start state of N but $q_Y^+ = q_N^-$ remains the YES state. On re-examination of the proof of theorem 13, we see that this N accepts co-$L(M)$. The proof for the other time classes are similar. **Q.E.D.**

(C) Continuing our discussion: by now it should be realized that the fundamental technique for space simulation is to 'reuse space'. This usually amounts to cycling through an exponential number of possibilities using the same space. In alternating time, the corresponding technique is to make exponentially many universal or existential

[13] Paul and Reischuk show that if t is time-constructible then $ATIME(t) = \text{co-}ATIME(t)$.

choices. While a deterministic space search proceeds from what is known to what is unknown, alternating time search proceeds in reverse direction: it guesses the unknown and tries to reduce it to the known. This remark may be clearer by the end of this chapter.

(D) We should caution that the research programs (A) and (B) have limitations: although the deterministic space and alternating time are similar, it is unlikely that they are identical. Another fundamental difficulty is that whereas sublinear deterministic space classes are important, it is easy to see that alternating machines do not allow meaningful sublinear time computations. This prompted Chandra, Kozen and Stockmeyer to suggest[14] a variation of alternating machines which we now extend to choice machines:

Definition 11 *(Addressable-input Machine Model)* An *addressable-input* choice machine M is one that is equipped with an extra *address* tape and two distinguished states called the *READ* and *ERROR* states. The address tape has a binary alphabet whose content is interpreted as an integer. Whenever M enters the READ state, the input head is instantaneously placed at the (absolute) position indicated by the address tape. If this position lies outside the input word, the ERROR state will be entered and no input head movement occurs. In addition to this special way of moving the input head, our machine machine can still move and use the input head in the standard fashion.

We assume the address tape is one-way (and hence is write-only). Hence for complexity purposes, space on the address tape is *not* counted. We also assume that after exiting from the READ state, the contents of the address tape is erased, in an instant. ■

The addressable-input model is defined so that such machines are at least as powerful as *ordinary* choice machines. However, it is not excessively more powerful; for instance the preceding theorem theorem 18 holds even with this model of alternation **Exercise:**. This addressable-input model now admits interesting alternating time classes with complexity as small as $\log n$ (not $\log\log n$, unfortunately). We will be explicit whenever we use this version of choice machines instead of the ordinary ones.

Exercise

Exercise 7.5.1: (i) Consider the language comprising (D, k) where D is the distance matrix as in the travelling salesman problem (TSP) in Chapter 3, and k is the length of the shortest tour. Show that this language is in PH.
(ii) Construct a "natural" language that apparently needs 4 alternations. □

End Exercise

7.6 Simulations by Alternating Time

We present efficient simulations of other complexity resources by alternating time. We begin with an alternating time simulation of deterministic space and reversals.

Theorem 20 *Let t, r be complexity functions such that $t(n) \geq 1 + n$. Under the addressable-input model,*

$$D\text{-}TIME\text{-}REVERSAL(t, r) \subseteq ATIME(O(r \log^2 t)).$$

If $r(n) \log^2 t(n) \geq n$, then this result holds under the ordinary model.

Proof. Given a deterministic M accepting in time-reversal (t, r), we show an alternating machine N accepting the same language $L(M)$ in time $O(r \log^2 t)$.

Recall the concept of a (full) trace[15] in the proof of chapter 2. On any input w, N existentially chooses some integer $r \geq 1$ and writes r full traces

$$\tau_1, \tau_2, \ldots, \tau_r,$$

on tape 1. These are intended to be the traces at the beginning of each of the r phases. On tape 2, N existentially chooses the time t_i (in binary) of each τ_i,

$$t_1 < t_2 < \cdots < t_r.$$

[14]In this suggestion, they are in good company: historically, the read-only input tape of Turing machines was invented for a similar purpose.

[15]Briefly, the trace of a configuration in a computation path records its state and for each tape, the scanned symbol, the absolute head position and the head tendencies.

We may assume τ_r is the trace when the machine accepts. Note that τ_1 (which we may assume correct) is simply the trace of the initial configuration and so $t_1 = 0$. Relative to this sequence, we say that an integer $t \geq 0$ *belongs to phase* j if $t_j \leq t < t_{j+1}$.

Then N proceeds to verify τ_r. To do this, it writes on tape 3 the pairs (τ_{r-1}, t_{r-1}) and (τ_r, t_r) and invokes a procedure *TRACE*. N accepts if and only if this invocation is *successful* (i.e., the procedure accepts). In general, the arguments to *TRACE* are placed on tape 3, and they have the form

$$(\sigma, s_0), (\tau, t_0)$$

where $s_0 < t_0$ are binary integers lying in the range

$$t_i \leq s_0 < t_0 \leq t_{i+1}$$

for some $i = 1, \ldots, r-1$, and σ, τ are traces such that the head tendencies in σ and in τ_i agree, and similarly the head tendencies in τ and in τ_i agree (with the possible exception of $t_0 = t_{i+1}$ and $\tau = \tau_{i+1}$). Intuitively, $TRACE(\sigma, s_0, \tau, t_0)$ accepts if σ is the trace at time s_0, τ is the trace at time t_0, and there is a (trace of a) path from σ to τ. *TRACE* does one of two things:

(i) Suppose $s_0 + 1 < t_0$. Let $t' = \lfloor (s_0 + t_0)/2 \rfloor$. Now *TRACE* existentially chooses a trace τ' where the head tendencies in τ' agree with those of σ. Then it universally chooses to recursively call $TRACE(\sigma, s_0, \tau', t')$ and $TRACE(\tau', t', \tau, t_0)$.

(ii) Suppose $s_0 + 1 = t_0$. *TRACE* verifies τ can be derived from σ in one step of M, and any head motion is consistent with the head tendencies in σ. Of course, we allow the head tendencies to be different but only when σ is the last trace in a phase (it is easy to determine if this is the case). Any head motion in the σ to τ transition causes the corresponding tape cell in τ to be 'marked'. Note that the marked cells were written in some previous phase (unless they are blanks), and our goal is to verify their contents. Suppose that the tape symbols and head positions in the $k+1$ tapes of τ are given by

$$b_0, \ldots, b_k, \qquad n_0, \ldots, n_k.$$

Then *TRACE* universally chooses to call another procedure *SYMBOL* with arguments (i, b_i, n_i, t_0) for each cell n_i in tape i that is marked. Intuitively, $SYMBOL(i, b_i, n_i, t_0)$ verifies that just before time t_0, the tape symbol in cell n_i of tape i is b_i.

We now implement $SYMBOL(i', b', n', t_0)$. If $i' = 0$ then we want to check that the input symbol at position n' is b'. This can be done in $O(\log n)$ steps, using the input addressing ability of our alternating machines (note that $r \log^2 t > \log n$). Otherwise, suppose that t_0 belongs to phase j_0. We then existentially choose some j,

$$j = 0, \ldots, j_0 - 1,$$

some t' and a trace σ'. Intuitively, this means that cell n' in tape i' was last visited by σ' which occurs at time t' in phase j. We want the following to hold:
(a) $t_j \leq t' < t_{j+1} \leq t_0$.
(b) The head tendencies σ' and in τ_j agree.
(c) The head i' is in position n' scanning symbol b' in σ'.
(d) On each tape, the head position in σ' lies in the range of possible cell positions for that phase j.
(e) On tape i', cell n' is not visited in any of phases $j+1, j+2, \ldots, j_0$.
Conditions (a)-(e) can be directly verified using the information on tapes 1 and 2. Armed with σ' and t', we then universally choose one of two actions: either call $TRACE(\tau_j, t_j, \sigma', t')$ or $TRACE(\sigma', t', \tau_{j+1}, t_{j+1})$. If $t_j = t'$ then the first call is omitted.

Correctness. Let us show that *TRACE* and *SYMBOL* are correct. Suppose input w is accepted by M. Then it is not hard to see that N accepts. To show the converse, suppose N accepts w relative to some choice of traces $\tau_1, \ldots, \tau_r$ in tape 1 and times $t_1, \ldots, t_r$ on tape 2. Suppose the call $TRACE(\sigma, s_0, \tau, t_0)$ is successful and t_0 belongs to phase j. Then this call generates a trace-path

$$(\sigma_0, \ldots, \sigma_m) \tag{4}$$

from $\sigma_0 = \sigma$ to $\sigma_m = \tau$ (where $m = t_0 - s_0$) with the following properties:

1. σ_{i-1} derives σ_i for $i = 1, \ldots, m$ according to the rules of M.
2. Each pair (σ_{i-1}, σ_i) in turn generates at most $k+1$ calls to *SYMBOL*, one call for each "marked" cell in σ_i. Each of these calls to *SYMBOL* leads to acceptance. For this reason we call (4) a 'successful' trace-path for this call to *TRACE*.
3. Some of these calls to *SYMBOL* in turn calls *TRACE* with arguments belonging some phase ℓ $(1 \le \ell < j)$. We call any such phase ℓ a *supporting phase* of $TRACE(\sigma, s_0, \tau, t_0)$.

Notice that if phase ℓ is a supporting phase for some accepting call to *TRACE* then there must be two successful calls of the form

$$TRACE(\tau_\ell, t_\ell, \sigma', t') \quad \text{and} \quad TRACE(\sigma', t', \tau_{\ell+1}, t_{\ell+1}) \tag{5}$$

for some σ', t'. We claim:

(a) If $TRACE(\sigma, s_0, \tau, t_0)$ accepts and phase ℓ is a supporting phase of $TRACE(\sigma, s_0, \tau, t_0)$, then the traces $\tau_1, \ldots, \tau_{\ell+1}$ on tape 1 and times $t_1, \ldots, t_{\ell+1}$ on tape 2 are correct (i.e., τ_i is the trace at the beginning of the ith phase at time t_i for $i = 1, \ldots, \ell + 1$.)
(b) If, in addition, we have that $\sigma = \tau_j$ and $s_0 = t_j$ for some $j = 1, \ldots, r$, then τ is indeed the trace of the t_0th configuration in the computation path of M on input w. (Note that (b) implies the correctness of *SYMBOL*.)

We use induction on ℓ. Case $\ell = 1$: τ_1 is always correct and $t_1 = 0$. By (5), we see directly that there must be two successful calls of the form $TRACE(\tau_1, t_1, \sigma', t')$ and $TRACE(\sigma', t', \tau_2, t_2)$. One immediately checks that this implies τ_2, t_2 are correct. This proves (a). Part (b) is immediate.

Case $\ell > 1$: for part (a), again we know that there are two successful calls of the form (5). But notice that phase $\ell - 1$ is a supporting phase for the first of these two calls: this is because in τ_ℓ, some tape head made a reversal that this means that this head scans some symbol last visited in phase $\ell - 1$. Hence by induction hypothesis, the traces $\tau_1, \ldots, \tau_\ell$ and times $t_1, \ldots, t_\ell$ are correct. Furthermore, as in (4), we have a successful trace-path from τ_ℓ to $\tau_{\ell+1}$. Each trace (except for τ_ℓ) in the trace-path in turn generates a successful call to *SYMBOL* with arguments belonging to some phase less than ℓ, and by induction (b), these are correct. Thus $\tau_{\ell+1}$ and $t_{\ell+1}$ are correct. For part (b), we simply note that $j-1$ is a support phase for such a call to *TRACE* by the preceding arguments. So by part (a), τ_j and t_j are correct. Then we see that there is a trace-path starting from τ_j as in (4) that is correct. Part (b) simply asserts that the last trace in (4) is correct. This completes our correctness proof.

Complexity. The guessing of the traces τ_i and times t_i on tapes 1 and 2 takes alternating time $O(r \log t)$. If the arguments of *TRACE* belongs to phase j, then *TRACE* may recursively call itself with arguments belonging to phase j for $O(\log t)$ times along on a computation path of N. Then *TRACE* calls *SYMBOL* which in turn calls *TRACE* but with arguments belonging to phase $< j$. Now each call to *TRACE* takes $O(\log t)$ alternating steps just to set up its arguments (just to write down the head positions). Thus it takes $O(\log^2 t)$ alternating steps between successive calls to *SYMBOL*. In the complete computation tree, we make at most r calls to *SYMBOL* along any path. This gives an alternating time bound of $O(r \log^2 t)$. **Q.E.D.**

COROLLARY 21 $DREVERSAL(r) \subseteq ATIME(r^3)$

We next show that alternating time is at least as powerful as probabilistic time. The proof is based on the following idea: suppose a probabilistic machine accepts an input x in $m \ge 0$ steps and T is the computation tree. If T is finite and all its leaves happen to lie a fixed level $m \ge 0$ ($m = 0$ is the level of the root) then it is easily seen that T is accepting iff the number of accepting leaves is more than half of the total (i.e. more than 2^{m-1} out of 2^m). In general, T is neither finite nor will all the leaves lie in one level. But we see that if T_m is the truncation of T to level m, then T_m if accepting iff the sum of the "weights" of accepting leaves in T_m is more than 2^{m-1}. Here we define a leaf at level i $(0 \le i \le m)$ to have a weight of 2^{m-i}. It is now easy to simulate a probabilistic machine M that uses time $t(n)$ by a deterministic machine N using space $t(n)$, by a post-order traversal of the tree $T_{t(n)}$. But we now show that instead of deterministic space $t(n)$, alternating time $t(n)$ suffices.

THEOREM 22 *For all complexity functions t, $PrTIME(t) \subseteq ATIME(t)$.*

Proof. Let M be a probabilistic machine that accepts in time t. We describe an alternating machine N that accepts in time t. Let x be an input and T_m be the computation tree of M on x restricted to configurations at level at most m. For any configuration C in T_m, define

$$VAL_m(C) = 2^{m - level(C)}\, Val_{T_m}(C)$$

where Val_{T_m} is, as usual, the least fixed point valuation of T_m. We abuse notation with the usual identification of the nodes of T_m with the configurations labeling them. Thus if C is the root then $VAL_m(C) = 2^m Val_\Delta(C)$. If C is not a leaf, let C_L and C_R denote the two children of C. Observe that

$$VAL_m(C) = VAL_m(C_L) + VAL_m(C_R).$$

Regarding $VAL_m(C)$ as a binary string of length $m+1$, we define for $i = 0, \ldots, m$,

$$\begin{aligned} BIT_m(C,i) &:= i^{th} \text{ bit of } VAL_m(C) \\ CAR_m(C,i) &:= i^{th} \text{ carry bit of the summation } VAL_m(C_L) + VAL_m(C_R) \end{aligned}$$

where we assume that $i = 0$ corresponds to the lowest order bit. It is easy to see that the following pair of mutually recursive formulas hold:

$$\begin{aligned} BIT_m(C,i) &= BIT_m(C_L,i)\overline{\oplus}BIT_m(C_R,i)\overline{\oplus}CAR_m(C,i-1) \\ CAR_m(C,i) &= \lfloor\frac{BIT_m(C_L,i)) + BIT_m(C_R,i) + CAR_m(C,i-1)}{2}\rfloor \end{aligned}$$

Here, $\overline{\oplus}$ denotes[16] the exclusive-or Boolean operation: $b\overline{\oplus}b' = 1$ iff $b \neq b'$. If $i = 0$, $CAR_m(C, i-1)$ is taken to be zero.

If C is a leaf, we need not define $CAR_m(C,i)$ but

$$BIT_m(C,i) = \begin{cases} 1 & \text{if } i = m - level(C) \text{ and } C \text{ answers YES;} \\ 0 & \text{otherwise.} \end{cases}$$

To simulate M on input x, N first guesses the value $m = t(|x|)$ in unary in tapes $1, 2$ and 3. Note that M accepts iff $VAL_m(x) > 2^{m-1}$, iff there exists an i, $0 \leq i < m-1$, such that

$$\text{Either} \quad BIT_m(C_0(x), m) = 1 \tag{6}$$
$$\text{or} \quad BIT_m(C_0(x), m-1) = BIT_m(C_0(x), i) = 1. \tag{7}$$

N checks for either condition (6) or (7) by an existential choice. In the latter case, N makes a universal branch to check that $BIT_m(C_0(x), m-1) = 1$ and, for some existentially guessed unary integer $0 < i < m-1$, $BIT_m(C_0(x), i) = 1$.

It remains to describe the subroutine to verify $BIT_m(C,i) = b$ for any arguments C, i, b. It is assumed that just before calling this subroutine the following setup holds. N has the first m cells on tapes 1,2 and 3 marked out. Head 1,2 and 3 are respectively keeping track of the integers i, $level(C)$ and $i + level(C)$, in unary. Moreover, because of the marked cells, it is possible to detect when these values equals 0 or m. The configuration C is represented by the contents and head positions of tapes 4 to $k+3$ (k is the number of work-tapes of M) and the input tape. This setup is also assumed when calling the subroutine to verify $CAR_m(C,i) = b$.

With this setup, in constant time, N can decide if C is a leaf of T_m (i.e. either C is terminal or $level(C) = m$) and whether $i = m - level(C)$. Hence, in case C is a leaf, the subroutine can determine the value of $BIT_m(C,i)$ in constant time. If C is not a leaf, say $C \vdash (C_L, C_R)$, then N guesses three bits, b_1, b_2 and c such that

$$b = b_1\overline{\oplus}b_2\overline{\oplus}c.$$

It then universally branches to verify

$$BIT_m(C_L,i) = b_1,\ BIT_m(C_R,i) = b_2,\ CAR_m(C,i-1) = c.$$

It is important to see that N can set up the arguments for these recursive calls in constant time. A similar subroutine for CAR_m can be obtained.

It remains to analyze the time complexity of N. Define the function t_m to capture the complexity of BIT_m and CAR_m:

$$t_m(d,i) = \begin{cases} 1 & \text{if } d = m \\ 1 + t_m(d+1, 0) & \text{if } (d < m) \wedge (i = 0) \\ 1 + \max\{t_m(d+1,i), t_m(d,i-1)\} & \text{else.} \end{cases}$$

[16]Normally, $\oplus$ denotes exclusive-or. We put a bar over $\oplus$ to distinguish it from the probabilistic-or operator.

The last case corresponds to the recursive calls for BIT_m and CAR_m when $i > 0$. An examination of the recursive equations for BIT_m and CAR_m reveals that the times to compute $BIT_m(C, i)$ and $CAR_m(C, i)$ are each bounded by $O(t_m(d, i))$ where $d = level(C)$. On the other hand, it is easily checked that from the recursive equations that t_m satisfy

$$t_m(d, i) = m - d + i + 1 = O(m)$$

since i and d lie in the range $[0..m]$. This proves that the time taken by N is $O(m) = O(t(|x|))$. **Q.E.D.**

This theorem, together with that in section 5, imply that deterministic space is at least as powerful as probabilistic or alternating time separately:

$$PrTIME(t) \cup ATIME(t) \subseteq DSPACE(t)$$

It is not clear if this can be improved to showing that deterministic space is at least as powerful as probabilistic-alternating time. If this proves impossible, then the combination of probabilism and alternation is more powerful than either one separately. This would not be surprising since probabilism and alternation seems to be rather different computing concepts. Our current best bound on simulating probabilistic-alternating time is given next.

THEOREM 23 *For all complexity functions t, $PrA\text{-}TIME(t) \subseteq ATIME(t \log t)$.*

Proof. We use the same notations as the proof of the previous theorem. Let M be a PAM that accepts in time t. Fix any input x and let T_m be the complete computation tree of M on x restricted to levels at most m, and define VAL_m, BIT_m and CAR_m as before. There is one interesting difference: previously, the values m and i in calls to the subroutines to verify $BIT_m(C, i) = b$ and $CAR_m(C, i) = b$ were encoded in unary. We now store these values in binary (the reader is asked to see why we no longer use unary).

Consider the verification of $BIT_m(C, i) = b$ for any inputs C, i, b, using alternating time. If C is a leaf this is easy. Suppose C is a TOSS-configuration. Then we must guess three bits b_1, b_2, c and verify that $BIT_m(C_L, i) = b_1$, $BIT_m(C_R, i) = b_1$ and $CAR_m(C, i-1) = c$. Here we use an idea from the design of logic circuits: in circuits for adding two binary numbers, the carry-bits can be rapidly generated using what is known as the 'carry-look-ahead' computation. In our context, this amounts to the following condition:

$CAR_m(C, i) = 1 \iff$ if there is a j ($j = 0, \ldots, i$) such that $BIT_m(C_L, j) = BIT_m(C_R, j) = 1$ and for all $k = j+1, \ldots, i$, either $BIT_m(C_L, k) = 1$ or $BIT_m(C_L, k) = 1$.

In $O(\log m)$ alternating steps, we can easily reduce these conditions to checking $BIT_m(C', j) = b'$ for some j, b' and C' a child of C. (We leave this detail as exercise.)

Finally, suppose C is a MIN-configuration (a MAX-configuration is handled similarly). By definition $BIT_m(C, i) = BIT_m(C_L, i)$ iff $VAL_m(C_L) < VAL_m(C_R)$, otherwise $BIT_m(C, i) = BIT_m(C_R, i)$. Now $VAL_m(C_L) < VAL_m(C_R)$ iff there exists a j ($0 \le j \le m$) such that

$$\begin{aligned}
BIT_m(C_L, h) &= BIT_m(C_R, h),\ (\text{for } h = j+1, j+2, \ldots, m),\\
BIT_m(C_L, j) &= 0,\\
BIT_m(C_R, j) &= 1.
\end{aligned}$$

Again, in $O(\log m)$ time, we reduce this predicate to checking bits of $VAL_m(C')$, C' a child of C.

To complete the argument, since each call to check a bit of $VAL_m(C)$ is reduced in $O(\log m)$ steps to determining the bits of $VAL_m(C')$ where $level(C') = level(C) + 1$, there are at most m such calls on any computation path. To generate a call to C', we use $O(\log m)$ time, so that the length of each path is $O(m \log m)$. **Q.E.D.**

7.7 Further Generalization of Savitch's Theorem

Savitch's theorem says that for all $s(n) \ge \log n$, $NSPACE(s) \subseteq DSPACE(s^2)$. Chapter 2 gives a generalization of Savitch's theorem. In this section, we further improve this along three directions: (i) by using alternating time, (ii) by allowing small space bounds $s(n)$, i.e., $s(n) < \log n$, and (iii) by extending the class of simulated machines from nondeterministic to alternating machines.

Consider what happens when $s(n) < \log n$. Savitch's proof method gives only the uninteresting result $NSPACE(s) \subseteq DSPACE(\log^2 n)$. Monien and Sudborough [14] improved this so that for $s(n) < \log n$,

$$NSPACE(s) \subseteq DSPACE(s(n) \log n).$$

Using addressable-input alternating machines, Tompa [21] improved the Monien-Sudborough construction to obtain:

$$NSPACE(s) \subseteq ATIME(s(n)[s(n) + \log n])$$

for all $s(n)$. Incorporating both ideas into the generalized Savitch's theorem of chapter 2, we get a new result:

THEOREM 24 *For all complexity functions $t(n) > n$,*

$$N\text{-}TIME\text{-}SPACE(t,s) \subseteq ATIME(s(n) \log \frac{n \cdot t(n)}{s(n)})$$

where the alternating machine here is the addressable-input variety.

Proof. Let M be a nondeterministic machine accepting in time-space (t, s). We describe a addressable-input alternating machine N to accept $L(\text{M})$. Let x be any input, $|x| = n$. M begins by existentially guessing $t = t(n)$ and $s = s(n)$ and marking out s cells on each work tape.

We number the n cells of the input tape containing x as $0, 1, \ldots, n-1$ (rather than the conventional $1, \ldots, n$). We will divide the cells $0, 1, \ldots, n-1$ of the input tape into intervals I_w (subscripted by words $w \in \{L, R\}^*$) defined as follows:

$i \in I_w \iff$ the most significant $|w|$ bits in the binary representation of i corresponds to w

where the correspondence between words in $\{L, R\}^*$ and binary strings is given by $L \leftrightarrow 0$ and $R \leftrightarrow 1$. It is also assumed here that the binary representation of i is expanded to exactly $\lceil \log n \rceil$ bits. Clearly I_w is an interval of consecutive integers. For example: with $n = 6$,

$$I_\epsilon = [0..5],\ I_L = [0..3],\ I_R = [4..5],\ I_{RR} = \emptyset.$$

Observe that

$$I_w = I_{wL} \cup I_{wR} \text{ and } |I_{wL}| \geq |I_{wR}| \geq 0.$$

The *L-end* (resp., *R-end*) *cell* of a non-empty interval is the leftmost cell (resp., rightmost) cell in that interval.

A storage configuration is a configuration in which the contents as well as head position of the input tape are omitted. Let S, S' be storage configurations of M, $d, d' \in \{L, R\}$, $w \in \{L, R\}^*$. Let $conf(S, d, w)$ denote the configuration in which the contents and head positions in the work-tapes are specified by S, with input tape containing the fixed x and the input head scanning the d-end cell of interval I_w. In the course of computation, N will evaluate the two predicates *REACH* and *CROSS* defined next. The predicate

$$REACH(S, S', d, d', w, m)$$

holds if there is a computation path π of length at most m from $conf(S, d, w)$ to $conf(S', d', w)$ where the input head is restricted to the interval I_w throughout the computation, and the space used is at most s. Recall that s is the guessed value of the maximum space usage $s(|x|)$. Let $\overline{L}$ denote R and $\overline{R}$ denote L. Then the predicate

$$CROSS(S, S', d, d', w, m)$$

holds if there is a computation path of length at most m from $conf(S, \overline{d}, wd)$ to $conf(S', \overline{d'}, wd')$ where the input head is restricted to the interval I_w throughout the computation, and the space used is at most s. Observe that the intervals I_{wd} and $I_{wd'}$ used in this definition are adjacent and $I_{wd} \cup I_{wd'} \subseteq I_w$ (if $d = d'$ then this inclusion is proper). We assume in this definition I_{wR} is non-empty; this automatically implies I_{wL} is non-empty. For instance: $CROSS(S, S', L, R, RLR, m)$ holds means there is a path from $conf(S, R, RLRL)$ to $conf(S', L, RLRR)$ of length at most m, as illustrated in the following figure.

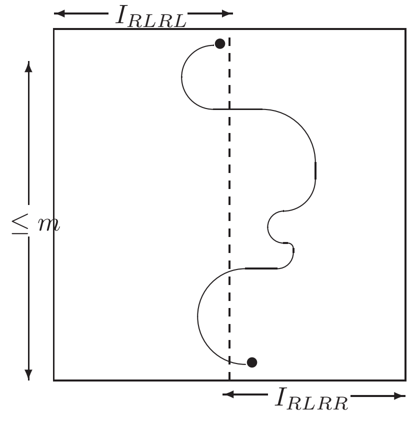


Figure 7.1 The input head positions implied by $CROSS(S, S', L, R, RLR, m)$

We may assume that when M halts, its input head returns to cell 0. The simulation of M amounts to the evaluation of $REACH(S_0, S_f, L, L, \epsilon, t)$ where S_0 is the (trivial) initial storage configuration (independent of x) and S_f is some existentially guessed accepting storage configuration that uses at most s space, λ being the empty string.

We describe the recursive evaluation of $REACH(S, S', d, d', w, m)$. It is assumed that the following setup holds at the time of calling the procedure: S and S' are represented by the contents and head positions on designated tapes of N (if M has k work-tapes, N uses $2k$ tapes for S and S'). The argument w is on the address tape of the N. The value m is written in binary on another tape. One of the following three cases hold:

(i) $|w| = \lceil \log n \rceil$: this condition can be checked in constant time (for instance, by keeping a tally of the length of w) provided that before the simulation begins, the unary representation of $\lceil \log n \rceil$ is guessed and verified once and for all. In this case $|I_w| = 1$ and N enters the read state to read the input symbol $x[w]$ indexed by w. From the definition of *REACH*, from now on N can ignore its input tape and call the predicate

$$REACHABLE(conf(S, d, w), conf(S', d', w), m)$$

where *REACHABLE* is the predicate in the original proof of the generalized Savitch's theorem (chapter 2, section 7). Recall that the predicate $REACHABLE(C, C', m)$ holds if there is a computation path from configuration C to configuration C' of at most m steps using at most s space. The original simulation in chapter 2 uses $O(s \log \frac{m}{s}) = O(s \log \frac{t}{s})$ space, but this is easily modified to give us alternating time $O(s \log \frac{t}{s})$ **Exercise:**. Another slight modification is to make *REACHABLE*reject at once if input head ever moves at any time during the simulation.

(ii) $|I_{wR}| = 0$: then call $REACH(S, S', d, d', wL, m)$. Note that $|I_{wR}| = 0$ iff the binary number corresponding to wR is greater than $n - 1$. This is easily checked, for instance, by entering the READ state and seeing if we next enter the ERROR state.

(iii) $|I_{wR}| \geq 1$ (so $|w| < \lceil \log n \rceil$): N existentially guesses whether there is a computation path π from $conf(S, d, w)$ to $conf(S', d', w)$ with the input head restricted to I_{wd}. If it guesses 'no' (and it will not make this guess unless $d = d'$) then it next calls

$$REACH(S, S', d, d, wd, m)$$

If it guesses 'yes' (it could make this guess even if $d = d'$) then it chooses existentially two storage configurations S'', S''' in the computation path π and then chooses universally to check one of the following:

$$REACH(S, S'', d, \overline{d}, wd, m),$$
$$CROSS(S'', S''', d, d', w, m),$$
$$REACH(S''', S', \overline{d'}, d', wd', m).$$

N existentially guesses one of the cases (i)-(iii), and then universally checks that its guess is correct as well as performs the respective actions described under (i)-(iii). This completes the description of *REACH*.

The subroutine for $CROSS(S, S', d, d', w, m)$ has two cases:

(i)' $m \leq s$: in this case, N can check the truth of the predicate in time s (since a nondeterministic machine is just a special case of alternation).

(ii)' $m > s$: N guesses two storage configurations S'', S''' and a value $d'' \in \{L, R\}$ and universally branches to check

$$CROSS(S, S'', d, d'', w, m/2),$$
$$REACH(S'', S''', \overline{d''}, \overline{d''}, wd'', m) and$$
$$CROSS(S''', S', d'', d', w, m/2).$$

We should explain this choice of S'', S''', d'': if there is a computation path from $conf(S, \overline{d}, wd)$ to $conf(S', \overline{d'}, wd')$ that makes $CROSS(S, S', d, d', w, m)$ true then this path can be broken up into several disjoint portions where the input head is confined to I_{wL} or to I_{wR} in each portion. Consider the portion π of the path that contains the configuration at time $m/2$: let π be confined to the interval $I_{wd''}$ for some d'', and let the storage configurations at the beginning and end of π be S'' and S''', respectively. With this choice of d'', S'', S''', it is clear that the recursion above is correct.

Note that it is unnecessary to check for $m \le s$ in *REACH* (since *REACH* does not reduce m in making recursive calls); likewise it is unnecessary to check for $|I_w| = 1$ in *CROSS* (since it is called by *REACH* only with $|I_w| \ge 2$). Observe that every two successive calls to *REACH* or *CROSS* result in either $|w|$ increasing by at least one or m decreasing to at most $m/2$. Hence in $2(\lceil \log n \rceil + \log(t/s)) = O(\log(nt/s))$ recursive calls, we reduce the input to the 'basis' cases where either $m \le s$ or $|I_w| = 1$. Each recursive call of *REACH* or *CROSS* requires us to guess the intermediate storage configurations S'', S''' in time $O(s)$. Hence in $O(s \log(nt/s)])$ steps we reach the basis cases. In these basis cases, the time used is either $O(s)$ time or that to compute $REACHABLE(C, C', m)$. The latter is $O(s \log(t/s))$ as noted before. The total time is the sum of the time to reache the basis cases plus the time for basis cases. This is $O(s \log(nt/s)])$. **Q.E.D.**

The structure of the preceding proof involves dividing at least one of two quantities in half until the basis case. An immediate consequence of the above results is this:

Corollary 25
(i) $NSPACE(s) \subseteq ATIME(s^2)$.
(ii) $PrTIME(n^{O(1)}) \subseteq ATIME(n^{O(1)}) = PrA\text{-}TIME(n^{O(1)}) = PSPACE$.

Borodin [3] observed that Savitch's theorem is capable of generalization in another direction. Incorporating Borodin's idea to the previous theorem yields the following "super" Savitch's theorem. Recall the definition of alternating complexity in section 3.

Theorem 26 *Let $t(n) > n, s(n)$ and $a(n)$ be any complexity functions. Then*

$$A\text{-}TIME\text{-}SPACE\text{-}ALTERNATION(t, s, a) \subseteq ATIME(s(n)[a(n) + \log \frac{n \cdot t(n)}{s(n)}])$$

where alternating machines are the addressable-input variety.

Proof. Suppose an alternating machine M accepts in time, space and alternating complexity of $t(n), s(n)$ and $a(n)$. On input x, the machine N begins by guessing the values $t_0 = t(|x|)$ and $s_0 = s(|x|)$. Let $T(x) = T_{t_0,s_0}(x)$ be the computation tree on x restricted to nodes at level $\le t_0$ and using space $\le s_0$. There are two procedures involved: The main procedure evaluates a predicate $ACCEPT(C)$ that (for any configuration C as argument) evaluates to true if $C \in T(x)$ and the least fixed point value $Val_T(C)$ of C is equal to 1. The other procedure we need is a variation of the predicate $REACH(S, S', d, d', w, m)$ in the proof of theorem 24. Define the new predicate

$$REACH'(S, S', v, v', w, m)$$

where S, S' are storage configurations, $v, v', w \in \{L, R\}^*$ and $m \ge 1$ such that $|wv| = |wv'| = \lceil \log n \rceil$. Let $conf(S, w)$ where $|w| = \lceil \log n \rceil$ denote the configuration whose storage configuration is S and input tape contains x and the input head is at position indicated by w. The predicate $REACH'$ evaluates to true provided there is a computation path π of length $\le m$ from $conf(S, wv)$ to $conf(S', wv')$ such that all the intermediate configurations C use space $\le s$ and has input head restricted to the interval I_w. We further require that

(a) C and $conf(S, wv)$ have opposite types, i.e., C is a MIN-configuration if and only if $conf(S, wv)$ is a MAX-configuration. Note that we assume M has no NOT-configurations.

(b) The computation path π we seek must have only configurations of the same type as C with the sole exception of its last configuration (which is equal to $conf(S, wv)$, naturally).

It is clear that we can compute $REACH'$ in alternating time $O(s \log t/s)$ as in the case of *REACH*.

The procedure $ACCEPT(C)$ proceeds as follows: suppose C is an MAX-configuration (resp., MIN-configuration). Then the algorithm existentially (resp., universally) chooses in time $O(s)$ a configuration C' with opposite type than C. Let $C = conf(S, v)$ and $C' = conf(S', v')$. Regardless of the type of C, the algorithm existentially chooses to call the following subroutines:

(1) $ACCEPT(C')$

(2) $\neg REACH'(S, S', v, v', \epsilon, t_0)$ where the values S, S', v, v' are related to C, C' as above. Of course, by $\neg REACH'$ we mean that the procedure first enters a NOT-state and then calls $REACH'$. (Here is an occasion where it is convenient to re-introduce NOT-states.) The reader should easily see that the procedure is correct.

We now analyze the complexity of the procedure *ACCEPT*. For any configuration C let T_C be the subtree of configurations reachable from C. Define $depth(C)$ to be the minimum k such that there is prefix T' of T_C such that T' is accepting and each path in T' has at most k alternation. In particular, observe that if x is accepted by M then $depth(C_0(x))$ is at most $a(|x|)$, with $C_0(x)$ the initial configuration. Let $W(k)$ be the (alternating) time required by the procedure for $ACCEPT(C)$ on input C with depth k. Then we have

$$W(k) = O(s) + \max\{s \log \frac{t}{s}, W(k-1)\}.$$

To see this, suppose C is an MAX- (resp., MIN-) configuration. Then $O(s)$ is the time to existentially (resp., universally) choose the configurations C' reachable from C; $s \log t/s$ is the time to decide the predicate $REACH'$; and $W(k-1)$ is the time to recursively call $ACCEPT(C')$. It is easy to deduce that $W(k)$ has solution given by:

$$W(k) = O(s \cdot [k + \log t/s]).$$

The theorem follows immediately. **Q.E.D.**

This is still not the last word on extensions of Savitch's theorem! We return to this in the next chapter.

7.8 Alternating Time versus Deterministic Time

The main result of this section is the following theorem:

Theorem 27 *For all t, $DTIME(t) \subseteq ATIME(\frac{t}{\log t})$.*

Tompa and Dymond [6] obtained this result by adapting the result of Hopcroft, Paul and Valiant [12] showing $DTIME(t) \subseteq DSPACE(t/\log t)$. Adleman and Loui [1] gave an interesting alternative proof of the Hopcroft-Paul-Valiant result. The Hopcroft, Paul and Valiant achievement showed for the first time that space is a more powerful resource than time in a general model of computation (namely, multitape Turing machines). For restricted models of computation, Paterson [17] already established that space is more powerful than time for simple Turing machines. Theorem 27 is also an improvement of the result of Paul and Reischuk who simulated deterministic time t using alternating time $O(t \log\log t/\log t)$. In the following, we shall assume the addressable-input model of alternation in case $t/\log t = o(n)$ in the theorem, but otherwise, the regular model suffices.

As an interesting corollary, in conjunction with the alternating time hierarchy theorem at the end of section 4, is that there are languages in *DLBA* that cannot be accepted in deterministic linear time.

7.8.1 Reduction of Simulation to a Game on Graphs.

First consider the simpler problem of simulating a deterministic Turing machine using as little deterministic space as possible. A key step is the reduction of this problem to a combinatorial question on graphs. Suppose a deterministic k-tape machine M accepts an input in $t > 0$ steps. Our goal is to describe a deterministic machine N that simulates M using as little space as possible.

Let $B = B(t) > 0$ be the *blocking factor*, left unspecified for now. For $i = 0, 1, \ldots, \lceil t/B \rceil$, let

$$t_i := iB$$

be *time samples*, and let the cells of each work-tape be grouped into *blocks* consisting of B consecutive cells. For each block b, let *neighborhood*(b) denote the set of 3 blocks consisting of b and the two adjacent blocks on either side of b. We construct a directed acyclic graph $G = (V, E)$ with node set

$$V = \{0, 1, \ldots, \lceil t/B \rceil\}$$

and *labels* for each node. The label for a node i consists of the following two pieces of information:

(i) positions $h_0, \ldots, h_k$ of the $k+1$ tape heads and

(ii) a state q.

We say that this label of node i is *correct* if at time sample t_i, the machine is in state q and the heads are at positions given by $h_0, \ldots, h_k$. We may say that block b is *visited* in time sample t_j if the label of node j says that there is a tape head somewhere in b. Note that this definition is relative to the labeling, regardless of its correctness. Once the labels are given, we can define an edge set E as follows. The edges in E are those (i, j) satisfying one of two requirements:

(a) There is a tape block b visited at time sample t_j, another tape block b' visited time sample t_i such that *neighborhood*$(b) \cap$ *neighborhood*(b') is non-empty and, previous to sample time t_j, b' is last visited in time sample t_i.
(b) There is a tape block b visited in time sample t_j such that *neighborhood*(b) contains a block that has never been visited in time samples before t_j, and $i = 0$.

If $(i, j) \in E$ then necessarily $i < j$, and if the labeling is correct then $(i, i+1)$ must be in E. Let *neighborhood*(j) denote the union of the blocks in *neighborhood*(b) where b range over all blocks visited in time sample t_j. Clearly *neighborhood*(j) has exactly $3k$ blocks. Let b be visited in time sample t_j. Then there ≤ 5 blocks b' such that *neighborhood*$(b') \cap$ *neighborhood*(b) is non-empty. Each such b' contributes an edge of the form $(i, j) \in E$ for some i. This implies that the indegree of each node in G is $\le 5k$. The outdegree of G (with the exception of node 0) is similarly bounded by $5k$.

A description of G together with its labels can be written down using at most

$$\frac{t \log t}{B}$$

space. This space is less than t if B is larger than $\log t$. We attempt to find such a graph G by testing successively larger values of t, and for each t, cycling through all ways of assigning labels. It remains to show how to verify a proposed labelling. The idea is that each node in G can be 'expanded' in the following sense: the *expansion* of a node $i \in G$ consists of the contents of the blocks in *neighborhood*(i) in time sample t_i. Note that the expansion of i can be encoded using $O(B)$ space. The edges of G define a predecessor-successor relationship: (i, j) is an edge mean that i is a *predecessor* of j, and j the *successor* of i. Next we make an important but elementary observation:

(*) If we already have the expansions of all the predecessors of node $i \ge 1$ then we may expand node i simply by simulating the machine starting from the moment $t_{i-1} = (i-1)B$.

To do this, we first reconstruct the contents of blocks in *neighborhood*$(i-1) \cup$ *neighborhood*(i), using the expansions of the predecessors of i. (There could be overlap among the predecessor expansions, but it is easy to only use the contents of the most recent version of a block.) Now simulate M starting from time sample t_{i-1} to time sample t_i. At the end of the simulation, we may assume that the expansion of node i is now available, in addition to the previous expansions. Details can be filled in by the reader. Let us say that a node i is *verified* if we confirm that its label (i.e., head positions and state) is correct.

(**) If the predecessors of node i are expanded and verified then we can also expand and verify node i.

This is because we can compare the state and head positions in the expansion of i with the labels of node i.

Now we can give a nondeterministic procedure to verify G: nondeterministically expand nodes, one at a time. At any moment, the tapes of the simulator contain some number of expanded nodes. Those nodes whose only predecessor is node 0 can be expanded at any moment; for any other node i, we can only expanded i if all its predecessors are expanded. At the end of expanding node i, we verify the label of i. We may nondeterministically *contract* any previous expansion if we wish; contraction is just the inverse of expansion. Of course we may contract a node only to re-expand it later. The space used by this procedure is $O(B)$ times the maximum number of expanded nodes at any moment. So to minimize space usage, we should contract nodes "at suitable moments". The graph G is said to be verified if its final node $\lceil t/B \rceil$ is verified in this process; we might as well assume that the label of $\lceil t/B \rceil$ always contains the accept state.

It is not hard to see that M accepts its input x iff there is a graph G that is verified by this procedure. We can make this procedure deterministic by cycling through all nondeterministic choices used in the expansion/contraction above. For a space-efficient method of verifying G, Hopcroft, Paul and Valiant showed a general strategy that never store more than $\frac{t}{B \log t}$ expanded nodes at any moment during the verification process. This means that the strategy never use more than $\frac{t}{\log t}$ space since each expanded node uses $O(B)$ space. This proves that $DTIME(t) \subseteq DSPACE(t/\log t)$. The details of this will not be explicitly described since it is essentially subsumed in the Tompa-Dymond alternating time implementation of the strategy, shown next.

7.8.2 A Pebble Game.

Now we transcribe the previous expansion and contraction process for verifying G into a combinatorial game on graphs. We are given a directed acyclic graph $G = (V, E)$ together with a *goal* node $i_0 \in V$. There is only one player in this game. There is an infinite supply of indistinguishable pebbles and each node of G can hold a single pebble. A node is said to be *pebbled* if there is a pebble in it; it is *empty* otherwise. Initially, all the nodes are empty. A *pebbling step* consists of placing a pebble on an empty node u, provided all predecessors of u are already pebbled. In particular, we can always pebble an empty *source node* (i.e., a node with no predecessors). An *unpebbling step* consists of removing a pebble from a pebbled node. A *play* is simply a sequence of pebbling or unpebbling steps, with the last step being the pebbling of the goal node i_0. At any moment during the play, there is some number of pebbles on the graph, and our aim (as the player) is to choose the steps in a play in order to minimize the maximum number k of pebbled nodes at any time during the play. This number k is called *the pebble cost* of the play.

The reader will have no difficulty making the connection between this pebble game and the simulation described earlier: pebbling (unpebbling) a node corresponds to expansion (contraction) of nodes.

A *game strategy* is a rule to play the game for any graph. A trivial game strategy is to pebble the nodes in topological order and never to unpebble any nodes. On an n node graph, the pebble cost is n with this strategy. Can we do better in general? The key result here says: *for any directed acyclic graph G on n nodes with in- and out-degree at most d, the strategy yields a play with pebble cost $O_d(n/\log n)$.*

We want an 'alternating version' of playing this pebbling game. As usual, alternation turns the problem inside-out (or rather, bottom-up): instead of proceeding from the source nodes to the goal node i_0, we first ask what is required to pebble the goal node. This is viewed as a "challenge" at node i_0. The challenge at a node u in turn spawns challenges at other nodes (which must include all predecessors of u). This is roughly the idea for our key definition:

Definition 12 A *pebbling tree* for a directed acyclic graph G with goal node i_0 is a finite tree T satisfying the following.[17] Each vertex u of T is associated with a triple $[i, X, Y]$ where X and Y are subsets of nodes of G, and i is a node of G. We called i the *challenged node*, X the *pebbling set*, Y the *unpebbling set* (at vertex u). At each vertex u, define the **current set** $C(u)$ of (currently) pebbled nodes at u by induction on the level of u: if u is the root then $C(u)$ is simply the pebbling set at u; otherwise if u is a child of u' then $C(u) = (C(u') - Y) \cup X$ where X (resp., Y) is the pebbling (resp., unpebbling) set at u. We require these properties:

(i) The challenged node at the root is the goal node i_0.

(ii) At each vertex u associated with $[i, X, Y]$, either $i \in X$ or else all the predecessors of i are contained in the current set $C(u)$.

(iii) If the pebbling set at vertex u is X, then u has $|X|$ children, and the set comprising the challenged nodes at these children is precisely X.

■

Remarks Note that (iii) implies that the pebbling set at a leaf must be empty; (ii) further implies that the predecessors of a challenged node at a leaf u is in $C(u)$. The concept of "pebble" in pebbling trees is distinct from the pebbles in the original pebbling game: in the literature, this distinction is made by giving colors (black and white) to the different concepts of pebbles. We may call the "pebbles" in pebbling trees "challenge pebbles", because they are targets to be achieved in the original pebbling game.

Interpretation: This tree is an abstract description of an alternating computation tree that verifies the labels of a graph G in the sense of the Hopcroft-Paul-Valiant simulation of a deterministic time t machine M. To make this precise, we first describe an alternating machine N that on input a labeled graph G with goal node i_0 behaves as follows: initially, N existentially guesses some expansion of node i_0 and writes this onto tape 1; tape 2 is empty. In general, N is in the following 'inductive stage':

Tape 1 contains the expansion $e(i')$ some node i',
Tape 2 holds some number of expansions of nodes in G.

Then N existentially deletes some expansions in tape 2 and existentially writes some (possibly zero) new expansions in tape 3. If no expansion of node i' is found in tapes 2 and 3 then N tries to produce one: first N checks that all the predecessors of i' are in tapes 2 and 3 (otherwise it rejects) and then simulate M from sample time $t_{i'-1}$ to sample time $t_{i'}$ and, as usual, assume that we now have an expansion $d(i')$ of i'. N can now verify if the expansion $e(i')$ agrees with the found or newly constructed $d(i')$ (if not, N rejects). To continue, either tape 3 is empty (in which case N accepts) or else N universally chooses to copy one of the expanded nodes from tape 3 to tape 1 and the rest onto tape 2. This completes the 'inductive stage'.

[17]To avoid confusing the nodes of T with those of G, we will refer to the nodes of T as 'vertices'.

We claim that N accepts iff there is a pebbling tree T. Suppose T exists. To show that N accepts, we describe an accepting computation tree T' of N that is modeled after T: each inductive stage of N corresponds to a vertex u of T. If u is associated with the triple $[i, X, Y]$ then tape 1 contains the expansion of node i and tape 2 contains the expansions of the set $C(u)$ of pebbled nodes at u. Then the sets Y, X corresponds (respectively) to the expansions that are deleted from tape 2 or existentially guessed in tape 3. If we choose T' in such a way that the guessed expansions in tape 3 are always correct, then T' would be an accepting computation tree. Conversely, if N accepts, then we can construct the pebbling tree by reversing the above arguments. ■

With this interpretation, it is easy to understand the following definition of complexity. The *pebbling time* at any vertex with label $[i, X, Y]$ is given by $1 + |X| + |Y|$. The pebbling time of a path of T is the sum of the pebbling times of nodes along the path. The *pebbling time* of T is the maximum pebbling time over all paths in T. The *pebbling space* of T is the maximum of $|C(u)|$ over all vertices u. These corresponds to alternating time and alternative space, respectively. Since we do not care about minimizing alternating space in the following proof, we may assume each unpebbling set Y is empty. We leave the following as an exercise:

LEMMA 28 *Let G be any directed acyclic graph and i_0 be a node in G. If there is a pebbling tree for (G, i_0) with pebbling time t, then we can pebble (G, i_0) using t pebbles.*

We come to the main lemma:

LEMMA 29 *Let G be any directed acyclic graph with m edges, and whose indegree and outdegree is at most d. For any goal node i_0 in G, there exists a pebbling tree for (G, i_0) with pebbling time of $O_d(m/\log m)$*

Proof. Let G be any graph described by the lemma and i_0 is any node in G, We describe a pebbling tree for (G, i_0) whose pebbling time is at most $P(m)$, where $P(m) = O_d(m/\log m)$. We may suppose m is sufficiently large. First partition the nodes V of G into two disjoint sets, $V = V_a \cup V_b$ such that

(i) There are no edges from V_b ('nodes below') to V_a ('nodes above'). So edges of G that cross between V_a and V_b must descend from above to below. Let G_a, G_b be the induced subgraphs with nodes V_a, V_b, respectively.

(ii) The number of edges m_b in V_b satisfies

$$\frac{m}{2} - \frac{m}{\log m} \leq m_b < \frac{m}{2} - \frac{m}{\log m} + d.$$

To see that such a partition exists, we offer a construction. Starting with V_b as the empty set, keep adding nodes into V_b in topological order until the number of edges of G_b satisfies the above inequalities (this is possible because additional node in V_b increases the number of edges by at most d).

Let $B \subseteq V_b$ comprise those nodes with at least one predecessor in V_a. Consider three cases.

CASE 1 Suppose the goal node i_0 is in V_a. Then a pebbling tree for (G_a, i_0) is also a pebbling tree for (G, i_0). This tree has a pebbling time at most

$$P(m - m_b) \leq P(\frac{m}{2} + \frac{m}{\log m}).$$

Assume $i_0 \in V_b$ in the remaining cases.

CASE 2 Suppose $|B| < 2m/\log m$. Then we construct the following pebbling tree T for (G, i_0). The pebbling set at the root of T is $B \cup \{i_0\}$. At each child u of the root, we consider two possibilities. If the challenged node at u is $i \in B$, then inductively construct a pebbling tree for (G_a, i) with pebbling time $P(m/2 + m/\log m)$. Otherwise the challenged node is i_0 and we inductively construct a pebbling tree for (G_b, i_0) with pebbling time $P(m/2 - m/\log m + d)$. The result is a pebbling tree for (G, i_0) with pebbling time at most

$$\frac{2m}{\log m} + P(\frac{m}{2} + \frac{m}{\log m}).$$

CASE 3 Suppose $|B| \geq 2m/\log m$. Consider a pebbling tree T_b for (G_b, i_0) with pebbling time $\leq P(m/2 - m/\log m + d)$. We convert T_b into a pebbling tree for (G, i_0): let u be any leaf of T_b with challenged node i. The predecessors of i in G_b are contained in $C(u)$, by definition of T_b. But i may bave predecessors in G but not in G_b: let $X(i)$ be this set of predecessors. We make $X(i)$ the pebbling set at u and create $|X(i)| \leq d$ children for u. Each child v of u has a challenged node $j \in V_a$. We can attach to v a pebbling tree T_j for

(G_a, j). Notice G_a has at most $m - m_b - |B| \leq (m/2) - (m/\log m)$ edges since are at least $|B| \geq 2m/\log m$ edges from V_a to V_b are not counted in G_a or G_b. Hence the pebbling time for T_j is at most $P(m/2 - m/\log m)$. This completes our description of the pebbling tree for (G, i_0). The pebbling time of this tree is equal to the pebbling time of T_b plus the pebbling time of any T_j's plus at most d. This is at most

$$2P(\frac{m}{2} - \frac{m}{\log m}) + d.$$

Taking the worst of these three cases, we obtain

$$P(m) \leq \max\{P(\frac{m}{2} + \frac{m}{\log m}) + \frac{2m}{\log m}, 2P(\frac{m}{2} - \frac{m}{\log m}) + d\}$$

We want to show that there is a constant $c = c(d) \geq 5$ such that for m', $P(m') \leq cm'/\log m'$. By making c sufficiently large, we may assume that the truth has been established for m large enough. Inductively, we have the the following derivation:

$$\begin{aligned}
P(\frac{m}{2} + \frac{m}{\log m}) + \frac{2m}{\log m} &= P(\alpha m) + \frac{2m}{\log m} \quad (\text{where } \alpha = \tfrac{1}{2} + \tfrac{1}{\log m}) \\
&\leq \frac{c\alpha m}{\log(\alpha m)} + \frac{2m}{\log m} \\
&\leq \frac{cm}{\log m}\left(\frac{\alpha \log m}{\log(\alpha m)} + \frac{2}{c}\right) \\
&\leq \frac{cm}{\log m}.
\end{aligned}$$

We also have

$$\begin{aligned}
2P(\tfrac{m}{2} - \tfrac{m}{\log m}) + d &\leq \frac{2c\left(\frac{m}{2} - \frac{m}{\log m}\right)}{\log\left(\frac{m}{2} - \frac{m}{\log m}\right)} + d \\
&\leq \frac{cm\left(1 - \frac{2}{\log m}\right)}{\log m + \log\left(\frac{1}{2} - \frac{1}{\log m}\right)} + d \\
&\leq \frac{cm\left(1 - \frac{2}{\log m}\right)}{\log m - 1.1} + d \\
&\leq \frac{cm}{\log m}\left(\frac{\log m - 2}{\log m - 1.1}\right) + d \\
&\leq \frac{cm}{\log m}.
\end{aligned}$$

Q.E.D.

We may now complete the proof of the main result showing a deterministic M that accepts in time t can be simulated in alternating time $O(t/\log t)$. In applying the above lemma, we may recall that the graph G obtained from a computation of M by using some blocking factor B has bounded in- and out-degrees except for node 0. To fix this, we can simply place a pebble at node 0 and the rest of the graph is now effectively bounded degree.

(1) Reserve tapes 1, 2 and 3 for later use. First we existentially choose the time t (tape 4) and blocking factor B (tape 5). Then we existentially choose a labeling of the graph G with nodes $V = \{0, \ldots, t/B\}$ (tape 6), and an edge set E (tape 7). Since each label uses $O(\log t)$ space, all this (when choice is correct) takes time $O(\frac{t \log t}{B})$.

(2) Universally choose to verify that E is correct relative to node labels, and to verify that the label of node t/B is correct. It takes time $O(\frac{t \log t}{B})$ to verify E. Verification of the label at node t/B is recursive as shown next.

(3) The verification of node $\lceil t/B \rceil$ amounts to simulating a pebbling tree T for $(G, \lceil t/B \rceil)$ (i.e., with $\lceil t/B \rceil$ as the goal node of G). We do this along the lines given by the "Interpretation" above. As before, each 'inductive stage' of our simulation of T corresponds to a vertex u of T: if $[i, X, Y]$ is associated with u then an expansion of the challenged node i is available on tape 1. The set of nodes previously expanded are available on tape 2. Since the pebbling time of T can be assumed to be $t/(B \log(t/B))$, we may assume that tape 2 has at most $t/(B \log(t/B))$ nodes. Since each expansion uses $O(B)$ space, tape 2 uses $O(t/\log(t/B))$ space. We existentially choose the pebbling set X at u and also their expansions, writing down these guesses on tape 3. (As noted before, we may assume Y is empty.) We then verify the challenged node i (it must either appear in tapes 2 or 3 or has all its predecessors expanded so that it can be simulated directly). This non-recursive verification of node i takes time $O(t/\log(t/B))$. To get to the next inductive stage, we universally choose to transfer one of the expanded nodes on tape 3 to tape 2, which is now the challenged node.

We have seen that the non-recursive parts of step (3) takes alternating time $O(t/\log(t/B))$. This time must be added to the total alternating time in the recursive parts of the computation. The recursive part of the computation, we claim is $O(B)$ times the pebbling time P. This is because each unit of pebbling time P can be associated with the guessing of an expanded node. But each expansion, when correct, takes space $O(B)$. It follows that the recursive part of the computation takes time $O(t/\log(t/B))$ since the pebbling time for the optimal tree is at most $O(t/[B \log(t/B)])$ (by preceding lemma, with $n = t/B$). Finally, if B is chosen to be $\log^2 t$, we get a time bound of $O(t/\log t)$ for steps (1),(2) and (3). (We could choose B as large as t^ϵ for any constant $\epsilon > 0$.) This concludes the proof of our main theorem.

Finally, we note that the space bound just obtained is the best possible in the sense that $\Omega(t/\log t)$ is a lower bound on the worst case pebbling time for the class bounded in-degree graphs [19].

EXERCISE

Exercise 7.8.1: In graph G, we say node v is an ancestor of node w if there is a path from v to w (so "ancestor" is the reflexive transitive closure of the "predecessor" relation).
(i) Show that in any pebbling tree T for (G, i_0), it is possible to restrict the pebbling sets at each vertex to the ancestors of i_0.
(ii) Prove lemma 28. **Hint:** Suppose $u_1, \ldots, u_k$ are the children of the root of T and x_i is the challenged node at u_i. Let T_j be the subtree of T rooted at u_j. Renumber the indices so that if x_i is an ancestor of x_j then $i < j$. You strategy need to take into account this topologically sorted sequence on $x_1, \ldots, x_k$. Is it true that T_j is a pebbling tree of (G, x_j)? □

END EXERCISE

7.9 Alternating Space

We show two results from Chandra, Kozen and Stockmeyer that relate alternating space and deterministic time.

Note that for a nondeterministic machine M, if there is an accepting path then there is one in which no configuration is repeated. The next lemma shows an analogous result for alternating machines.

LEMMA 30
(a) Let M be any choice machine. If T is an accepting computation tree for an input w then there is an accepting computation tree T' for w with the following properties:

- *each computation path in T' is a (prefix of a) computation path in T*
- *if $u, v \in T'$ are vertices such that u is a proper ancestor of v and both u and v are labeled by the same configuration, then $Val_{T'}(v) \sqsubset Val_{T'}(u)$ (strict ordering).*

(b) If, in addition, M is an alternating machine then we can assume that v is a leaf of T'.

Proof. (a) The result follows if we show another accepting computation tree T' with fewer vertices. Let T_v denote the subtree of T rooted at v consisting of all descendents of v. There are two cases: if $Val_T(u) \sqsubseteq Val_T(v)$ then we can form T' from T by replacing T_u with T_v. By monotonicity, T' is still accepting.
(b) This simply follows from part (a) since for alternating machines, $Val_T(v) \sqsubset Val_T(u)$ implies that $Val_T(v) = \bot$. In that case, we might as well prune away all proper descendents of v from T.

Q.E.D.

THEOREM 31 *For all complexity functions s,*

$$ASPACE(s) \subseteq DTIME(n^2 \log n O(1)^s).$$

Proof. Let M be an ordinary alternating machine accepting in space s. Later we indicate how to modify the proof if M has the addressable-input capability. We will construct a deterministic N that accepts $L(M)$ in the indicated time. On an arbitrary input w, N proceeds in stages: in the mth stage, N enumerates (in tape 1) the set Δ_m defined to be all the configurations C of M on input w where C uses at most m space. Note that each configuration can be stored in $m + \log n$ space, and there are $nO(1)^m$ configurations, so we use

$$(m + \log n)nO(1)^m = n \log n O(1)^m$$

space on tape 1. Then N enumerates (in tape 2) the edges of the computation tree T_m whose nodes have labels from Δ_m and where no node $u \in T_m$ repeats a configuration that lie on the path from the root to u, except when u is a leaf. Clearly this latter property comes from the previous lemma. Using the information in tape 1, it is not hard to do this enumeration of edges in a 'top-down' fashion (we leave the details to the reader). Furthermore each edge can be produced in some constant number of scans of tape 1, using time $n \log n O(1)^m$. Thus the overall time to produce all $nO(1)^m$ edges is $n^2 \log n O(1)^m$. Now we can compute the least fixed point $Val_{T_m}(u)$ value at each node $u \in T_m$ in a bottom-up fashion, again $O(n \log n O(1)^m)$ per node for a total time of $n^2 \log n O(1)^m$. This completes our description of the mth stage.

The previous lemma shows that if M accepts w then at some mth stage, $m \leq s(|x|)$, T_m is accepting. Since the time for the mth stage is $n^2 \log n O_1(1)^m$, the overall time over all stages is

$$\sum_{m=1}^{s(n)} n^2 \log n O_1(1)^m = n^2 \log n O_2(1)^{s(n)}.$$

It remains to consider the case where M has the addressable-input capability. We first note that we never have to use more than $O(\log n)$ space to model the address tape (if M writes more than $\log n$ bits, we ignore the tape from that point onwards since it will lead to error when a READ is attempted). Hence the above space bounds for storing a configuration holds. Furthermore, the time to generate the contents of tapes 1 and 2 remains asymptotically unchanged. Similarly for computing the least fixed point Val_{T_m}. This concludes the proof. **Q.E.D.**

THEOREM 32 *For all $t(n) > n$, $DTIME(t) \subseteq ASPACE(\log t)$.*

Proof. Let M accept in deterministic time t. We describe an alternating machine N to accept $L(M)$ in space $\log t$. For this simulation, N can be the ordinary variety of alternating machine. We may assume that M is a simple Turing machine and M never moves its tape head to the left of its initial position throughout the computation. (Otherwise, we first convert M into a simple Turing machine accepting in time $t(n)^2$ with these properties. How?) Let x be any word accepted by M. Let $C_0, C_1, \ldots, C_m$, $m = t(|x|)$, be the unique accepting computation path of M on x. We assume that the final accepting configuration is repeated as many times as needed in this path. Let each C_i be encoded as a string of length $m + 2$ over the alphabet

$$\Gamma = \Sigma \cup [Q \times \Sigma] \cup \{\sqcup\}$$

where Σ is the tape alphabet of M, Q the state set of M, and $[Q \times \Sigma]$ is the usual composite alphabet. Furthermore, we may assume the the first and last symbol of the string is the blank symbol $\sqcup$. Let $\alpha_{i,j}$ denote the jth symbol in configuration C_i ($i = 0, \ldots, m; j = 1, \ldots, m + 2$).

N will be calling a subroutine $CHECK(i, j, b)$ that verifies whether $\alpha_{i,j} = b$ where $b \in \Gamma$. N begins its overall computation by existentially choosing the integer m, the head position h ($1 \leq h \leq m + 2$) and a symbol $b' = [q_a, c]$ and then it calls $CHECK(m, h, b')$. The subroutine is thus verifying that that M is scanning the symbol c at position h when M enters the accept state q_a. All integer arguments are in binary notation.

In general, the subroutine to verify if $\alpha_{i,j} = b$ (for any i, j, b) operates as follows: if $i = 0$ or $j = 1$ or $j = m$, N can directly do the checking and accept or reject accordingly. Otherwise, it existentially chooses the symbols b_{-1}, b_0, b_{+1} such that whenever b_{-1}, b_0, b_{+1} are consecutive symbols in some configuration of M then in the next instant, b_0 becomes b. Now N universally chooses to call

$$CHECK(i - 1, j - 1, b_{-1}), CHECK(i - 1, j, b_0), CHECK(i - 1, j + 1, b_{+1}).$$

It is important to realize that even if b does not contain the tape head (i.e., $b \notin [Q \times \Sigma \times I]$), it is possible for b_{-1} or b_{+1} to contain the tape head. The space used by N is $O(\log m)$.

Correctness. If M accepts then it is clear that N accepts. The converse is not obvious. To illustrate the subtlety, suppose $\mathit{CHECK}(i, j, b)$ accepts because $\mathit{CHECK}(i-1, j-\epsilon, b_\epsilon)$ accepts for $\epsilon = -1, 0, 1$. In turn, $\mathit{CHECK}(i-1, j-1, b_{-1})$ accepts because $\mathit{CHECK}(i-2, j-1, b')$ (among other calls) accepts for some b'; similarly $\mathit{CHECK}(i-1, j, b_0)$ accepts because $\mathit{CHECK}(i-2, j-1, b'')$ (among other calls) accepts for some b''. But there is no guarantee that $b' = b''$ since these two calls occur on different branches of the computation tree. Another source of inconsistency is that the existential guessing of the b_j's may cause more than one tape head to appear during one configuration. Nevertheless, we have a correctness proof that goes as follows. First observe that if $\mathit{CHECK}(i, j, b)$ is correct if $i = 0$. Moreover, given j there is a unique b that makes $\mathit{CHECK}(0, j, b)$ accept. Inductively, assume $\mathit{CHECK}(i, j, b)$ is correct for all j, b and that $\mathit{CHECK}(i, j, b)$ and $\mathit{CHECK}(i, j, b')$ accept implies $b = b'$. Then it is easy to see that $\mathit{CHECK}(i+1, j, b)$ must be correct for all j, b; moreover, because of determinism, $\mathit{CHECK}(i+1, j, b)$ and $\mathit{CHECK}(i+1, j, b')$ accept implies $b = b'$. [This is why *CHECK* must universally call itself three times: for instance, if *CHECK* only makes two of the three calls, then the correctness of these two calls does not imply the correctness of the parent.] So we have shown that the symbols $\alpha_{i,j}$ are uniquely determined. In particular $\alpha_{m,h} = b'$ in the initial guess is correct when the machine accepts. **Q.E.D.**

The reader should see that this proof breaks down if we attempt to simulate nondeterministic machines instead of deterministic ones.

Combining the two theorems yields the somewhat surprising result:

Corollary 33 *For $s(n) \ge \log n$, $\mathit{ASPACE}(s) = \mathit{DTIME}(O(1)^s)$.*

7.10 Final Remarks

This chapter introduced choice machines to provide a uniform framework for most of the choice modes of computation. It is clear that we can extend the basic framework to other value sets S (analogous to the role of *INT*) provided S is equipped with a partial order $\sqsubseteq$ such that limits of $\sqsubseteq$-monotonic chains are in S and S has a $\sqsubseteq$-least element (such an S is called a *complete partial order*). The reader familiar with Scott's theory of semantics will see many similarities. For relevant material, see [22].

We have a precise relationship between alternating space and determistic time: for $s(n) \ge \log n$,

$$\mathit{ASPACE}(s) = \mathit{DTIME}(O(1)^s). \tag{8}$$

What is the precise relationship between alternating time and deterministic space? Although we have tried to emphasize that alternating time and deterministic space are intimately related, they are not identical. We know that

$$\mathit{ATIME}(s) \subseteq \mathit{DSPACE}(s) \subseteq \mathit{NSPACE}(s) \subseteq \mathit{ATIME}(s^2). \tag{9}$$

for $s(n) \ge \log n$. How 'tight' is this sequence? It is unlikely that that the first two inclusions could be improved in the near future.

From (8) and (9), we get find new characterizations of some classes in the canonical list:

$$\begin{aligned} P &= \mathit{ASPACE}(\log n) \\ \mathit{PSPACE} &= \mathit{ATIME}(n^{O(1)}) \\ \mathit{DEXPT} &= \mathit{ASPACE}(n) \\ \mathit{EXPS} &= \mathit{ATIME}(O(1)^n). \end{aligned}$$

What is the relationship of alternating reversal with deterministic complexity? Of course, for alternating machines, we must take care to simultaneously bound reversal with either time or space in order to get meaningful results. Other complexity measures for alternating machines have been studied. Ruzzo [20] studied the *size* (i.e., the number of nodes) of computation trees. In particular, he shows

$$\mathit{A\text{-}SPACE\text{-}SIZE}(s(n), z(n)) \subseteq \mathit{ATIME}(s(n)\log z(n)).$$

King [13] introduced other measures on computation trees: *branching* (i.e., the number of leaves), *width* (below), *visit* (the maximum number of nodes at any level). Width is not so easy to motivate but in the special case of binary trees (which is all we need for alternating machines), it is the minimum number of pebbles necessary to pebble the root of the tree. Among the results, he shows (for $s(n) \ge \log n$),

$$\begin{aligned} \mathit{A\text{-}SPACE\text{-}WIDTH}(s(n), w(n)) &= \mathit{NSPACE}(s(n)w(n)), \\ \mathit{A\text{-}SPACE\text{-}VISIT}(s(n), v(n)) &\subseteq \mathit{ATIME}(s^2(n)v(n)). \end{aligned}$$

Exercises

[7.1] Verify the identities in section 2 on interval algebra.

[7.2] Some additional properties of the lattice *INT*:
(a) Show that the two orderings $\le$ and $\sqsubseteq$ are 'complementary' in the following sense: for all I and J, either I and J are $\le$-comparable or they are $\sqsubseteq$-comparable.
(b) Show that I and J are both $\le$-comparable and $\sqsubseteq$-comparable iff $I \approx J$ where we write $[x, y] \approx [x, v]$ if $x = u$ or $y = v$.
(c) Extend $\wedge$ and $\vee$ to arbitrary sets $S \subseteq INT$ of intervals: denote the meet and join of S by $\wedge S$ and $\vee S$. Show that *INT* forms a complete lattice with least element 0 and greatest element 1.

[7.3] Consider the 2-ary Boolean function *inequivalence* (also known as *exclusive-or*) denoted $\not\equiv$. We want to extend this to a function on intervals in *INT*. One way to do this is to use the equation

$$x \not\equiv y = (x \wedge \neg y) \vee (\neg x \wedge y)$$

valid for Boolean values, but now interpreting the $\wedge, \vee$ and $\neg$ as functions on intervals. For instance,

$$\begin{aligned}[0.2, 0.5] \not\equiv [0.6, 0.7] &= ([0.2, 0.5] \wedge [0.3, 0.4]) \vee ([0.5, 0.8] \wedge [0.6, 0.7]) \\ &= [0.2, 0.4] \vee [0.5, 0.7] = [0.5, 0.7].\end{aligned}$$

Can you find other equations for $\not\equiv$ that are valid in the Boolean case such that the extension to intervals are not the same function?

[7.4] * (i) Consider generalized probabilistic machines in which we allow k-ary versions of the coin-tossing function, $\oplus\!\!\!\!\!\!\;\top_k$ for all $k \ge 2$. Show that these can be simulated by ordinary probabilistic machines with at most a constant factor slow-down in time.
(ii) Show the same result for stochastic machines where we now allow $\oplus\!\!\!\!\!\!\;\top_k, \oplus_k, \otimes_k$ for all k.

[7.5] Our definition of B-machines attaches a basis function $\gamma(q) \subseteq B$ to each state q. A more general definition is to attache a basis function $\gamma(q, a_0, \ldots, a_k)$ to each combination $(q, a_0, \ldots, a_k)$ where a_i are tape symbols valid for tape i. Under what conditions is it possible for our (official) choice machines to simulate the behavior of these more general choice machines.

[7.6] (Hong) Consider basis sets B that are subsets of the 16 Boolean functions on 2 variables. As usual, we assume that the identity, 0 and 1 constant functions are not explicitly mentioned when we display B. For two bases B, B', say that B *linearly simulates* B' if for every B'-choice machine M$'$, there is a B-machine M accepting the same language such that if M$'$ accepts in time $t(n)$ then M accepts in time $O_{B,B'}(t)$. Say B and B' are *linearly equivalent* if they can linearly simulate each other.
(a) Prove that every such basis set B is linearly equivalent to one of the following 5 bases:

$$B_0 := \emptyset, B_1 := \{\vee\}, B_2 := \{\wedge\}, B_3 := \{\overline{\oplus}\}, B_4 := \{\vee, \wedge\}$$

where $\overline{\oplus}$ is the exclusive-or function.
(b) By simple set inclusions, it is clear that B_0 can be linearly simulated by the others and B_4 can linearly simulate B_1 and B_2. Show that B_4 can also linearly simulate B_3. **Hint:** Use the same idea as the proof for elimination of negation.
(c)** Show that these 5 classes are distinct up to linear equivalence. (Note: it is known that B_1 is distinct from B_0.)

[7.7] ** Let $B = \{\vee, \oplus\!\!\!\!\!\!\;\top, \neg\}$. Can negation be eliminated from B-machines operating in polynomial time?

[7.8] Generalize choice machines by allowing values from any $\sqsubseteq$-partially ordered set F that has a $\sqsubseteq$-mimimal element $\perp \in F$ and such that any $\sqsubseteq$-monotonic non-decreasing sequence has a least upper bound in F. In particular, let F be the Borel sets (obtain from *INT* under the operation of intersection, difference and countable unions).

[7.9] * Extend valuation theory for acceptors to transducers.

[7.10] (Paul, Praus, Reischuk) Construct a simple (i.e., one work-tape and no input tape) alternating machine that accepts palindromes in linear time and a constant number of alternation. **Hint:** Guess the positions (in binary) of about $\log n$ equally spaced-out input symbols, writing these directly under the corresponding symbols. The positions of all the other symbols can be determined relative to these guessed positions.

[7.11] (Paul, Praus, Reischuk) Show that if a language can be accepted by an alternating Turing machine in time $t(n)$ then it can be accepted by a simple alternating Turing machine in time $O(t(n))$. **Hint:** For a computation path $C_1 \vdash C_2 \vdash, \ldots, \vdash C_m$, let its *trace* be $\tau_1, \tau_2, \ldots, \tau_m$ where τ_i contains the state, the symbols scanned on each of the tapes (including the input tape) and the direction of each tape head in the transition $C_i \vdash C_{i+1}$. The head directions are undefined for τ_m. (Note: this definition of trace does not include head positions, in contrast to a similar definition in chapter 2.) Begin by guessing the trace of the paths in an accepting computation tree – you must use universal and existential guessing corresponding to the state in τ_i. To verify correctness of the guess, the technique for the previous question is useful.

[7.12] (Paterson, Paul, Praus, Reischuk) For all $t(n)$, if a language is accepted by a nondeterministic simple Turing machine in time $t(n)$ then it can be accepted by an alternating machine in time $n + t^{1/2}(n)$.

[7.13] Show the tight complexity relationships between the ordinary SAM's and the addressable-input version of SAM's. More precisely, give efficient time/space/reversal simulations of the addressable-input machines by ordinary machines.

[7.14] Rederive the various simulation of SAM's in this chapter in the case where the SAMs is the addressable-input model. In particular, show that $ATIME(t) \subseteq DSPACE(t)$.

[7.15] Obtain a lower bound on the space-time product of alternating Turing machines which accepts the palindrome language L_{pal}.

[7.16] * Obtain the tight bound of $\Theta(\log n)$ on the alternating time complexity for the multiplication problem define as follows:

$$L_{mult} = \{x\#y\#z\# : x, y, z \in \{0,1\}^*, x \cdot y = z\}.$$

Use the addressable-input model of machine.

[7.17] ** A very natural function that one would like to add to our basis sets is the cut-off function $\delta_{\frac{1}{2}}$ defined at the end of section 2. Note that it gives us a model of oracle calls in which the complexity of the oracle machine is taken into account. Of course, this is not continuous: extend the theory of valuations to allow monotonic, piece-wise continuous functions. (A functions is piece-wise continuous if it has a finite number of discontinuities.)

[7.18] Explore the power of choice machines: suppose the basis functions are rational, linear convex combinations of their arguments: more precisely, the valuation functions f have the form

$$f(x_1, \ldots, x_n) = \sum_{i=1}^{n} a_i x_i$$

where the a_i's are positive rational numbers depending on f and $\sum_i a_i = 1$. How do these machines compare to SAMs?

[7.19] Give a sufficient condition on a family F of complexity functions such that $ATIME(F) = PrA\text{-}TIME(F) = DSPACE(F)$. **Hint:** Consider the case $F = n^{O(1)}$.

[7.20] * Can the alternating version of I. Simon simulation (section 6) be improved? In particular, try to improve $DREVERSAL(r) \subseteq ATIME(r^3)$.

[7.21] In section 7, we saw a reduction of Turing machine simulation to graph pebbling, due to Hopcroft-Valiant-Paul. An alternative definition of the edge set E is this: "define (j, i) to be an edge of G if $j < i$ and there is some block b visited in the ith time interval and last visited in the jth time interval." Using this definition of G, what modifications are needed in the proof showing $DTIME(t) \subseteq DSPACE(t/\log t)$?

[7.22] What is the number of alternations in the proof of $DTIME(t) \subseteq ATIME(t/\log t)$? Can this be improved?

[7.23] Show that if $t(n)$ is time constructible then co-$ATIME(t(n)) \subseteq ATIME(n + t(n))$. HINT: why do you need the "$n+$" term?
NOTE: For instance, if $t(n) \geq 2n$, $ATIME(n+t(n)) = ATIME(t(n))$, and so $ATIME(t(n))$ is closed under complementation. This is essentially the result of Paul and Reischuk.

[7.24] * A **probabilistic-alternating finite automaton** (pafa) is a PAM with no work-tape and whose input tape is restricted so that in each step, the input head must move to the right. Moreover, the machine must halt upon reading the first blank symbol after the input word. The special cases of alternating finite automaton (afa) or probabilistic finite automaton (pfa) is defined analogously.
(i) (Chandra-Kozen-Stockmeyer) Prove that an afa accepts only regular languages.
(ii) (Starke) Show the following language (well-known to be non-regular) $L = \{0^m 1^n : m \geq n\}$ can be accepted by a pfa.

[7.25] Recall an alternating finite automaton (afa) defined in the previous question. Let us define a **generalized afa** (gafa) to be an afa in which each state q has an arity $k(q) \geq 0$ and is assigned a **generalized Boolean function**

$$\gamma(q) : \{0, 1, \perp\}^{k(q)} \to \{0, 1, \perp\}.$$

(i) Let M be an alternating Turing acceptor with no work-tapes and whose input head cannot move right. For the purposes of this question, assume that M has only $\wedge$- and $\vee$-states. So M looks very similar to an afa, except that it can make "ϵ-moves": this is a terminology from automata theory: an ϵ-move is one where the automaton can change state without moving its input head, and this is performed nondeterministically. Show that such moves can be eliminated in the sense that there is a generalized afa that accepts the same language as M.
(ii) (Open) Characterize the kinds of generalized Boolean functions that can arise in the generalized afa in part (i).

[7.26] (Freivalds) Show a probabilistic finite automata to recognize the language $\{0^n 1^n : n \geq 1\}$ with bounded error. **Hint:** We are basically trying to check that the number of 1's and number of 0's are equal. Show that the following procedure works:
a. Choose a coin with probability $p << \frac{1}{2}$ of getting a *head*.
b. Toss coin for each 0 in input and each 1 in input. If we get all *heads* for 0's and at least one *tail* for the 1's then we say we have a 0-*win*. If we get at least one *tail* for the 0's and all *heads* for the 1's, we have a 1-*win*. All other outcomes result in a *tie*.
c. Repeat this experiment until we have at least k 0-wins or k 1-wins. We accept if and only if there is at least one 1-win and at least one 0-win.
(For more information, see [10].)

[7.27] (Ruzzo, King) Show the following simulations among measures for alternating computation, as stated in the concluding section: for $s(n) \geq \log n$,

$$\begin{aligned} \textit{A-SPACE-SIZE}(s(n), z(n)) &\subseteq \textit{ATIME}(s(n) \log z(n)), \\ \textit{A-SPACE-WIDTH}(s(n), w(n)) &= \textit{NSPACE}(s(n)w(n)), \\ \textit{A-SPACE-VISIT}(s(n), v(n)) &\subseteq \textit{ATIME}(s^2(n)v(n)). \end{aligned}$$

[7.28] (King) Recall the definitions of branching and width resource for alternating machines in the concluding section. Show that the branching resource (simultaneously bounded with space) has the linear complexity reduction property: for $s(n) \geq \log n$,

$$\textit{A-SPACE-BRANCH}(s(n), b(n)) = \textit{A-SPACE-BRANCH}(s(n), b(n)/2).$$

Show the same result for width resource: for $s(n) \geq \log n$,

$$\textit{A-SPACE-WIDTH}(s(n), w(n)) = \textit{A-SPACE-WIDTH}(s(n), w(n)/2).$$

[7.29] Show that if a graph with goal node (G, i_0) has a pebbling tree with pebbling time t then it can be pebbled with t pebbles. Is the converse true?

[7.30] (Paul, Tarjan, Celoni, Cook)
(a) A *level graph* is a directed acyclic graph with bounded in-degree such that the vertices can be partitioned into 'levels' and edges only go from level i to level $i+1$. Show that every level graph on n vertices can be pebbled using $O(\sqrt{n})$ pebbles.
(b) Show an infinite family of graphs with indegree 2 that requires $\Omega(\sqrt{n})$ pebbles to pebble certain vertices.

Bibliography

[1] L. Adleman and M. Loui. Space-bounded simulation of multitape Turing machines. *Math. Systems Theory*, 14:215–222, 1981.

[2] László Babai and Shlomo Moran. Arthur-Merlin games: a randomized proof system, and a hierarchy of complexity classes. *Journal of Computers and Systems Sciences*, 36:254–276, 1988.

[3] A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *J. ACM*, 28:1:114–133, 1981.

[4] A. Condon and R. Ladner. Probabilistic game automata. *Journal of Computers and Systems Sciences*, 36:452–489, 1988.

[5] Peter Crawley and Robert Dilworth. *Algebraic theory of lattices.* Prentice-Hall, 1973.

[6] P. Dymond and M. Tompa. Speedups of deterministic machines by synchronous parallel machines. *Journal of Computers and Systems Sciences*, 30:149–161, 1985.

[7] J. T. Gill. Computational complexity of probabilistic Turing machines. *SIAM J. Comp.*, 6(4):675–695, 1977.

[8] S. Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proofs. *17th ACM Symposium STOC*, pages 291–304, 1985.

[9] Shafi Goldwasser and Michael Sipser. Private coins versus public coins in interactive proof systems. *18th ACM Symposium STOC*, pages 59–68, 1986.

[10] Albert G. Greenberg and Alan Weiss. A lower bound for probabilistic algorithms for finite state machines. *Journal of Computer and System Sciences*, 33:88–105, 1986.

[11] Jia-wei Hong. *Computation: Computability, Similarity and Duality.* Research notices in theoretical Computer Science. Pitman Publishing Ltd., London, 1986. (available from John Wiley & Sons, New York).

[12] J. E. Hopcroft, W. J. Paul, and L. G. Valiant. On time versus space. *Journal of Algorithms*, 24:332–337, 1977.

[13] Kimberley N. King. Measures of parallelism in alternating computation trees. *ACM Symp. on Theory of Computing*, 13:189–201, 1981.

[14] Burkhard Monien and Ivan Hal Sudborough. On eliminating nondeterminism from Turing machines which use less than logarithm worktape space. In *Lecture Notes in Computer Science*, volume 71, pages 431–445, Berlin, 1979. Springer-Verlag. Proc. Symposium on Automata, Languages and Programming.

[15] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1966.

[16] Christos H. Papadimitriou. Games against nature. *Journal of Computers and Systems Sciences*, 31:288–301, 1985.

[17] Michael S. Paterson. Tape bounds for time-bounded Turing machines. *Journal of Computers and Systems Sciences*, 6:116–124, 1972.

[18] W. J. Paul, Ernst J. Praus, and Rüdiger Reischuk. On alternation. *Acta Informatica*, 14:243–255, 1980.

[19] W. J. Paul, R. E. Tarjan, and J. R. Celoni. Space bounds for a game on graphs. *Math. Systems Theory*, 11:239–251, 1977. (See corrections in Math. Systems Theory, 11(1977)85.).

[20] Walter L. Ruzzo. Tree-size bounded alternation. *ACM Symp. on Theory of Computing*, 11:352–359, 1979.

[21] Martin Tompa. An improvement on the extension of Savitch's theorem to small space bounds. Technical Report Technical Report No. 79-12-01, Department of Computer Sci., Univ. of Washington, 1979.

[22] Klaus Weihrauch. *Computability*. Springer-Verlag, Berlin, 1987.

[23] Chee K. Yap. On combining probabilistic and alternating machines. Technical report, Univ. of Southern California, Comp. Sci. Dept., January 1980. Technical Report.

Contents