Introduction to Computational Complexity

Martin Tompa
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington, U.S.A. 98195

Contents

List of Figures

Preface

These are the lecture notes from a graduate course on Computational Complexity taught at the University of Washington. This topic fits in the middle of three of the fundamental areas of the Theory of Computation, which can be summarized with respect to their approaches to computational problems as follows:

- Computability: Determine whether an algorithm exist that solves a given problem.
- Computational Complexity: For those problems that are computable, determine a coarse analysis of their time and space requirements. Such analyses may well ignore the differences between polynomials, and concentrate on the difference between polynomial and exponential behavior.
- Analysis of Algorithms: For those problems that can be solved in polynomial time, determine a more exact analysis of their time and space requirements.

This text adopts some approaches that will appear unconventional. For example, alternating Turing machines are introduced very early, and deterministic and nondeterministic Turing machines treated as special cases. This simplifies many proofs, such as that of Savitch's Theorem (Theorem 4.14), the $\mathcal{P}$-completeness of the circuit value problem (Theorem 7.11), the $\mathcal{NP}$-completeness of the satisfiability problem (Theorem 8.7), and the $\mathcal{PSPACE}$-completeness of the quantified Boolean formula problem (Theorem 10.3).

Another unconventional approach is to use log space reducibility rather than polynomial time reducibility when reducibility is first introduced in Chapter 5, and to begin with $\mathcal{NL}$-completeness rather than the more important $\mathcal{NP}$-completeness. The reason for this decision is twofold. First, the generic reduction in proving the $\mathcal{NL}$-completeness of the directed graph connectivity problem (Theorem 6.5) is much simpler than the generic reduction normally used to prove the $\mathcal{NP}$-completeness of the satisfiability problem, and thus gives the student a good "warmup" for the more important completeness proofs to come. Second, the $\mathcal{NP}$-completeness proof of the satisfiability problem given in Theorem 8.7 is greatly simplified by the machinery built up in Chapter 7 on $\mathcal{P}$-complete problems.

I am indebted to Larry Ruzzo for exposing me to many of the approaches in this text, and for countless discussions about this material. I am thankful for the many students who attended lectures faithfully, served as notetakers, asked embarassing questions, made perceptive comments, and generally make teaching exciting and rewarding.

— Martin Tompa

Part I

Models, Measures, and General Relationships

Chapter 1

Computational Models

1.1. Why Turing Machines?

There are two reasons for the use of Turing machines as a model of computation. The first is that they provide a compromise between the mathematical simplicity of models such as the lambda calculus or recursive functions, and the computational realism one would find in random access machines or Pascal programs. (Note, for instance, that the entire semantics of Turing machines will be given in the short Section 1.2.) The second reason is that the choice actually makes little difference, since all reasonable models are equivalent in the domains of computability and coarse analysis.

1.2. Definition of Alternating Turing Machines

Alternating Turing machines were introduced by Chandra, Kozen, and Stockmeyer [3]. For simplicity, the particular variant defined here will have one tape and be used for set recognition. It should be noted that the definitions can be extended easily to multiple tapes or function computation.

Definition 1.1: An *alternating Turing machine* is an 8-tuple $M = (Q, E, A, F, \Gamma, \Sigma, q_0, \delta)$ satisfying the following properties:

- Q is a finite set (the "states"), partitioned into 3 subsets:
  1. E (the "existential states"),
  2. A (the "universal states"), and
  3. F (the "final states").
- Γ is a finite set (the "worktape alphabet").
- $\Sigma \subseteq \Gamma$. (Σ is the "input alphabet".)
- $\not{b} \in \Gamma - \Sigma$. ($\not{b}$ is the "blank symbol".)
- $Q \cap \Gamma = \emptyset$.

- $q_0 \in Q$. (q_0 is the "start state".)
- $\delta : (Q - F) \times \Gamma \to 2^{Q \times (\Gamma - \{\not b\}) \times \{L,R\}}$. ($\delta$ is the "transition function".)

The next two definitions specify what the computation steps of a Turing machine look like.

Definition 1.2: A *configuration* of an alternating Turing machine M is a string uqv, where $q \in Q$, and $u, v \in \Gamma^*$.

(The intuition is that M is in state q, $uv \in \Gamma^*$ is the nonblank content of the tape, and M's head is reading the first symbol of v, or $\not b$ if $v = \epsilon$, the empty string. In general, the configurations uqv, $uqv\not b$, and $\not b uqv$ are considered equivalent, so we can always assume that $u \neq \epsilon$ and $v \neq \epsilon$.)

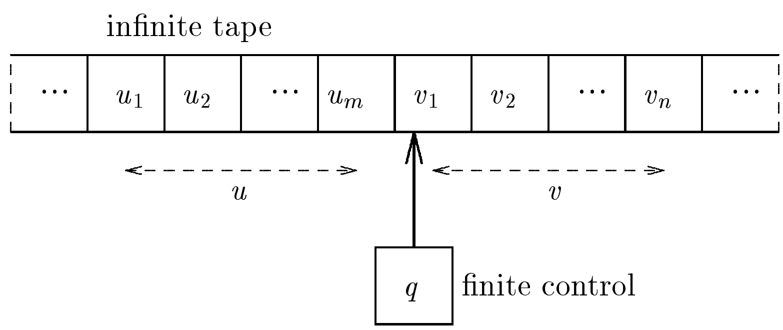

Figure 1.1: Visualizing an Alternating Turing Machine in Configuration uqv

Definition 1.3: The *one-step transition relation* $\vdash_M$ for an alternating Turing machine M is defined as follows. For any $u, v \in \Gamma^*$, $q, q' \in Q$, and $a, b, c \in \Gamma$,

$$uqav \vdash_M ucq'v \quad \text{if and only if} \quad (q', c, R) \in \delta(q, a), \text{ and}$$

$$ubqav \vdash_M uq'bcv \quad \text{if and only if} \quad (q', c, L) \in \delta(q, a).$$

Definition 1.4: If P and P' are configurations of M and $P \vdash_M P'$, P' is called an *immediate successor* of P.

The remaining definitions specify how a Turing machine starts and finishes.

Definition 1.5: A configuration uqv is *halting* if and only if there is no configuration P such that $uqv \vdash_M P$.

(This can happen if $q \in F$, of if $\delta(q, a) = \emptyset$, where a is the first symbol of v.)

Definition 1.6:

- If $q \in F$, or $q \in A$ and uqv is halting, then uqv is a *final* configuration.
- If $q \in E$, then uqv is an *existential* configuration.
- If $q \in A$, then uqv is a *universal* configuration.

Definition 1.7: A configuration P is said to be *accepting* if and only if one of the following three conditions holds:

1. P is a final configuration, or
2. P is an existential configuration and $(\exists Q)((P \underset{M}{\vdash} Q)$ & $(Q$ is accepting$))$, or
3. P is a universal configuration and $(\forall Q)((P \underset{M}{\vdash} Q) \Rightarrow (Q$ is accepting$))$.

Definition 1.8: M *accepts* the input string $x \in \Sigma^*$ if and only if q_0x is an accepting configuration.

Definition 1.9: The *language* accepted by M is $L(M) = \{x \in \Sigma^* \mid M \text{ accepts } x\}$.

1.3. Nondeterministic and Deterministic Turing Machines

Given the definition of alternating Turing machines, it is straightforward to define the more familiar deterministic and nondeterministic Turing machines:

Definition 1.10: A *nondeterministic Turing machine* is an alternating Turing machine for which $A = \emptyset$.

Definition 1.11: A *deterministic Turing machine* is a nondeterministic Turing machine for which $|\delta(q, a)| \leq 1$ for all $q \in Q - F$ and $a \in \Gamma$.

Our alternating Turing machines are defined to have one tape, but the definition can be extended to k tapes. In this case, configurations become k-tuples $(u_1qv_1, u_2qv_2, \ldots, u_kqv_k)$, and the transition function, as an example of the generalization, becomes $\delta : (Q - F) \times \Gamma^k \to 2^{Q \times (\Gamma - \{\not{b}\})^k \times \{L,N,R\}^k}$, where N denotes no move of the tape head on the corresponding tape.

1.4. Examples of Turing Machines

Example 1.12: Given an undirected graph $G = (V, E)$, a *Hamiltonian cycle* is a cycle that passes through each vertex exactly once. Assume $|V| = v$. The nondeterministic Turing machine M described below, given an encoding of such a graph, determines whether it has a Hamiltonian cycle. M nondeterministically chooses and records on its tape an ordered list $(u_0, u_1, \ldots, u_{v-1})$ of v of the vertices. If any vertex appears twice on this list, M rejects (i.e., halts in a nonfinal configuration). If there is an i such that $\{u_i, u_{(i+1) \bmod v}\} \notin E$, M rejects. Otherwise, M accepts (i.e., halts in a final configuration).

Example 1.13: As an example of an alternating Turing machine, consider the problem of determining if white has a winning strategy in chess. An alternating Turing machine M maintains an encoding of the current board position on its tape and, beginning from the initial position, alternates between

- existentially choosing white's next move from among the legal alternatives and
- universally choosing black's next move from among the legal alternatives,

until the game reaches a terminal configuration. M then accepts if and only if white has won in this configuration.

In words, what M is doing is checking that "there is a first move by white such that, for all second moves by black, there is a third move by white such that ... white wins."

1.5. Computation Trees and Accepting Subtrees

Definition 1.14: Let M be an alternating Turing machine, and P be a configuration of M. The *P-computation tree of M* is a tree (possibly infinite) with nodes labeled by configurations of M such that

1. the root is labeled P, and
2. each node labeled Q has a child labeled R for each R such that $Q \underset{M}{\vdash} R$.

Definition 1.15: Let M be an alternating Turing machine, and x an input. The *computation tree of M on x* is the $(q_0 x)$-computation tree of M.

Example 1.16: The computation tree of a deterministic Turing machine on a given input is a (possibly infinite) simple path. Since, in each configuration, there is at most one possible choice of the next configuration, the computation tree never branches. Note that the computation tree is finite in this case if and only if the Turing machine halts. Notice also that the same configuration may label two different nodes of the computation tree, but in the case of a deterministic Turing machine that would mean the machine never halts.

Definition 1.17: Let M be an alternating Turing machine, and P a configuration of M. An *accepting P-subtree A of M* is a subtree of the P-computation tree C of M, with the following properties:

1. A includes the root of C,

2. for every node v of A labeled by an existential configuration, v has one of its children from C,

3. for every node v of A labeled by a universal configuration, v has all of its children from C,

4. A has no infinite paths, and

5. all of A's leaves are labeled by final configurations.

Definition 1.18: Let M be an alternating Turing machine, and x an input. An *accepting subtree* of M on x is an accepting (q_0x)-subtree of M.

Example 1.19: An accepting subtree of a nondeterministic Turing machine is a simple (finite) path whose internal nodes are labeled by existential configurations and whose single leaf is labeled by a final configuration. In Example 1.12, for instance, this path would correspond to the computation that correctly guesses (and verifies) a Hamiltonian cycle in the input graph.

The following theorem relates the definition of acceptance to the notion of an accepting subtree.

Theorem 1.20: M accepts x if and only if there is an accepting subtree of M on x.

Proof: Left as an exercise. (Hint: Using induction, prove the generalization that P is an accepting configuration if and only if there is an accepting P-subtree of M.) $\square$

Definition 1.21: A set is *recursively enumerable* if and only if it is accepted by some deterministic Turing machine.

Theorem 1.22: Nondeterministic Turing machines and alternating Turing machines accept exactly the recursively enumerable sets.

Proof: That they accept at least the recursively enumerable sets follows from the fact that deterministic Turing machines are a special case of nondeterministic Turing machines and alternating Turing machines. The other direction is a corollary of results that will be proved in Chapter 4. These results (Theorems 4.7 and 4.11) show not only that alternating Turing machines can be simulated by deterministic Turing machines, but how efficient that simulation is. $\square$

1.6. Exercises

1. (a) Give definitions analogous to those in Section 1.2 for a k-tape alternating Turing machine.

   (b) Prove that any k-tape alternating Turing machine can be simulated by a one-tape alternating Turing machine.

2. (a) Describe a nondeterministic Turing machine that, given the encoding of a directed graph G and two distinguished vertices s and t, accepts if and only if there is a path in G from s to t.

   (b) Describe a deterministic Turing machine that accepts the same language.

3. Prove Theorem 1.20.

Chapter 2

Complexity Measures

Having established in Chapter 1 how and what a Turing machine computes, we now turn to the definition of how efficiently it computes.

2.1. Definitions

The first two definitions establish the amount of time and space used by a particular Turing machine on a particular input.

Definition 2.1: An alternating Turing machine M on input x *runs in time at most* t if and only if there is an accepting subtree of M on x whose height is at most t.

Definition 2.2: An alternating Turing machine M on input x *runs in space at most* t if and only if there is an accepting subtree of M on x each of whose nodes is labeled by a configuration of length at most t.

The next definition specifies the amount of time or space used by a particular Turing machine on its worst case inputs.

Definition 2.3: An alternating Turing machine M runs in *time* (*space*) $T(n)$ if and only if, for every $x \in L(M)$, M on input x runs in time (respectively, space) at most $T(|x|)$.

Notice that this definition puts no bound on the time or space used by M on any x not accepted by M.

Definition 2.4: $\mathrm{DTIME}(T(n)) = \{L \mid$ there is a multitape deterministic Turing machine that accepts L and runs in time $T(n)\}$.

Definition 2.5: $\mathrm{DSPACE}(T(n)) = \{L \mid$ there is a multitape deterministic Turing machine that accepts L and runs in space $T(n)\}$.

Definition 2.6: $\mathrm{NTIME}(T(n)) = \{L \mid$ there is a multitape nondeterministic Turing machine that accepts L and runs in time $T(n)\}$.

Definition 2.7: NSPACE($T(n)$) = $\{L \mid$ there is a multitape nondeterministic Turing machine that accepts L and runs in space $T(n)\}$.

Definition 2.8: ATIME($T(n)$) = $\{L \mid$ there is a multitape alternating Turing machine that accepts L and runs in time $T(n)\}$.

Definition 2.9: ASPACE($T(n)$) = $\{L \mid$ there is a multitape alternating Turing machine that accepts L and runs in space $T(n)\}$.

Convention 2.10: Throughout this text, n will refer to the length of the input string.

2.2. Review of Order Notation

Let $\mathcal{R}$ be the set of real numbers, and $f, g : \mathcal{R} \rightarrow \mathcal{R}$ be two functions. The following definitions provide a convenient notation for comparing the rates of growth of f and g. (See Knuth [25] for more discussion.)

- $f(n) = O(g(n))$ if and only if $(\exists c)(\exists n_0)(\forall n \geq n_0)\ |f(n)| \leq cg(n)$.
- $f(n) = \Omega(g(n))$ if and only if $(\exists c)(\exists n_0)(\forall n \geq n_0)\ f(n) \geq cg(n)$.
- $f(n) = \Theta(g(n))$ if and only if $(\exists c)(\exists c')(\exists n_0)(\forall n \geq n_0)\ cg(n) \leq f(n) \leq c'g(n)$.
- $f(n) = o(g(n))$ if and only if $\lim_{n\rightarrow\infty} \dfrac{f(n)}{g(n)} = 0$.
- $f(n) = \omega(g(n))$ if and only if $\lim_{n\rightarrow\infty} \dfrac{f(n)}{g(n)} = \infty$.

2.3. Sublinear Space Bounds

The problem with the definition of space as given in Definition 2.2 is that there are no machines with sublinear (i.e., $o(n)$) space bounds: the entire input, whose length is n, is included in the initial configuration, and hence in the space bound. In order to extend these definitions to sublinear space bounds, we will assume the following modifications from now on, unless specified otherwise:

Convention 2.11:

1. There will be a separate input tape of length n plus two endmarkers. This tape is read-only.
2. Configurations contain the position h of the input tape head in binary, but not the contents of the input tape. Therefore, a configuration of a machine with an input tape and k additional worktapes looks like $(h, u_1qv_1, u_2qv_2, \ldots, u_kqv_k)$.
3. Since the input x is no longer part of the configuration, the notation $P \vdash_{M} Q$ will be replaced by $P \vdash_{M,x} Q$.

4. The definition of space will not include the length of the input tape, nor the length of the input head position h.

Example 2.12: $L = \{ww \mid w \in \{0,1\}^*\}$ can be accepted by a deterministic Turing machine in space $O(\log n)$. This can be accomplished as follows. First M uses a counter to calculate the length n of the input. This takes $O(\log n)$ space on a worktape. If n is odd, M rejects the input. If n is even, M starts at the first input symbol, storing it in the finite control, and uses a second counter to move the input head $n/2$ cells to the right. M rejects if this is not the same symbol as in the finite control. Otherwise M moves the input head left $n/2 - 1$ cells and repeats this check for the second input symbol. M performs this check for each symbol until the input head moves off the right end of the input, at which point M accepts. Each counter contains an index between 1 and n, so M uses space $O(\log n)$.

2.4. Sublinear Time Bounds

The extension to sublinear space bounds suggests doing the same for time bounds. One might argue that any interesting Turing machine cannot have a sublinear time bound, as it takes at least n steps just to read the input. However, this argument fails for nondeterministic or alternating Turing machines, provided they are given random access to the input tape rather than sequential access. Consider, for instance, the language that consists of all strings that contain a 1. A nondeterministic machine could guess and record the position of the 1 in time $O(\log n)$, and use its supposed random access to the input to verify that there is a 1 in that position. This idea of random access or indexing is formalized in the following definition.

Definition 2.13: An *indexing Turing machine* has an input tape with no head, and a special "index tape" in addition to its other worktapes. The next move depends on the ith input symbol if the nonblank portion of the index tape to the left of the head is the binary encoding of i for $1 \le i \le n$, and depends on the endmarker symbol otherwise. The length of the nonblank portion of the index tape is included in the machine's space bound.

Convention 2.14: Unless explicitly stated otherwise, we will assume from now on that all deterministic and nondeterministic Turing machines are not indexing machines, but that all other alternating Turing machines are.

Example 2.15: An indexing nondeterministic Turing machine can calculate the length n of the input as follows. On the index tape guess n one bit at a time, verify that the nth input symbol is in the input alphabet, and verify that the $(n+1)$st symbol is an endmarker, rejecting if either of these is not the case. It takes time $O(\log n)$ to do this.

Example 2.16: The complement of the language $L = \{ww \mid w \in \{0,1\}^*\}$ from Example 2.12 can be accepted by an indexing nondeterministic Turing machine in time $O(\log n)$. Compute the length n of the input as in Example 2.15. If n is odd, accept. Otherwise, guess i satisfying $1 \le i \le n/2$, and accept if and only if $x_i \ne x_{i+n/2}$, where $x_1 x_2 \ldots x_n$ is the input. The sum $i + n/2$ and the other computations necessary can be calculated in time $O(\log n)$.

Any nondeterministic Turing machine requires time at least n to accept the language L from Example 2.16, but it only takes an alternating Turing machine time $O(\log n)$ to do the same. These are both left as simple exercises. Here is an example of a slightly more interesting language that can be accepted by an alternating Turing machine in $O(\log n)$ time:

Example 2.17: Let L be the set of strings $A\#B$ that are encodings of two equal sets (which we will also refer to as A and B). More specifically, $A = A_0\$A_1\$\ldots\$A_a$ and $B = B_0\$B_1\$\ldots\$B_b$, where for simplicity $A_i, B_j \in \{0,1\}^m$ for some $m = 2^p - 1$.

The alternating Turing machine first determines n, m, a, b, and h where h is the index of the marker $\#$, in a manner similar to that of Example 2.15. The goal is to accept if and only if $A \subseteq B$ and $B \subseteq A$, that is,

$$(\forall i)(\exists j)A_i = B_j \text{ and } (\forall j)(\exists i)B_j = A_i.$$

The machine will universally check each of these two conjuncts. Here is how it checks the first, the second being done in an analogous manner:

1. Universally choose and record i, with $0 \leq i \leq a$.
2. Existentially choose and record j, with $0 \leq j \leq b$.
3. Universally choose and record k, with $1 \leq k \leq m$.
4. Accept if and only if $A_{i,k} = B_{j,k}$, where these are the kth bits of A_i and B_j, respectively.

The machine requires time $O(\log a)$ to universally choose i, time $O(\log b)$ to existentially choose j, and time $O(\log m)$ to universally choose k; furthermore, a, b, and m are each at most n. To find $B_{j,k}$ in step 4, the alternating Turing machine needs to calculate $h + j2^p + k$ on its index tape, which can be done in $O(\log n)$ time.

Alternating Turing machines, and particularly indexing alternating Turing machines, may seem unmotivated at first, but there is a direct correspondence to Boolean circuits (Ruzzo [39]) that makes them an excellent model of parallel computations; this will be explored further in Section 7.5. Furthermore, they will turn out to be extremely helpful in understanding the complexity of natural problems, much as nondeterministic Turing machines have proven to be.

Add other forward pointers.

2.5. Exercises

1. Let I be the set of invertible $\sqrt{n} \times \sqrt{n}$ matrices over $\mathcal{Z}_2$ (the integers modulo 2). Show that $I \in \mathrm{NTIME}(O(n^{3/2})) \cap \mathrm{NSPACE}(O(\sqrt{n}))$.

2. Prove that an indexing deterministic Turing machine can compute the length n of its input in time $O(\log n)$.

3. Show that the language L from Example 2.16 cannot be recognized by an indexing nondeterministic Turing machine in time less than n.

4. Prove that Set Equality (from Example 2.17) can be solved by an alternating Turing machine in time $O(\log n)$ for *arbitrary* subsets of $\{0, 1\}^*$.

5. Show that every regular language is in $\mathrm{DTIME}(O(n)) \cap \mathrm{DSPACE}(O(1))$.

6. Show that every regular language is in $\mathrm{ATIME}(O(\log n))$.

Chapter 3

The Power of Increasing Resources

Now that the models and complexity measures have been defined and understood, we are in a position to pose some typical motivating questions in complexity theory.

1. More vs. Less: Given more time or space, can machines in a fixed model necessarily solve more difficult problems? This is the topic of this chapter.

2. Resource and Model comparison: What is the relationship between time and space? What are the relationships among deterministic, nondeterministic, and alternating Turing machines? This is the topic of Chapter 4.

3. Feature comparison: Are $k+1$ tape machines more powerful than k tape machines? Are indexing machines more powerful than non-indexing machines? Some of these questions will be addressed in each of these chapters.

3.1. Constant Factor Speedup Theorems

This section answers the simplest questions of the first type: doubling the space or time adds no computational power.

Theorem 3.1: For every $\epsilon > 0$ and every space bound $S(n)$,

$$\mathrm{DSPACE}(S(n)) \subseteq \mathrm{DSPACE}(\lceil \epsilon S(n) \rceil),$$

and similarly for NSPACE and ASPACE.

Proof: Let $c = \lceil 1/\epsilon \rceil$. Let M be a deterministic Turing machine with worktape alphabet Γ that runs in space $S(n)$. Construct a deterministic Turing machine N with worktape alphabet Γ^c, and compress every c symbols on each worktape of M into one symbol on the corresponding worktape of N. (See Figure 3.1 for an example with $\epsilon = 1/3$.) The remaining details are left as an exercise. □

Theorem 3.2: For every $\epsilon > 0$ and every time bound $T(n) \geq n$,

$$\mathrm{DTIME}(T(n)) \subseteq \mathrm{DTIME}(n + \lceil \epsilon T(n) \rceil),$$

and similarly for NTIME.

$\cdots$	1	2	3	4	5	6	$\cdots$

is compressed to

	1	4	
$\cdots$	2	5	$\cdots$
	3	6	

Figure 3.1: Compressing a Worktape for Constant Space Reduction

Proof: One might be inclined to use the same proof as in Theorem 3.1, but this falls a bit short. If the symbols are grouped as in that proof, the Turing machine might oscillate between adjacent cells on the uncompressed tape that were in different cells on the compressed tape (for instance, cells 3 and 4 in Figure 3.1), so the construction fails to yield any speedup. The solution is to overlap the contents of adjacent compressed cells.

Let $c = 2\lceil 1/\epsilon \rceil$. Let M be a deterministic Turing machine with worktape alphabet Γ that runs in time $T(n) \geq n$. We construct a deterministic Turing machine N with worktape alphabet Γ^{3c-2}, whose compressed tapes are illustrated in Figure 3.2 for the case $\epsilon = 1/2$.

$\cdots$	3	4	5	6	7	8	9	$\cdots$
				$\Uparrow$				

is compressed to

	-2	2	6	
	-1	3	7	
	0	4	8	
	1	5	9	
$\cdots$	2	6	10	$\cdots$
	3	7	11	
	4	8	12	
	5	9	13	
	6	10	14	
	7	11	15	
		$\Uparrow$		

Figure 3.2: Compressing a Worktape for Constant Time Reduction

If M begins the next c steps by reading tape cell i, then N's head is reading the tape cell that has i in its middle c tracks. In one step of N we can simulate the next c moves of M before having to move N's heads. After these c steps of M, N repositions its head if necessary, carrying in its finite control the updated values of the $2c-2$ overlapping cells of M. Any inconsistent values in the opposite direction on this tape will be corrected when N's head passes through the cell in that direction.

N starts the simulation by compressing the input tape of M onto an extra worktape and "rewinding" that tape (i.e., moving its head to the left end). It takes n steps to compress the input tape, n/c to rewind, and $T(n)/c$ to simulate the $T(n)$ steps of M. Hence, the time used by N is at most

$$n + \frac{n}{c} + \frac{T(n)}{c} \leq n + \frac{2T(n)}{c} \leq n + \epsilon T(n).$$

$\square$

Theorem 3.2 shows how to speed up deterministic and nondeterministic time by a constant factor. The same simulation does not work for alternating time, since the machine being simulated may alternate several times during the next c steps, whereas the simulating machine has only 1 corresponding step, which must be either existential or universal. Nonetheless, a different technique does work for alternating time:

Theorem 3.3: For every $\epsilon > 0$ and every time bound $T(n) \geq \log_2 n$,

$$\mathrm{ATIME}(T(n)) \subseteq \mathrm{ATIME}\left(\lceil \log_2 n \rceil + \lceil \epsilon T(n) \rceil\right).$$

Proof: The proof is left as an exercise. (Hint: simulate c steps by 2 steps rather than 1.) □

3.2. Hierarchy Theorems

The original motivation behind Theorems 3.1, 3.2, and 3.3 remains: is it possible to enhance the power of a Turing machine by allowing it more time or more space? In some sense the constant factor speedups are cheats, since there is a corresponding blowup in the alphabet size and transition function. They are interesting, however, because they tell us not to look for added power within a constant factor increase in time or space. In this section we will consider what happens when we increase the time or space by more than a constant factor.

3.2.1. The Deterministic Space Hierarchy Theorem

Definition 3.4: $S(n)$ is *(deterministic) space constructible* if and only if there is a deterministic Turing machine M with one worktape such that, on all inputs x, M halts after visiting (and marking) exactly $S(|x|)$ cells on its worktape.

Example 3.5: Any common function $S(n)$ that you will encounter is space constructible. For example, the functions $\lceil \log_2 n \rceil$, n, n^2, 2^n, and $n!$ are all space constructible. If $S(n)$ and $T(n)$ are space constructible, then so are $S(n) + T(n)$, $S(n)T(n)$, and $(S(n))^{T(n)}$. The proofs of these facts are left as exercises.

Example 3.6: The function $\lceil \log_2 \log_2 \log_2 n \rceil$ is not space constructible.

Example 3.7: The function $S(n)$ given below is not space constructible, since it is not even computable.

$$S(n) = \begin{cases} n & \text{if } n \text{ is the binary encoding of a deterministic Turing machine that always halts} \\ 2n & \text{otherwise} \end{cases}.$$

Theorem 3.8 is the deterministic space hierarchy theorem.

Theorem 3.8 (Hartmanis, Lewis, and Stearns [14]): For any space constructible function $S(n) = \Omega(\log n)$, and any $s(n) = o(S(n))$, [1]

$$\text{DSPACE}(S(n)) - \text{DSPACE}(s(n)) \neq \emptyset.$$

Proof: By Theorem 3.1, we can assume without loss of generality that $S(n) \geq 2\log_2 n + 2$ for all sufficiently large n. We will construct a deterministic Turing machine M that runs in space $O(S(n))$ and disagrees on at least one input with every possible deterministic Turing machine that runs in space $s(n) = o(S(n))$.

Construct M as follows. On input $x = 0^i 1 j$, where $i \geq 0$ and $j \in \{0,1\}^*$, with $|x| = n$,

1. M marks off $S(n)$ cells on each of two worktapes. This can be done in space $S(n)$, by Definition 3.4.
2. M treats j as the binary encoding of some deterministic Turing machine T_j and, except as modified below, simulates T_j on input x, recording successive binary encodings of T_j's configurations within the marked cells of one of its worktapes. (If j does not encode a deterministic Turing machine, then M rejects x.)
   (a) If T_j halts in a final configuration, then M halts in a nonfinal configuration.
   (b) If T_j halts in a nonfinal configuration, then M halts in a final configuration.
   (c) If the encoding of one of T_j's configuration runs over the $S(n)$ marked cells, then M accepts.
   (d) If T_j runs for more than $2^{S(n)}$ steps, as timed on the second marked-off tape, then M accepts.

By construction, M uses space $O(S(n))$.

All that remains to show is that, for any deterministic Turing machine T running in space $s(n) = o(S(n))$, $L(T) \neq L(M)$.

Let $T = T_j$. There is a constant c_j such that any configuration of T_j on any accepted input can be encoded in binary using $c_j s(n) + \lceil \log_2 n \rceil$ bits, where c_j depends only on the size of T_j's worktape alphabet, the number of tapes in T_j, and the number of states in T_j, and the $\lceil \log_2 n \rceil$ bits encode the input head position. Note that T_j has at most $2^{c_j s(n) + \lceil \log_2 n \rceil}$ distinct configurations.

Since $S(n) \geq 2\log_2 n + 2$ and $s(n) = o(S(n))$,

$$\lim_{n\to\infty} \frac{s(n)}{S(n) - \lceil \log_2 n \rceil} \leq \lim_{n\to\infty} \frac{s(n)}{S(n)/2} = 0 < \frac{1}{c_j}.$$

Thus, for sufficiently large n,

$$\frac{s(n)}{S(n) - \lceil \log_2 n \rceil} \leq \frac{1}{c_j},$$

[1]Actually, $s(n) = o(S(n))$ can be replaced by the weaker condition

$$\lim_{n\to\infty} \inf \frac{s(n)}{S(n)} = 0,$$

but in practice the condition given always suffices.

which implies that, for sufficiently large n,

$$c_j s(n) + \lceil \log_2 n \rceil \leq S(n). \tag{3.1}$$

Choose any $n > |j|$ satisfying Inequality (3.1), and let $x = 0^{n-|j|-1}1j$. Then M accepts x if and only if T_j does not accept x: If T_j accepts or rejects x while M uses space at most $S(n)$ and T_j uses time at most $2^{S(n)}$ (cases 2a and 2b), then M halts with the opposite answer. If M uses more space than $S(n)$ on x then, by Inequality (3.1), T_j uses more space than $s(n)$ and hence cannot accept x, whereas by case 2c, M accepts x. Finally, if T_j runs for time greater than $2^{S(n)} \geq 2^{c_j s(n) + \lceil \log_2 n \rceil}$, then T_j must have repeated some configuration and hence cannot accept x, whereas by case 2d, M accepts x. □

As a consequence of Theorem 3.8, there are arbitrarily complex languages among the recursively enumerable languages.

3.2.2. The Deterministic Time Hierarchy Theorem

Because of the need to diagonalize over all multitape machines with some fixed number of tapes, the deterministic time hierarchy theorem is not quite as tight as the deterministic space hierarchy theorem. In Lemma 4.8 we will show how to simulate a multitape deterministic Turing machine running in time $T(n)$ by a one-tape deterministic Turing machine running in time $O((T(n))^2)$. To get the tightest known deterministic time hierarchy theorem, we will need a better time bound, which is achieved at the expense of extra worktapes:

Lemma 3.9 (Hennie and Stearns [17]): For any k, if L is a language accepted by a k-worktape deterministic Turing machine in time $T(n)$, then L is accepted by a 2-worktape deterministic Turing machine in time $O(T(n) \log T(n))$.

Open Problem 3.10: Simulate k tapes by any constant number of tapes with a blowup in time that is $o(\log T(n))$.

Definition 3.11: $T(n)$ is *(deterministic) time constructible* if and only if there is a deterministic Turing machine that, on all inputs x, runs for exactly $T(|x|)$ steps and then halts.

Theorem 3.12 is the deterministic time hierarchy theorem.

Theorem 3.12 (Hartmanis and Stearns [15]): For any function $t(n) \geq n$, and any time constructible function $T(n) = \omega(t(n) \log t(n))$, [2]

$$\mathrm{DTIME}(T(n)) - \mathrm{DTIME}(t(n)) \neq \emptyset.$$

[2] As in Theorem 3.8, $T(n) = \omega(t(n) \log t(n))$ can be replaced by the weaker condition

$$\lim_{n \to \infty} \inf \frac{t(n) \log t(n)}{T(n)} = 0.$$

Proof: The proof is similar to that of Theorem 3.8, but uses Lemma 3.9 in order to simulate T_j (which may have arbitrarily many worktapes) by M (which has a fixed number of worktapes). □

A typical consequence of the deterministic time hierarchy theorem is that DTIME$(n) \subset$ DTIME(n^2). This is because $n^2 = \omega(n \log n)$. However, that theorem is not tight enough to have as an immediate consequence that DTIME$(n) \subset$ DTIME$(n \log n)$.

Exercise 3.13: Consider what might further complicate the hierarchy theorems for nondeterministic Turing machines. Hint: It is not a problem for alternating Turing machines.

3.3. Exercises

1. Complete the details of Theorem 3.1.

2. Prove Theorem 3.3.

   (Hint: Convert the alternating Turing machine into one in which at most one input symbol is read on any path of the computation tree.)

3. Prove that, if $S(n)$ and $T(n)$ are space constructible, then so are $S(n)+T(n)$, $S(n)T(n)$, and $(S(n))^{T(n)}$.

4. Prove Theorem 3.12. You may assume the result in Lemma 3.9.

5. Do Exercise 3.13.

Chapter 4

Basic Relationships Among the Models and Measures

This section begins a study of the most fundamental relationships among time and space, and determinism, nondeterminism, and alternation.

Theorem 4.1: For any $T(n) \geq \log_2 n$,

$$\begin{aligned}&\mathrm{DTIME}(T(n)) \subseteq \mathrm{NTIME}(T(n)) \subseteq \mathrm{ATIME}(T(n)) \subseteq \\ &\mathrm{DSPACE}(T(n)) \subseteq \mathrm{NSPACE}(T(n)) \subseteq \mathrm{ASPACE}(T(n)) = \bigcup_{c>1} \mathrm{DTIME}\left(c^{T(n)}\right).\end{aligned}$$

Proof: The four containments $\mathrm{DTIME}(T(n)) \subseteq \mathrm{NTIME}(T(n)) \subseteq \mathrm{ATIME}(T(n))$ and $\mathrm{DSPACE}(T(n)) \subseteq \mathrm{NSPACE}(T(n)) \subseteq \mathrm{ASPACE}(T(n))$ are immediate from the facts that a deterministic Turing machine is a special case of a nondeterministic Turing machine, which in turn is a special case of an alternating Turing machine. The remaining three containments will be proved in Theorems 4.7, 4.9, and 4.11. □

Theorem 4.2: For any time constructible function $T(n)$, at least one of the containments in Theorem 4.1 is proper.

Proof: For $T(n) < n$, Example 2.16 and the remarks following it show that $\mathrm{DTIME}(T(n)) \subset \mathrm{NTIME}(T(n))$. For $T(n) \geq n$, Theorem 3.12 shows that $\mathrm{DTIME}(T(n)) \subset \mathrm{DTIME}\left(2^{T(n)}\right)$. □

Open Problem 4.3: Prove that any one of the containments in Theorem 4.1 is proper. Example 2.16 and the remarks following it show that $\mathrm{DTIME}(T(n)) \neq \mathrm{NTIME}(T(n)) \neq \mathrm{ATIME}(T(n))$ for all $T(n) < n$. Paul, Pippenger, Szemerédi, and Trotter [34] have shown that $\mathrm{DTIME}(T(n)) \neq \mathrm{NTIME}(T(n))$ for any $T(n) = O(n)$ (and also very slightly faster growing functions $T(n)$). Other than these few values, the problem is open.

4.1. Time vs. Space on a Fixed Model

We provide some "warmups" for the simulations remaining from Theorem 4.1 by proving the same relationships between time and space, but keeping the model fixed.

Proposition 4.4: DTIME$(T(n)) \subseteq$ DSPACE$(T(n))$, and similarly for nondeterministic and alternating Turing machines. In fact, any Turing machine that runs in time $T(n)$ runs in space $O(T(n))$ itself.

Proof: Any Turing machine running in time $T(n)$ can visit at most $T(n)$ different cells on each worktape, and so itself runs in space $O(T(n))$. The result then follows from Theorem 3.1. $\Box$

Proposition 4.5: For any $S(n) \geq \log_2 n$, DSPACE$(S(n)) \subseteq \bigcup_{c>1}$ DTIME$\left(c^{S(n)}\right)$. In fact, any deterministic Turing machine that runs in space $S(n)$ runs in time $2^{O(S(n))}$ itself.

Proof: A deterministic Turing machine with space $S(n)$ has only $2^{O(S(n))}$ distinct configurations, provided $S(n) \geq \log_2 n$. Thus, if it has not halted within $2^{O(S(n))}$ steps, it must be in an infinite loop. Therefore any deterministic Turing machine that accepts will do so itself within $2^{O(S(n))}$ steps. $\Box$

Proposition 4.6: For any $S(n) \geq \log_2 n$, NSPACE$(S(n)) \subseteq \bigcup_{c>1}$ NTIME$\left(c^{S(n)}\right)$, and similarly for alternating Turing machines. In fact, any Turing machine that runs in space $S(n)$ runs in time $2^{O(S(n))}$ itself.

Proof: The argument is similar to the previous one although, due to the nondeterminism, there is no guarantee that a machine that repeats a configuration will never accept. However, it is easy to see that if an alternating Turing machine accepts, there exists an accepting subtree in which no configuration is repeated along any path. $\Box$

4.2. ASPACE$(S(n)) \subseteq \bigcup_{c>1}$ DTIME$\left(c^{S(n)}\right)$

In this section we prove the first of the containments left unproved in Theorem 4.1.

Theorem 4.7 (Chandra, Kozen, and Stockmeyer [3]): For any $S(n) \geq \log_2 n$,

$$\text{ASPACE}(S(n)) \subseteq \bigcup_{c>1} \text{DTIME}\left(c^{S(n)}\right).$$

Proof: Let A be an alternating Turing machine that runs in space $S(n)$. We will construct a deterministic Turing machine D that simulates A. The most naive approach is for D to traverse some accepting subtree of A. This tree, however, may have height $2^{O(S(n))}$ and size $2^{2^{O(S(n))}}$, which make its traversal impossible within $2^{O(S(n))}$ steps.

Notice that, although the accepting subtree may be double exponential in size, it has only a single exponential number of distinct configurations. This suggests that the idea of traversing an accepting subtree may still be sound, but it needs a more sophisticated implementation. The idea is to identify all the identical subtrees, yielding a directed graph of size $2^{O(S(n))}$.

CONSTRUCTION: Assume for the moment that $S(n)$ can be computed by a deterministic Turing machine in time $2^{O(S(n))}$. On input x, D constructs a directed graph $G = (V, E)$ such that V is the set of space $S(n)$ configurations of A on input x, and $(P, Q) \in E$ if and only if $P \underset{A,x}{\vdash} Q$. D records the graph on a worktape by simply listing the pairs in E. Now D runs the following algorithm:

comment: label all accepting configurations, backwards from the final to the initial configuration;
label all final configurations 0;
for $t = 1, 2, \ldots$ **repeat**
  for all unlabeled vertices P **do**
    begin
      if P is existential **and** there is an immediate successor of P that is labeled $t - 1$
        then label P with t;
      if P is universal **and** all immediate successors of P are labeled $t - 1$ or less
        then label P with t
    end
until no vertices are labeled t;
if initial configuration is labeled
  then accept
  else reject.

Since $S(n)$ may not be constructible by a deterministic Turing machine in time $2^{O(S(n))}$, D instead runs the simulation above for $S = 1, 2, 3, \ldots$, accepting if and only if the procedure above accepts for one of these values.

CORRECTNESS: By an induction on t given below, P is labeled t if and only if the minimum height of any accepting P-subtree is t. The correctness then follows from Theorem 1.20, since the initial configuration is given some (finite) label if and only if there is an accepting subtree of A on x (of finite height).

BASIS ($t = 0$): P is labeled 0 if and only if P is final, which occurs if and only if the minimum height of any accepting P-subtree is 0.

INDUCTION ($t > 0$):

Case 1: P is existential. Then P is labeled t if and only if some immediate successor Q of P is labeled $t-1$, and no immediate successor R of P has a lesser label. By the induction hypothesis, this occurs if and only if the minimum height of any accepting Q-subtree is $t - 1$, and for no immediate successor R of P is there a shallower accepting R-subtree. By the definition of accepting subtree, this occurs if and only if the minimum height of any accepting P-subtree is t.

Case 2: P is universal. Then P is labeled t if and only if all immediate successors Q of P are labeled $t - 1$ or less. By the induction hypothesis, this occurs if and only if, for all immediate successors Q of P, the minimum height of any accepting Q-subtree is at most $t-1$. By the definition of accepting subtree, this occurs if and only if the minimum height of any accepting P-subtree is t.

ANALYSIS: Assume for the moment that $S(n)$ is computable in time $2^{O(S(n))}$. G has $2^{O(S(n))}$ vertices and edges (since $S(n) \geq \log n$), and can be constructed in time $2^{O(S(n))}$. (To do so, notice that D must find the input symbol indexed by A, in order to determine if $(P, Q) \in E$.) Finding immediate successors of P in the edge list takes time $2^{O(S(n))}$. There are $2^{O(S(n))}$ iterations of the inner loop. There are also $2^{O(S(n))}$ iterations of the outer loop, since at least one vertex is labeled in each iteration. The total running time is thus $2^{O(S(n))} + (2^{O(S(n))})^3 = 2^{O(S(n))}$.

Running this procedure for $S = 1, 2, 3, \ldots, S(n)$ multiplies the running time by at most a constant. (Note that this may cause D to run forever if $x \notin L(A)$, but that is no problem, since the definitions only require D to accept and run in time $2^{O(S(n))}$ for $x \in L(A)$.)

□

4.3. DTIME($T(n)$) $\subseteq$ ASPACE($\log T(n)$)

4.3.1. Simulating Multiple Tapes by a Single Tape

In order to simulate time-bounded deterministic Turing machines by space-bounded alternating Turing machines, it will be convenient to make the simplifying assumption that the deterministic Turing machine is a 1-tape machine, rather than a k-tape machine. Thus, we must show that a k-tape machine can be simulated by a 1-tape machine without excessive time penalty.

Lemma 4.8: If L is accepted by a k-tape deterministic Turing machine M in time $T(n)$, then L is accepted by a 1-tape deterministic Turing machine N in time $O((T(n))^2)$. (N's single tape is, of course, a read/write tape.) Moreover, the position of N's single tape head is a function of time alone, and is independent of N's particular input.

Proof:

CONSTRUCTION: N has $2k+1$ tracks on its tape, with the contents of M's ith tape on track $2i-1$, and a mark on track $2i$ indicating the position of M's ith head. On track 0 N keeps a left and right endmarker, which initially coincide at the position of the tape head. (As usual, the "tracks" are just a conceptual device for talking about an expanded worktape alphabet.)

N stores M's state in its finite control. As an invariant of the simulation, N begins simulating each step of M with its single tape head at the left endmarker on track 0, and with all k head marks between the two endmarkers. In one pass to the right endmarker, N collects in its finite control the k symbols under M's tape heads. N now has in its finite control everything it needs to compute M's next move. It updates M's state in its finite control, and updates its tape as follows. In a return pass to the left endmarker, it rewrites the k tape cells under M's heads. In a pass to the right endmarker, it moves all the marks whose corresponding heads move right, and moves the right endmarker one cell right. In a final pass to the left endmarker, it moves all the marks whose corresponding heads move left, and moves the left endmarker one cell left. Notice that all k marks remain between the two endmarkers.

ANALYSIS: After simulating t steps of M, the nonblank portion of N's tape has length $2t+1$. Since N makes 4 passes over this to simulate a step of M, the total time is at most

$$\sum_{t=1}^{T(n)} 8t = O((T(n))^2).$$

□

Although it depends on Lemma 4.8, the simulation of deterministic time by alternating space in the next section does not use the fact that the 1-tape machine's head movement is independent of the particular input. This fact, called "obliviousness", will be useful later in Section 7.5 when we discuss the simulation of deterministic Turing machines by circuits. *Obliviousness used there?*

There are languages (for instance, the language L of Example 2.12) that can be accepted by 2-tape deterministic Turing machines in $O(n)$ time, but require $\Omega(n^2)$ time on any 1-tape deterministic Turing machine (Hennie [16]), so Lemma 4.8 is optimal to within a constant factor.

4.3.2. Simulating Deterministic Time by Alternating Space

The main result of this section is the converse of Theorem 4.7.

Theorem 4.9 (Chandra, Kozen, and Stockmeyer [3]): For any $T(n) \geq n$,

$$\mathrm{DTIME}(T(n)) \subseteq \mathrm{ASPACE}(\log T(N)).$$

Proof:

CONSTRUCTION: Let M be a deterministic Turing machine that runs in time $T(n)$. Using Lemma 4.8, there is a 1-tape deterministic Turing machine $N = (Q, Q-F, \emptyset, F, \Gamma, \Sigma, q_0, \delta)$ that runs in time $O((T(n))^2)$ and accepts the same language. Assume without loss of generality that N moves its head at every step.

Let C_0, C_1, ..., C_t be an "accepting computation" of N on input x; that is, C_0 is the initial configuration of N on x, C_t is a final configuration, and $C_i \underset{N,x}{\vdash} C_{i+1}$ for all $0 \leq i < t$. Note that $t = O((T(n))^2)$, where as usual $n = |x|$. Let $C_{i,j}$ be the jth symbol of C_i, or $\not b$ if j is too large or too small, where indexing is relative to the position $j = 1$ of the first input symbol in C_0. $C_{i+1,j}$ depends only on $C_{i,j-1}$, $C_{i,j}$, $C_{i,j+1}$, and $C_{i,j+2}$. Specifically, $C_{i+1,j} = \mathit{local}(C_{i,j-1}, C_{i,j}, C_{i,j+1}, C_{i,j+2})$, where

$$\mathit{local}(b_{-1}, b_0, b_1, b_2) = \begin{cases} b_0, & \text{if } b_{-1}, b_0, b_1 \notin Q \\ q, & \text{if } b_{-1} \in Q \text{ and } \delta(b_{-1}, b_0) = \{(q, a, R)\} \\ a, & \text{if } b_{-1} \in Q \text{ and } \delta(b_{-1}, b_0) = \{(q, a, L)\} \\ a, & \text{if } b_0 \in Q \text{ and } \delta(b_0, b_1) = \{(q, a, R)\} \\ b_{-1}, & \text{if } b_0 \in Q \text{ and } \delta(b_0, b_1) = \{(q, a, L)\} \\ b_0, & \text{if } b_1 \in Q \text{ and } \delta(b_1, b_2) = \{(q, a, R)\} \\ q, & \text{if } b_1 \in Q \text{ and } \delta(b_1, b_2) = \{(q, a, L)\} \\ \perp, & \text{otherwise} \end{cases}$$

where $\perp \notin Q \cup \Gamma$ is a special "undefined" symbol.

Given $x = x_1 x_2 \cdots x_n$, the alternating Turing machine A does the following:

existentially choose t;
  comment: running time;
existentially choose j with $-t \leq j \leq t$;
  comment: final head position;
existentially choose $q \in F$;

$check(t, j, q)$;

where *check* is defined as follows:

procedure $check(i, j, b)$
comment: accepts if and only if $C_{i,j} = b$;
if $i = 0$
  then if $(j = 0$ **and** $b = q_0)$ **or** $(1 \leq j \leq n$ **and** $b = x_j)$ **or** $(((j < 0)$ **or** $(j > n))$ **and** $b = \not{b})$
    then accept
    else reject
  else begin
    existentially choose $b_{-1}, b_0, b_1, b_2 \in Q \cup \Gamma$;
    if $b \neq local(b_{-1}, b_0, b_1, b_2)$ **then reject**;
    universally choose $\Delta \in \{-1, 0, 1, 2\}$;
    $check(i - 1, j + \Delta, b_\Delta)$;
  end
end .

CORRECTNESS: By induction on i, the configuration A is in when it calls $check(i, j, b)$ is accepting if and only if $C_{i,j} = b$, where $C_0, C_1, \ldots$ is the computation of N on input x. Details are left as an exercise.

ANALYSIS: A needs space to store i, j, and n, plus constant space for b, b_{-1}, b_0, b_1, b_2, and Δ. Note that the recursive call to *check* doesn't need storage for a stack, since the call is tail-recursive: we can reuse the space from i, j, and b to store $i - 1$, $j + \Delta$, and b_Δ. Since $0 \leq i \leq t$ and $-2t \leq j \leq 3t$, the total space is

$$O(\log t + \log n) = O(\log((T(n))^2) + \log n) = O(\log T(n) + \log n) = O(\log T(n))$$

since, by assumption, $T(n) \geq n$.

□

Example 4.10: One particularly important example of Theorems 4.7 and 4.9 is that

$$\mathcal{P} = \text{ASPACE}(\log n),$$

where $\mathcal{P}$ is defined to be $\bigcup_{c>0} \text{DTIME}(n^c)$, that is, the class of languages accepted in deterministic polynomial time. It is noteworthy that this important time-bounded complexity class can be characterized by a space-bounded complexity class, particularly with such a small space bound. This fact will be exploited frequently in Chapter 7.

4.4. ATIME$(T(n)) \subseteq$ DSPACE$(T(n))$

Theorem 4.11 is the last containment remaining to complete the proof of Theorem 4.1.

Theorem 4.11 (Chandra, Kozen, and Stockmeyer [3]): For any $T(n)$,

$$\text{ATIME}(T(n)) \subseteq \text{DSPACE}(T(n)).$$

Lower bound $T(n) \geq \log n$ purposely omitted: if A only writes small indices on its index tape, D doesn't need much space to count up to them.

Proof:

CONSTRUCTION:

Let A be an alternating Turing machine that runs in time $T(n)$. By Proposition 4.4, A also runs in space $T(n)$, including the space A uses on its index tape. Construct a deterministic Turing machine D that, given input x, traverses the computation tree of A on x as follows. Assume for the moment that $T(n)$ is computable by a deterministic Turing machine in space $T(n)$. D will accept x if and only if *accepting*$(P_0, T(n))$ returns true, where P_0 is the initial configuration of A on x, and *accepting* is the function given in Figure 4.1.

```
function accepting(P, t) returns boolean
comment: returns true if and only if A on input x has an accepting P-subtree of height at most
  t ;
begin
   if P is a final configuration then return true ;
   if t = 0 then return false ;
   if P is existential
      then begin
         b ← false ;
         for all Q such that P ⊢_{A,x} Q do
            b ← (b or accepting(Q, t − 1)) ;
         return b ;
      end
      else begin comment: P is universal ;
         b ← true ;
         for all Q such that P ⊢_{A,x} Q do
            b ← (b and accepting(Q, t − 1)) ;
         return b ;
      end
end .
```

Figure 4.1: The Function *accepting*

Since $T(n)$ may not be computable by D in space $T(n)$, D instead runs *accepting*(P_0, T) for $T = 1, 2, 3, \ldots$, halting and accepting if and only if one of these invocations returns true. (Note once again that this may cause D to run forever and/or use too much space if $x \notin L(A)$, but that is no problem, since the definitions only require D to accept and run in space $T(n)$ for $x \in L(A)$.)

CORRECTNESS: By induction on t, *accepting*(P, t) returns true if and only if A on input x has an accepting P-subtree of height at most t. The details are left as an exercise.

ANALYSIS: Assume for the moment that $T(n)$ is computable by D in space $T(n)$. D can record any configuration P of A in space $T(n)$ since A runs in space $T(n)$, including the space used on A's index tape. If not for the recursive calls, D would certainly run in space $T(n)$, because D never needs more than two configurations P and Q at any time. (Note that D needs space $T(n)$ for a counter to run up to the contents of A's index tape. It must do so in order to check whether the indexed input symbol supports the transition $P \vdash_{A,x} Q$.)

Unfortunately, D needs a stack of height $T(n)$ to keep track of the recursive calls in progress. To avoid having $T(n)$ bits per stack entry (which would be needed to store the entire configuration at each recursive call), it suffices to store on the stack which of the constant number of transitions in A's transition function was used to generate Q from P. When an element is popped from the stack, this information is sufficient to reconstruct P from Q, and to find the next value of Q.

The space used for trying $T = 1, 2, 3, \ldots, T(n)$ is at most $T(n)$. □

4.5. Savitch's Theorem

With the constant factor speedup theorems (Theorems 3.1, 3.2, and 3.3) and the hierarchy theorems (Theorems 3.8 and 3.12), we have completed our investigation of more versus less of a single resource on a single model. We now return to the question of relationships among the different models and measures.

Recall Theorem 4.1: For any $T(n) \geq \log_2 n$,

$$\begin{aligned}&\mathrm{DTIME}(T(n)) \subseteq \mathrm{NTIME}(T(n)) \subseteq \mathrm{ATIME}(T(n)) \subseteq \\ &\mathrm{DSPACE}(T(n)) \subseteq \mathrm{NSPACE}(T(n)) \subseteq \mathrm{ASPACE}(T(n)) = \bigcup_{c>1} \mathrm{DTIME}\left(c^{T(n)}\right).\end{aligned}$$

If we were to try to simulate, say, nondeterministic space by deterministic space, we could go through the appropriate six containments to arrive at the exponential blowup

$$\mathrm{NSPACE}(S(n)) \subseteq \bigcup_{c>1} \mathrm{DSPACE}\left(c^{S(n)}\right). \tag{4.1}$$

Perhaps, though, this exponential blowup is an artifact of going through alternating space and deterministic time: the speculations that nondeterministic space is presumably so much weaker than alternating space, and deterministic space presumably so much stronger than deterministic time might lead one to conjecture that Containment (4.1) can be improved.

On the other hand, from a naive point of view Containment (4.1) appears to be optimal, because the deterministic Turing machine "needs" to traverse the nondeterministic Turing machine's entire computation tree, which may have size $2^{2^{\Omega(S(n))}}$, so just recording the name of the node the deterministic Turing machine is working on requires space $2^{\Omega(S(n))}$.

Such was the prevailing view from the mid-1960's until 1970, when Savitch proved that Containment (4.1) could be improved dramatically.

Definition 4.12: If M is an alternating Turing machine with input x and configurations P and Q, and d is an integer, we say $P \vdash_{M,x}^{d} Q$ if and only if there exist configurations $P = P_0, P_1, P_2, \ldots, P_d = Q$, such that $P_i \vdash_{M,x} P_{i+1}$ for all $0 \leq i < d$.

Definition 4.13: We say $P \vdash_{M,x}^{\leq d} Q$ if and only if $P \vdash_{M,x}^{c} Q$ for some $c \leq d$.

Theorem 4.14 (Savitch [40], Chandra, Kozen, and Stockmeyer [3]): For any $S(n) \geq \log_2 n$,

$$\mathrm{NSPACE}(S(n)) \subseteq \mathrm{ATIME}((S(n))^2).$$

Proof:

Construction: Let N be a nondeterministic Turing machine that runs in space $S(n)$. Construct an alternating Turing machine A that, on input x, simulates N on x. A's strategy is divide-and-conquer on an accepting computation of N on x. More specifically, A executes the following:

existentially choose $S(n)$;
let P_0 be the initial configuration of N on x;
existentially choose P_f, a final configuration of N;
existentially choose k;
reach(P_0, P_f, k);

where *reach* is the following procedure:

procedure *reach*(P, Q, k)
comment: accepts if and only if $P \vdash^{\leq 2^k}_{N,x} Q$;
if $k = 0$
  then if $(P = Q$ **or** $P \vdash_{N,x} Q)$ **then accept else reject**
  else begin
    existentially choose R;
    universally choose $b \in \{0, 1\}$;
    case b of
      0: *reach*$(P, R, k-1)$;
      1: *reach*$(R, Q, k-1)$
    end
  end .

Correctness: By induction on k, we will prove that the configuration A is in when it invokes *reach*(P, Q, k) is accepting if and only if $P \vdash^{\leq 2^k}_{N,x} Q$.

Basis ($k = 0$): When $k = 0$, A accepts if and only if $P = Q$ or $P \vdash_{N,x} Q$, which in turn is true if and only if $P \vdash^{\leq 2^k}_{N,x} Q$.

INDUCTION ($k > 0$):

"if" clause: Assume $P \vdash_{N,x}^{\leq 2^k} Q$. Let R be a "midpoint" configuration in this computation, that is, $P \vdash_{N,x}^{\leq 2^{k-1}} R$ and $R \vdash_{N,x}^{\leq 2^{k-1}} Q$. By the induction hypothesis, the configurations corresponding to $reach(P, R, k-1)$ and $reach(R, Q, k-1)$ are both accepting configurations. Then in $reach(P, Q, k)$, there exists an R such that both recursive calls will accept. That is, the configuration corresponding to $reach(P, Q, k)$ is accepting.

"only if" clause: Assume the configuration that A is in when it invokes $reach(P, Q, k)$ is accepting. Then there exists an R such that $reach(P, R, k-1)$ and $reach(R, Q, k-1)$ each correspond to accepting configurations. By the induction hypothesis, $P \vdash_{N,x}^{\leq 2^{k-1}} R$ and $R \vdash_{N,x}^{\leq 2^{k-1}} Q$. Thus, $P \vdash_{N,x}^{\leq 2^k} Q$.

ANALYSIS: If N accepts x, then each configuration P of N on x has $|P| = O(S(n))$, since $S(n) \geq \log_2 n$. The time N takes on x is $2^{O(S(n))}$, by Proposition 4.6.

Constructing P_0 and guessing P_f take time $O(S(n))$. The guessed time k need only satisfy $2^k = 2^{\Theta(S(n))}$, so the bits of k can be guessed in time $O(\log S(n))$.

For the case $k = 0$, checking whether $P = Q$ or $P \vdash_{N,x} Q$ can be done deterministically in $O(S(n))$ time. (Note that $S(n) \geq \log_2 n$ is needed here to copy N's input head position from P to the index tape.)

For the case $k \neq 0$, each level of recursion uses $O(S(n))$ time to guess R. Since the depth of recursion is $k = O(S(n))$, the total time is $O((S(n))^2)$. (Note that the two recursive calls are done in parallel, rather than sequentially.) □

Corollary 4.15 (Savitch [40]): For any $S(n) \geq \log_2 n$,

$$\text{NSPACE}(S(n)) \subseteq \text{DSPACE}((S(n))^2).$$

Savitch's theorem was subsequently generalized to include sublogarithmic space bounds:

Theorem 4.16 (Monien and Sudborough [31], Tompa [46]): For any $S(n)$,

$$\text{NSPACE}(S(n)) \subseteq \text{ATIME}(S(n)(S(n) + \log n)).$$

4.6. Other Containments Among the Complexity Classes

Theorem 4.17 (Dymond and Tompa [9]): For any $T(n) \geq n$,

$$\text{DTIME}(T(n)) \subseteq \text{ATIME}(T(n)/\log T(n)).$$

Corollary 4.18 (Hopcroft, Paul, and Valiant [18]): For any $T(n) \geq n$,

$$\text{DTIME}(T(n)) \subseteq \text{DSPACE}(T(n)/\log T(n)).$$

Open Problem 4.19: Find other relationships among the six complexity classes of Theorem 4.1. As possible examples,

- NTIME($T(n)$) $\subseteq$ NSPACE($T(n)/\log T(n)$)?
- ATIME($T(n)$) = DSPACE($T(n)$)?

  We know ATIME($T(n)$) $\subseteq$ DSPACE($T(n)$) $\subseteq$ ATIME($(T(n))^2$), from Theorems 4.11 and 4.14.
- DSPACE($T(n)$) = NSPACE($T(n)$)?

  We know DSPACE($T(n)$) $\subseteq$ NSPACE($T(n)$) $\subseteq$ DSPACE($(T(n))^2$), from Corollary 4.15.

4.7. Closure Under Complementation

We start with the definition of closure under complementation.

Definition 4.20: Let $\mathcal{C}$ be a set of languages over some alphabet Σ (i.e., $\mathcal{C} \subseteq 2^{\Sigma^*}$). Then $\mathcal{C}$ is *closed under complementation* if and only if for every language $L \in \mathcal{C}$, it is also the case that $\overline{L} \in \mathcal{C}$, where $\overline{L} = \Sigma^* - L$ is the complement of L.

Proposition 4.21: If $S(n) \geq \log_2 n$ is space constructible, then DSPACE($S(n)$) is closed under complementation.

Proof: This is very similar to part of the proof of Theorem 3.8. Final and nonfinal states are interchanged, and a counter is used so that the complementing machine can accept if the original one runs forever. □

Exercise 4.22: Prove that Proposition 4.21 is false for arbitrary nonconstructible bounds $S(n)$.

Similar propositions hold for DTIME, ATIME, and ASPACE. In the case of the alternating machines, existential and universal states are interchanged and deMorgan's laws applied.

The nondeterministic complexity classes are conspicuously missing from this list. The problem is that we cannot just change existential to universal states, as is done for alternating machines. More generally, the difficulty is that the complementing machine is to accept if and only if *all* paths in the original machine's computation tree are rejecting, which seems like a problem that cannot be solved with only existential choice.

In 1964, Kuroda [26] posed the "LBA (linear bounded automaton) question": is NSPACE(n) closed under complementation? He was interested in this because the class of context-sensitive languages is exactly NSPACE(n) and, with the exception of the context-sensitive languages, the question of closure under complementation had been settled for all the classes of the "Chomsky hierarchy" (i.e., for the regular languages, the context-free languages, and the recursively enumerable languages). The knowledgeable experts agreed that the answer to the LBA question was surely "no", for the reasons described above. The question remained open until 1987, when Immerman [19] and Szelepcsényi [45] independently and simultaneously announced the following surprising result:

Theorem 4.23 (Immerman [19], Szelepcsényi [45]): If $S(n) \geq \log_2 n$ is space constructible, then NSPACE($S(n)$) is closed under complementation.

Proof:

CONSTRUCTION: Let $S(n)$ be space constructible, and M be a nondeterministic Turing machine running in space $S(n)$ with input x and initial configuration P_0. M, $S(n)$, x, and P_0 will all be global variables to the subroutines defined below.

We start by defining two simple nondeterministic subroutines *guess* and *continue_iff_reachable*.

function *guess* **returns** boolean
begin
  existentially choose $b \in \{true, false\}$;
  return b
end .

The nondeterministic subroutine *continue_iff_reachable*(k, P) simulates M on input x starting at P_0 for k steps, aborting (i.e., halting in a nonfinal configuration) if and only if it has not reached configuration P within those k steps. Thus, *continue_iff_reachable* is a filter that allows the algorithm to continue if and only if $P_0 \vdash_{M,x}^{\leq k} P$. Here are the details:

procedure *continue_iff_reachable*(k, P)
comment: does not abort if and only if $P_0 \vdash_{M,x}^{\leq k} P$;
begin
  $Q \leftarrow P_0$;
  do $k+1$ **times**
    if $Q = P$
      then return
      else existentially choose Q **from** $\{R \mid Q \vdash_{M,x} R\}$;
  reject
end .

The central subroutine is *count*, which is used to count the number of configurations of M that are reachable from the initial configuration P_0. Let $R_k = \{Q \mid P_0 \vdash_{M,x}^{\leq k} Q\}$. The subroutine *count*, given arguments k and $|R_{k-1}|$, outputs $|R_k|$. It does so in the following manner. For each configuration Q, increment a counter d if and only if $Q \in R_k$. To determine if $Q \in R_k$, guess $|R_{k-1}|$ configurations P, and verify that each is in R_{k-1}. For each such P, test if $P = Q$ or $P \vdash_{M,x} Q$. If so, increment d and go to the next value of Q. If not, but there were $|R_{k-1}|$ configurations P found in R_{k-1}, then go to the next Q without incrementing d. Otherwise abort, as there was a wrong guess for some P.

Notice that we cannot test whether $Q \in R_k$ by simply calling *continue_iff_reachable*(k, Q), since this would cause *count* to abort the first time an unreachable Q was encountered. Notice also that we cannot afford the space to generate all $|R_{k-1}|$ configurations P simultaneously.

The details for the function *count* are given in Figure 4.2.

```
function count(k, hyp) returns integer
comment: count(k, |R_{k-1}|) = |R_k| ;
begin
   d ← 0 ;          comment: d counts elements of R_k ;
   for all Q do
      comment: test if Q ∈ R_k ;
      begin
         c ← 0 ;          comment: c counts elements of R_{k-1} ;
         for all P do
            comment: test if P ∈ R_{k-1} ;
            if guess
               then begin
                  continue_iff_reachable(k − 1, P) ;          comment: i.e., P ∈ R_{k-1} ;
                  c ← c + 1 ;
                  if (P = Q) or (P ⊢_{M,x} Q)          comment: i.e., Q ∈ R_k ;
                     then begin
                        d ← d + 1 ;
                        go to nextQ
                     end
               end
         if c ≠ hyp then reject;          comment: wrong guess somewhere ;
         nextQ:
      end ;
   return d
end .
```

Figure 4.2: The Nondeterministic Subroutine *count*

Exercise 4.24: Explain why the "**if** *guess*" test is necessary.

Given the subroutine *count*, it is relatively straightforward to write a nondeterministic algorithm that accepts the complement of M. The method it uses is to iteratively compute $|R_k|$ until k is the running time of M on x. Having done this, it guesses $|R_k|$ nonfinal, reachable configurations and verifies that the correct guesses were made. The details are given in Figure 4.3.

CORRECTNESS OF *count*: To prove the correctness of *count* we want to prove that $count(k, |R_{k-1}|) = |R_k|$ for all k. Assume that $hyp = |R_{k-1}|$. Then, for each Q, we will prove that

(a) there is some sequence of nondeterministic choices that always reaches the label nextQ rather than aborting, and

(b) when the label nextQ is reached, d has been incremented if and only if $Q \in R_k$.

procedure *Mrejects*(x)
comment: accepts x if and only if M does not accept x ;
begin
  compute $S(|x|)$;
  $hyp \leftarrow 1$;   **comment:** $|R_0| = 1$;
  for k **from** 1 **until** hyp does not change **do**
    $hyp \leftarrow count(k, hyp)$;   **comment:** after this, $hyp = |R_k|$;
  comment: k is now the maximum running time of M, and hyp is the number of reachable configurations ;
  $c \leftarrow 0$;   **comment:** c counts elements of R_k ;
  for all nonfinal P **do**
    if *guess*
      then begin
        continue_iff_reachable(k, P) ;
        $c \leftarrow c + 1$
      end ;
  if $c = hyp$ **then accept else reject**
end .

Figure 4.3: The Main Procedure *Mrejects*

Once we prove both (a) and (b) the final value of d is certainly $|R_k|$ on every computation path that has not aborted.

Proof of (a): In the "**if** *guess*" test, guess true for P if and only if $P \in R_{k-1}$. For those $P \in R_{k-1}$ guess a correct computation $P_0 \overset{\leq k-1}{\underset{M,x}{\vdash}} P$ in *continue_iff_reachable*. This ensures that c will attain the value $hyp = |R_{k-1}|$ and will fail the test "**if** $c \neq hyp$", unless of course the branch "**go to** nextQ" is taken earlier. In either case, the label nextQ is reached.

Proof of (b):

"Only if" clause: Since d was incremented there must be a computation $P_0 \overset{\leq k-1}{\underset{M,x}{\vdash}} P$, where either $P = Q$ or $P \underset{M,x}{\vdash} Q$. Hence, $Q \in R_k$.

"If" clause: Assume d has not been incremented but the label nextQ is reached. That means that c has attained the value $hyp = |R_{k-1}|$, we have found all $|R_{k-1}|$ configurations $P \in R_{k-1}$, and for none of them was it true that $P = Q$ or $P \underset{M,x}{\vdash} Q$. Thus $Q \notin R_k$.

Correctness of *Mrejects*: By induction on k and the correctness of *count*, after the first **for** loop in *Mrejects*, hyp is the number of distinct configurations reachable (in any number of steps) from P_0. (Each time an iteration of the **for** loop is completed, $hyp = |R_k|$. The loop is exited when hyp does not change, which means we have accounted for all reachable configurations.)

To prove the correctness of *Mrejects* we want to prove that *Mrejects* accepts x if and only if M does not accept x.

"Only if" clause: Suppose *Mrejects* accepts x. Then it has found hyp reachable but nonfinal configurations, and these are all the reachable configurations. Thus M does not accept x.

"If" clause: Suppose *Mrejects* rejects x. Then for all nondeterministic choices in the "**if** *guess*" test (and in particular, the choice of true if and only if P is reachable), it failed to find *hyp* reachable nonfinal configurations. Hence, at least one of the reachable configurations of M is final, so M accepts x.

ANALYSIS: It remains to show that each of these procedures runs in space $O(S(n))$.

1. Function *guess* runs in $O(1)$ space.

2. Procedure *continue_iff_reachable* needs to keep only two configurations of M and this takes space $O(S(n))$, since $S(n) \geq \log_2 n$. It also needs to count up to k, but we will see below that $k = 2^{O(S(n))}$.

3. Function *count* keeps two configurations P and Q, and two counters c and d, where $c \leq d \leq |R_k| = 2^{O(S(n))}$. Therefore the space needed for counters and configurations is $O(S(n))$.

4. Procedure *Mrejects* maintains counters k, c, and *hyp*, and configuration P. Since c and *hyp* are each at most the number of distinct configurations, each is $2^{O(S(n))}$. Since *hyp* increases by at least 1 every time k is incremented by 1, $k \leq hyp$. Hence the counters and configuration take space $O(S(n))$. Finally, $S(n)$ is space constructible, so can be computed in space $O(S(n))$. □

Open Problem 4.25: Determine whether NTIME$(T(n))$ is closed under complementation for $T(n) \geq n$.

Open Problem 4.26: Find other applications of the "inductive counting" technique of Theorem 4.23. For example, can it be used to simulate limited forms of alternation?

Exercise 4.27: Show that an alternating Turing machine that runs in space $S(n)$ and alternates between existential and universal configurations only a constant number of times can be simulated by a nondeterministic Turing machine that runs in space $S(n)$. What goes wrong with your simulation if the alternating Turing machine alternates more than a constant number of times?

4.8. Exercises

1. Describe in detail how the graph G in the proof of Theorem 4.7 can be constructed in time $2^{O(S(n))}$. Point out the places where the assumption $S(n) \geq \log_2 n$ is used.

2. Carefully prove the correctness part of Theorem 4.9.

3. Explain carefully what would go wrong in Theorem 4.9 if N were nondeterministic instead of deterministic. Assume that *local* is changed to a relation $local(b, b_{-1}, b_0, b_1, b_2)$ that is true if and only if $C_{i+1,j} = b$ is consistent with $C_{i,j+\Delta} = b_\Delta$, for all $\Delta \in \{-1, 0, 1, 2\}$. Why would it be surprising if Theorem 4.9 *did* hold for nondeterministic time?

4. Carefully prove the correctness part of Theorem 4.11.

5. What changes do you need to make to the proof of Theorem 3.8 to make it work with NSPACE substituted for DSPACE? Justify your answer.

6. (a) Demonstrate that Proposition 4.21 is false for some nonconstructible space bound.

   (b) Prove that if $S(n) \geq n$ is the space bound of some deterministic Turing machine, but $S(n)$ is not space constructible, then DSPACE($S(n)$) is not closed under complementation.

7. Determine why the method of Theorem 4.23 does not resolve the question of whether NTIME($T(n)$) is closed under complementation for $T(n) \geq n$.

8. Do Exercise 4.27.

Part II

Case Studies, Reducibility, and Completeness

In Part I we examined results for arbitrary space and time bounds. In Part II, we will specialize in particular time and space bounds, specifically polynomials and logarithms. This shift of attention from arbitrary bounds to specific bounds is motivated by the complexity of computational problems encountered in practice.

Chapter 5

Reducibility, Completeness, and Closure Under Reductions

5.1. Log Space Reducibility

In order to introduce the notion of reducibility, we need to generalize deterministic Turing machines from machines that accept languages to machines that compute functions. To allow our machines to compute functions, we add a one-way, write-only output tape, initially blank, to the deterministic Turing machine model. The contents of this tape are not included in the space bound of the machine.

Definition 5.1: A deterministic Turing machine M *computes a partial function* f if and only if for every input x, if x is in the domain of f, then M halts with $f(x)$ written on the output tape, else M does not halt.

The next definition introduces the notion of "reducibility", which is the means for comparing the complexities of two problems.

Definition 5.2: Let A and B be languages. We say A is *log space, many-one reducible* to B (denoted $A \leq_m^{\mathcal{L}} B$) if and only if there is a (total) function f computable by a deterministic Turing machine M in space $O(\log n)$ such that $x \in A$ if and only if $f(x) \in B$, for all x. In such a case we say M *reduces* A to B.

This says that recognizing A is no harder than computing f (which, by the definition, is "easy") plus recognizing B. This definition is stricter than other common definitions of reducibility in two ways. Firstly, it is common to allow the reducing machine polynomial time rather than only $O(\log n)$ space. Secondly, one often encounters "Turing reducibility" of A to B, which means that you can solve problem A if given arbitrary access to a subroutine that solves problem B. Many-one reducibility is stricter in the sense that, to solve A, the reduction can only call the subroutine for B once at the very end, and must return the value that invocation returns.

Next we show that reducibility is transitive, which is its most important property.

Lemma 5.3: If $A \leq_m^{\mathcal{L}} B$ and $B \leq_m^{\mathcal{L}} C$, then $A \leq_m^{\mathcal{L}} C$.

Proof: Let M reduce A to B and N reduce B to C. On input x we want to output $g(f(x))$, where M computes f and N computes g. Note that $g(f(x))$ is the correct output, since $x \in A$ if and only if $f(x) \in B$ if and only if $g(f(x)) \in C$. The problem is that the length of the intermediate value $f(x)$ could be polynomial in $n = |x|$ (for example, if M outputs a symbol at each step), and the reduction does not have that much space.

Instead, imagine running N on (nonexistent) input $f(x)$. For any i, whenever N needs the ith symbol of $f(x)$, simulate M on x until it produces the ith output symbol (throwing away the previous $i-1$ symbols rather than writing them). The space required for this is $O(\log|x|) = O(\log n)$ for M and $O(\log|f(x)|) = O(\log(n^{O(1)})) = O(\log n)$ for N and i. $\square$

5.2. Hardness, Completeness, and Closure Under Reductions

Definition 5.4: Let $\mathcal{C}$ be a set of languages. A language B is $\leq_m^{\mathcal{L}}$*–hard* for $\mathcal{C}$ if and only if for every $A \in \mathcal{C}$, $A \leq_m^{\mathcal{L}} B$. B is $\leq_m^{\mathcal{L}}$*–complete* for $\mathcal{C}$ if and only if $B \in \mathcal{C}$ and B is $\leq_m^{\mathcal{L}}$*–hard* for $\mathcal{C}$. [1]

Definition 5.5: Let $\mathcal{C}$ be a set of languages. We say $\mathcal{C}$ is *closed under* $\leq_m^{\mathcal{L}}$ if and only if $A \leq_m^{\mathcal{L}} B$ and $B \in \mathcal{C}$ implies $A \in \mathcal{C}$.

The following proposition explains why complete problems are so important: if you understand the complexity of any one complete problem for a class, you understand the complexity of the entire class.

Proposition 5.6: Let $\mathcal{C}$ and $\mathcal{D}$ be sets of languages. Suppose $\mathcal{C}$ is closed under $\leq_m^{\mathcal{L}}$ and B is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{D}$. Then $B \in \mathcal{C}$ if and only if $\mathcal{D} \subseteq \mathcal{C}$.

Proof:

"If" clause: Suppose $\mathcal{D} \subseteq \mathcal{C}$. $B \in \mathcal{D}$ since B is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{D}$, so $B \in \mathcal{C}$.

"Only if" clause: Suppose $B \in \mathcal{C}$. Let A be an arbitrary language in $\mathcal{D}$. Since B is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{D}$, $A \leq_m^{\mathcal{L}} B$. Then $A \in \mathcal{C}$ since $\mathcal{C}$ is closed under $\leq_m^{\mathcal{L}}$. $\square$

The next proposition shows how to use one hard problem to get more.

Proposition 5.7: Let $\mathcal{C}$ be a set of languages. Suppose that A is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{C}$, and $A \leq_m^{\mathcal{L}} B$. Then B is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{C}$.

Proof: Let E be an arbitrary language in $\mathcal{C}$. $E \leq_m^{\mathcal{L}} A$, since A is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{C}$. Then $E \leq_m^{\mathcal{L}} B$ by Lemma 5.3. $\square$

[1] In the literature, these terms have been called "log space hard" and "log space complete". We will use the stated terminology in order to avoid confusion with other possible $O(\log n)$ space reducibilities.

Chapter 6

Deterministic and Nondeterministic Logarithmic Space

Let

$$\begin{aligned}
\mathcal{L} &= \mathrm{DSPACE}(\log n), \\
\mathcal{NL} &= \mathrm{NSPACE}(\log n), \\
\mathcal{P} &= \bigcup_{c>0} \mathrm{DTIME}(n^c) = \mathrm{ASPACE}(\log n), \text{ and} \\
\mathcal{NP} &= \bigcup_{c>0} \mathrm{NTIME}(n^c).
\end{aligned}$$

From Theorem 4.1, $\mathcal{L} \subseteq \mathcal{NL} \subseteq \mathcal{P} \subseteq \mathcal{NP}$. It is an open question (part of Open Problem 4.3 on page 20) whether any of these containments is proper; it is even possible that $\mathcal{L} = \mathcal{NP}$. We concentrate now on the first of these containments:

Open Problem 6.1: Does $\mathcal{L} = \mathcal{NL}$?

As motivation, we know from Savitch's Theorem (Corollary 4.15), that $\mathcal{L} \subseteq \mathcal{NL} \subseteq \mathrm{DSPACE}(\log^2 n)$, so the classes $\mathcal{L}$ and $\mathcal{NL}$ are "very close".

Proposition 6.2: $\mathcal{L}$ is closed under $\leq_m^{\mathcal{L}}$.

Proof: Suppose $A \leq_m^{\mathcal{L}} B$ and $B \in \mathcal{L}$. It is an easy exercise to prove that $B \in \mathcal{L}$ if and only if $B \leq_m^{\mathcal{L}} \{1\}$. By Lemma 5.3, $A \leq_m^{\mathcal{L}} \{1\}$, so $A \in \mathcal{L}$. □

The next proposition shows that concentrating on $O(\log n)$ space may have ramifications to higher space bounds.

Proposition 6.3: If $\mathcal{L} = \mathcal{NL}$, then $\mathrm{DSPACE}(S(n)) = \mathrm{NSPACE}(S(n))$ for all space constructible $S(n) \geq \log_2 n$.

Proof: Let L be accepted by a nondeterministic Turing machine N in space $S(n)$. Consider $L' = \{x10^{2^{S(n)}-n-1} \mid x \in L \text{ and } |x| = n\}$.

$L' \in \mathcal{NL}$, as follows: Check that the input is of the form $y = x10^*$, where $|y| = 2^{S(|x|)}$, and then simulate N on x. The space needed to compute $S(|x|)$, measure $|y|$, and simulate N is all $S(|x|) = \log_2 |y|$.

Since $\mathcal{L} = \mathcal{NL}$, L' is accepted by a deterministic Turing machine D in space $\log_2 n$.

Then $L \in \mathrm{DSPACE}(S(n))$, as follows: On input x, simulate D on $y = x10^{2^{S(n)}-n-1}$. If D's input head tries to leave the right end of x, use a worktape to keep track of its position within y. The space needed is $S(n)$ to compute $S(|x|)$ and keep track of D's input head position, along with another $\log_2 \left(2^{S(n)}\right) = S(n)$ space to simulate D. □

The method of proof in Proposition 6.3 is called "padding", since the strings in L are padded with useless symbols to obtain L'.

6.1. Directed Graph Connectivity

Proposition 5.7 shows how to get more hard problems once you have one. To get the first hard language B for a class, you have to do a generic reduction, that is, reduce a generic language in the class to B. We now present a problem that is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{NL}$.

Let $STCON = \{(G, s, t) \mid G = (V, E)$ is (the encoding of) a directed graph that has a path from vertex s to vertex $t\}$.

Convention 6.4: We will assume that all graphs are encoded as follows: the elements of V are encoded as the binary representations of (not necessarily consecutive) integers, each of length $O(\log |V|)$, and E is encoded as a list of ordered pairs of these binary representations.

Theorem 6.5 (Savitch [41]): $STCON$ is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{NL}$.

Proof: The proof has two parts: we must show that $STCON \in \mathcal{NL}$, and that $STCON$ is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{NL}$.

1. $STCON \in \mathcal{NL}$: Start at s. In general, if the current vertex is u, nondeterministically choose some v such that $(u, v) \in E$, and replace u by v. Accept if and only if the vertex t is reached. The space to record u and v is $O(\log n)$.

2. $STCON$ is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{NL}$: Let $A \in \mathcal{NL}$ be an arbitrary language accepted by some nondeterministic Turing machine N in space $O(\log n)$. Construct a deterministic Turing machine D that reduces A to $STCON$, as follows.

CONSTRUCTION: Given x, D outputs (G, s, t), where

$$\begin{aligned} t &\text{ is } \text{not a configuration of } N \text{ on } x, \\ G &= (V, E), \\ V &= \{P \mid P \text{ is a configuration of } N \text{ on } x\} \cup \{t\}, \end{aligned}$$

$$
\begin{aligned}
E &= \{(P,Q) \mid P \vdash_{N,x} Q\} \cup \{(P,t) \mid P \text{ is a final configuration}\}, \text{ and} \\
s &\text{ is the initial configuration of } N \text{ on } x.
\end{aligned}
$$

Correctness: Let $\vdash^*_{N,x}$ be the reflexive, transitive closure of $\vdash_{N,x}$. Then

$$
\begin{aligned}
x \in A \quad &\text{if and only if} \quad N \text{ accepts } x \\
&\text{if and only if} \quad s \vdash^*_{N,x} P, \text{ for some final configuration } P \\
&\text{if and only if} \quad \text{there is a path from } s \text{ to } t \text{ in } G \\
&\text{if and only if} \quad (G,s,t) \in STCON.
\end{aligned}
$$

Analysis: D can cycle through the configurations P of N in $O(\log n)$ space, since N runs in space $O(\log n)$. For each such P, D can output the edges $\{(P,Q) \mid P \vdash_{N,x} Q\}$ in $O(\log n)$ space. Note that D produces output with size polynomial in n, but this output is not counted toward the space bound. □

Corollary 6.6: $STCON \in \mathcal{L}$ if and only if $\mathcal{L} = \mathcal{NL}$.

Proof: This follows from Propositions 5.6 and 6.2, and Theorem 6.5. □

Thus, Open Problem 6.1 is equivalent to resolving whether the particular language $STCON$ is in $\mathcal{L}$.

Corollary 6.7: If $STCON \in \mathcal{L}$, then DSPACE($S(n)$) = NSPACE($S(n)$) for all space constructible $S(n) \geq \log_2 n$.

Proof: This follows from Proposition 6.3 and Corollary 6.6. □

6.2. Undirected Graph Problems

6.2.1. Directed vs. Undirected Connectivity

Exercise 6.8: Let $\overline{STCON} = \{(G,s,t) \mid G \text{ has no path from } s \text{ to } t\}$. Show that $\overline{STCON}$ is $\leq_m^{\mathcal{L}}$-complete for $\mathcal{NL}$.

Now consider the undirected version of $STCON$, namely

$$USTCON = \{(G,s,t) \mid \ G \text{ is an undirected graph with a path from } s \text{ to } t\ \}.$$

Then $USTCON \in \mathcal{NL}$.

Open Problem 6.9: Is $USTCON \in \mathcal{L}$? Alternatively, is $USTCON \leq_m^{\mathcal{L}}$–complete for $\mathcal{NL}$?

The remainder of this section suggests evidence that $USTCON$ is not as hard as $STCON$, and hence is not $\leq_m^{\mathcal{L}}$–complete for $\mathcal{NL}$.

Definition 6.10: For any $k \geq 1$, $\mathcal{SC}^k = \{L \mid L$ is accepted by some deterministic Turing machine in $O(\log^k n)$ space and polynomial time, simultaneously$\}$.

Note that $\mathcal{SC}^1 = \mathcal{L}$, and $\mathcal{SC}^k \subseteq \mathcal{P} \cap \mathrm{DSPACE}(\log^k n)$, with equality unlikely in the latter.

Definition 6.11: $\mathcal{SC} = \bigcup_{k \geq 1} \mathcal{SC}^k$.

Theorem 6.12 (Nisan [32]): $USTCON \in \mathcal{SC}^2$.

Open Problem 6.13: Is $STCON \in \mathcal{SC}$? The conjecture is "no", and some (relatively weak) evidence is provided by Tompa [47]. If the answer is indeed "no", then $STCON \not\leq_m^{\mathcal{L}} USTCON$, since $\mathcal{SC}^k$ is closed under $\leq_m^{\mathcal{L}}$ for $k \geq 1$.

6.2.2. Shortest Paths in Undirected Graphs

Although $USTCON$ seems unlikely to be $\leq_m^{\mathcal{L}}$–complete for $\mathcal{NL}$, the problem of finding shortest paths in undirected graphs is. More specifically, let

$$USP = \{(G,s,t,k) \mid G \text{ is an undirected graph with a path of length at most } k \text{ from } s \text{ to } t\ \}.$$

Convention 6.14: We assume all integers to be encoded in binary, unless otherwise specified.

Theorem 6.15: USP is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{NL}$.

Proof:

1. $USP \in \mathcal{NL}$: This is as in the proof of Theorem 6.5, but also count the length of the path traversed from s, and reject if this exceeds $\min(n,k)$. The reason for this "min" is that, if the encoding of k has $\omega(\log n)$ bits, there will not be enough space for the counter. However, a shortest path never passes through any vertex more than once. Hence, $|V| < n$ is an upper limit on the length of the path.

2. USP is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{NL}$: We need not do another generic reduction, since we already have a problem $STCON$ that is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{NL}$. Hence, by Proposition 5.7, it suffices to show that $STCON \leq_m^{\mathcal{L}} USP$.

CONSTRUCTION: On input (G,s,t), where $G=(V,E)$, we will make $|V|$ copies of V, with edges corresponding to E between consecutive copies. More precisely, output (G',s',t',k), where

$$\begin{aligned} G' &= (V',E'), \\ V' &= \{v_{i,j} \mid i \in V, 1 \leq j \leq |V|\}, \end{aligned}$$

$$\begin{aligned}
E' &= \{\{v_{i,j}, v_{k,j+1}\} \mid (i,k) \in E, 1 \le j \le |V|-1\} \cup \{\{v_{t,j}, v_{t,j+1}\} \mid 1 \le j \le |V|-1\}, \\
s' &= v_{s,1}, \\
t' &= v_{t,|V|}, \text{ and} \\
k &= |V| - 1.
\end{aligned}$$

CORRECTNESS: We need to show that $(G, s, t) \in STCON$ if and only if $(G', s', t', k) \in USP$.

"Only if" clause: Assume that $(G, s, t) \in STCON$. Then there is an $l \le |V|$ such that G has a path of length exactly $l-1$ from s to t. Then G' has a path of length $l-1$ from $v_{s,1}$ to $v_{t,l}$, and hence a path of length $|V|-1$ from $v_{s,1}$ to $v_{t,|V|}$. Thus, $(G', s', t', k) \in USP$.

"If" clause: Assume that $(G', s', t', k) \in USP$. Then there is an $l \le |V|$ such that G' has a path of length exactly $l-1$ from $v_{s,1}$ to $v_{t,l}$ that does not pass through $v_{t,j}$ for any $j < l$. Because the length $l-1$ of this path equals the difference of the second subscripts, and because it does not use any of the edges $\{v_{t,j}, v_{t,j+1}\}$, it corresponds to a path in G from s to t. Thus, $(G, s, t) \in STCON$.

ANALYSIS: To construct G', we need enough space to store the index j so that, for each input $i \in V$, we can output $|V|$ copies $v_{i,j}$, and for each input $(i,k) \in E$, we can output $|V|-1$ copies $\{v_{i,j}, v_{k,j+1}\}$. Since n is an upper limit on $|V|$, at most $\log n$ bits are needed to store j. $\square$

For additional problems $\le_m^{\mathcal{L}}$-complete for $\mathcal{NL}$, see Jones, Lien, and Laaser [21].

6.3. Exercises

1. For any language B, prove that $B \in \mathcal{L}$ if and only if $B \leq_m^{\mathcal{L}} \{1\}$.

2. Let the "formula value problem" be defined as $FV = \{(F, A) \mid F$ is a propositional formula with some number k of variables, and A is a satisfying truth assignment for F, that is, A makes F true$\}$. Show that $FV \in \mathcal{L}$.

3. Let *CYCLE* be the set of directed graphs that contain a cycle. Prove that *CYCLE* is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{NL}$.

4. A directed graph G is said to be k*-connected* if and only if, for every set U of $k-1$ vertices and for every pair (v, w) of vertices not in U, there is a path from v to w that does not contain any vertex in U. Let $CONNECTIVITY = \{(G, k) \mid G$ is k-connected$\}$. Prove that *CONNECTIVITY* is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{NL}$.

   (Hint: Use Exercise 4.27.)

5. Let *2UNSAT* be the set of unsatisfiable propositional formulas in conjunctive normal form with at most two literals per clause. Prove that *2UNSAT* is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{NL}$. What is the complexity of *2SAT*, the set of satisfiable formulas in conjunctive normal form with at most two literals per clause? (I.e., is it in $\mathcal{NL}$? Is it $\leq_m^{\mathcal{L}}$–hard for $\mathcal{NL}$?)

6. Do Exercise 6.8.

7. Prove that $\mathcal{SC}$ is closed under $\leq_m^{\mathcal{L}}$.

Chapter 7

Deterministic Polynomial Time

7.1. Motivating Completeness for $\mathcal{P}$

Having dealt with the first containment in $\mathcal{L} \subseteq \mathcal{NL} \subseteq \mathcal{P} \subseteq \mathcal{NP}$, we move on to $\mathcal{P}$ and a study of $\leq_m^{\mathcal{L}}$–completeness for $\mathcal{P}$. One motivation, of course, is that any problem $\leq_m^{\mathcal{L}}$–complete for $\mathcal{P}$ will be in $\mathcal{L}$, or $\mathcal{NL}$, or $\mathcal{SC}$, if and only if all problems in $\mathcal{P}$ are. The following complexity class encompasses all three of these classes:

Definition 7.1: $\mathcal{POLYLOG} = \bigcup_{c>0} \mathrm{DSPACE}(\log^c n)$.

Note that $\mathcal{POLYLOG} = \bigcup_{c>0} \mathrm{NSPACE}(\log^c n)$ due to Corollary 4.15, and that $\mathcal{POLYLOG} = \bigcup_{c>0} \mathrm{ATIME}(\log^c n)$ due to Theorems 4.11 and 4.14. It is also clear that $\mathcal{SC} \subseteq \mathcal{POLYLOG}$, since $\mathcal{SC}$ is a time-bounded version of $\mathcal{POLYLOG}$.

As an example of a problem in $\mathcal{POLYLOG}$, group isomorphism (*i.e.*, given the multiplication tables of two finite groups G and H, is $G \cong H$?) is in $\mathrm{DSPACE}(\log^2 n)$ [29], but not known to be in $\mathcal{NL}$ or in $\mathcal{SC}$, or even in $\mathcal{P}$.

Open Problem 7.2: Is $\mathcal{P} \subseteq \mathcal{POLYLOG}$? The conjecture is "no", and some evidence is given by Cook and Sethi [6].

Since $\mathcal{POLYLOG}$ is closed under $\leq_m^{\mathcal{L}}$ (left as an exercise to the reader), any problem that is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{P}$ is in $\mathcal{POLYLOG}$ if and only if $\mathcal{P} \subseteq \mathcal{POLYLOG}$, by Proposition 5.6.

Another motivation is that problems that are $\leq_m^{\mathcal{L}}$–complete for $\mathcal{P}$ are "inherently sequential", assuming $\mathcal{P} \not\subseteq \mathcal{POLYLOG}$, meaning that they cannot be very efficiently parallelized. We will return to this topic in Section 7.5.

7.2. Boolean Circuits

The immediate reason to introduce Boolean circuits is that they are the subject of the first problem that will be shown $\leq_m^{\mathcal{L}}$–complete for $\mathcal{P}$. In addition, we will have occasion to study circuits as computational devices in Section 7.5.

Definition 7.3: A *(Boolean* or *combinational) circuit* is an acyclic, oriented, directed graph with two distinguished subsets of vertices, called *inputs* and *outputs*. ("Oriented" means that the edges directed into any particular vertex are ordered.) The vertices have indegrees and labels as follows:

- *Inputs* have indegree 0 and are labeled consecutively from the set of indeterminates $\{x_1, x_2, \ldots, x_n\}$.
- All noninput vertices are called *gates*. A gate with indegree d is labeled by any function $f : \{0,1\}^d \to \{0,1\}$.

In what follows, unless specified otherwise it is assumed that the maximum indegree d is a constant with respect to the number n of input labels. Such circuits are said to have *bounded fanin*.

Definition 7.4: The *value* $\mathrm{val}(u) \in \{0,1\}$ of vertex u in a circuit on input $(b_1, b_2, \ldots, b_n) \in \{0,1\}^n$ is defined as follows:

- If u is an input with label x_i then $\mathrm{val}(u) = b_i$.
- If u is a gate with label $f : \{0,1\}^d \to \{0,1\}$, and the d vertices with edges directed to u are $u_1, u_2, \ldots, u_d$, then $\mathrm{val}(u) = f(\mathrm{val}(u_1), \mathrm{val}(u_2), \ldots, \mathrm{val}(u_d))$.

A circuit with 1 output vertex u is said to *output* b on input $(b_1, b_2, \ldots, b_n)$ if the value of u on input $(b_1, b_2, \ldots, b_n)$ is equal to b.

We usually restrict the labels f to be from {AND, OR, NOT}.

Definition 7.5: If the labels f must be in {AND, OR}, then the circuit is called *monotone*.

Note that arbitrary functions cannot be computed by monotone circuits. For example, the function NOT cannot be so computed. Only "monotone functions" can be computed by monotone circuits, where a monotone function has the property that, if any input is changed from 0 to 1, the output cannot change from 1 to 0.

The next definitions introduce natural complexity measures for circuits.

Definition 7.6: The *size* of a circuit C is the number of gates in C.

Definition 7.7: If u is a vertex of a circuit C, then $\mathrm{depth}(u)$ is the length of any longest path from any input to u. The *depth of* C is the maximum, over all vertices u in C, of $\mathrm{depth}(u)$.

If a circuit is viewed as a sequential computing device, then size is a measure of sequential time, because each gate represents one operation. On the other hand, if a circuit is viewed as a parallel computing device, then depth is the appropriate measure of time, since all gates at a given depth can be evaluated in parallel. In the latter case, size is a measure of the number of parallel processors required.

7.3. One-Read Alternating Turing Machines

Definition 7.8: A *one-read alternating Turing machine* is an alternating Turing machine that, on each path of its computation tree, writes only one string of the form ia on its index tape, where $i \in \{0,1\}^*$ and $a \in \Sigma$. It then enters a special "read" state q_{read} and halts in the next step, accepting if and only if $x_i = a$, where x_i is the ith input symbol. Its computation on that path, and in particular whether it accepts, depends on no other input symbol.

Lemma 7.9: An alternating Turing machine M that runs in $T(n) \geq \log_2 n$ time and $S(n) \geq \log_2 n$ space can be simulated by a one-read alternating Turing machine N that runs in $O(T(n))$ time and $O(S(n))$ space.

Proof:

CONSTRUCTION: On input x, N simulates M, but records M's index tape on a separate worktape. If M has i written on its index tape, N does the following to simulate one more step of M:

existentially choose $a \in \Sigma$;
universally choose $b \in \{0,1\}$;
case b **of**
  0: **if** $x_i = a$ **then accept else reject**;
  1: continue simulating M as though M received the response $x_i = a$
end .

N does the test "$x_i = a$?" by writing ia on its index tape and entering q_{read}.

Example 7.10: Suppose $\Sigma = \{a, b\}$, and on some computation path in M we have two steps that read input characters x_i and x_j, respectively. Then the corresponding portion of the computation tree for N is shown in Figure 7.1.

CORRECTNESS: The only new universal configurations that can possibly be accepting are those for which the correct value x_i was guessed, and those are accepting if and only if M's continuation is accepting.

ANALYSIS: Along any path in the original computation tree for M, each step is replaced by 3 steps of N; in addition, at the end of the path there is an added $O(\log n)$ time to copy i to the index tape. Thus, the total time is $3T(n) + \log_2 n = O(T(n))$, since $T(n) \geq \log_2 n$. The only additional space that N uses is the $\log_2 n$ space needed to record the contents of M's index tape on a separate worktape. Since $S(n) \geq \log_2 n$, this is $O(S(n))$. Note, though, that this method may greatly increase the number of alternations between existential and universal configurations. □

7.4. The Circuit Value Problem

Let

$$MCV = \{(C, b_1, b_2, \ldots, b_k) \mid C \text{ is a monotone circuit with } k \text{ inputs that outputs 1 on input } (b_1, b_2, \ldots, b_k)\ \} .$$

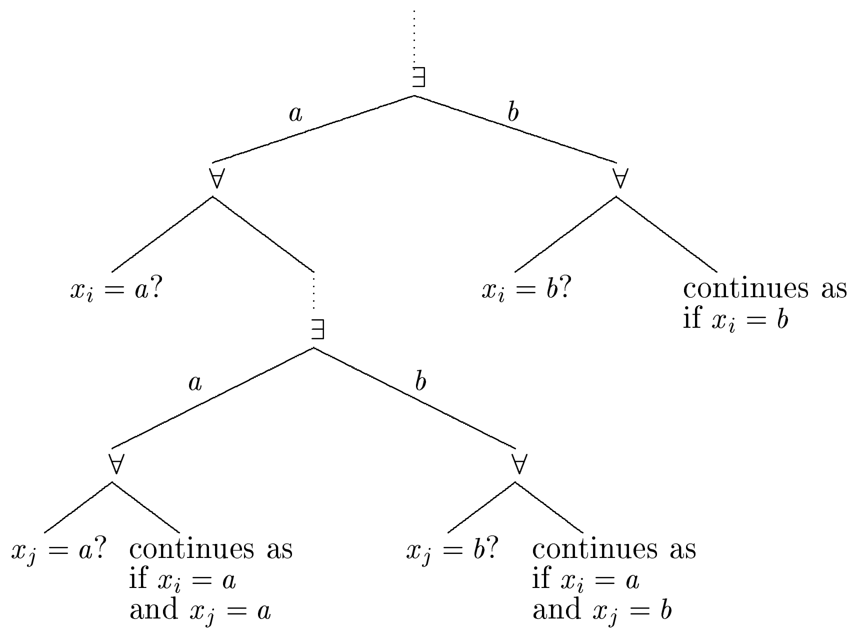

Figure 7.1: Computation Tree for One-Read Alternating Turing Machines

Theorem 7.11 (Ladner [27], Goldschlager [11]): *MCV* is $\leq_m^{\mathcal{L}}$-complete for $\mathcal{P}$.

Proof:

1. $MCV \in \mathcal{P}$: By Theorem 4.7 it suffices to show that *MCV* is accepted by an alternating Turing machine A that runs in space $O(\log n)$.

Construction: Starting at C's output gate, A does the following. Let u be the vertex currently being evaluated by A.

- Case 1: If u is an input of C labeled x_i, then A accepts if and only if $b_i = 1$.
- Case 2: If u is an OR (AND) gate with input vertices v_0 and v_1, then A existentially (respectively, universally) chooses $b \in \{0, 1\}$, replaces u with v_b, and continues by evaluating v_b.

Correctness: By induction on depth(u), A is in an accepting configuration when evaluating u if and only if val$(u) = 1$.

Basis (depth$(u) = 0$): Then u is an input. By Case 1 above, A accepts if and only if val$(u) = b_i = 1$.

Induction (depth$(u) > 0$): Assume the induction hypothesis holds for all vertices v such that depth$(v) \leq k$. Let u be a vertex in C such that depth$(u) = k + 1$, and $u = v_0$ OR v_1 ($u = v_0$ AND v_1). By the induction hypothesis, A is in an accepting configuration when evaluating v_0 if and only if val$(v_0) = 1$, and A is in an accepting configuration when evaluating v_1 if and only if val$(v_1) = 1$. By the construction, A is in an accepting configuration when evaluating u if and only if A is in an accepting configuration when evaluating v_0 or (respectively, and) when evaluating v_1, which occurs if and only if $\mathrm{val}(v_0) = 1$ or (respectively, and) $\mathrm{val}(v_1) = 1$, which in turn occurs if and only if $\mathrm{val}(u) = 1$.

ANALYSIS: Since $(C, b_1, b_2, \ldots, b_k)$ is already in A's input, A need only store u, v_b, and i, all three of which use $O(\log n)$ space.

2. MCV is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{P}$: Let L be an arbitrary language in $\mathcal{P}$. By Theorem 4.9, we can assume that L is accepted by an alternating Turing machine A in space $\log_2 n$. It suffices to produce a deterministic Turing machine D running in $O(\log n)$ space that reduces L to MCV. Without loss of generality, we will make three simplifying assumptions about A:

1. By Lemma 7.9, we can assume that A is a one-read alternating Turing machine.
2. Assume that every configuration of A has either zero or two immediate successors.
3. Assume that A uses a separate worktape to count how many steps it has taken. The purpose of this is to prevent A from repeating any configuration along any path of its computation tree. If this counter exceeds the maximum running time of A (which is easily computable since its space bound is $\log_2 n$), then A halts and rejects.

CONSTRUCTION: On input x, D outputs $(C, b_1, b_2, \ldots, b_k)$, where C contains a vertex u_P for every configuration P of A, and has labels and edges as follows:

- If P is an existential (universal) configuration with $P \underset{A,x}{\vdash} Q$ and $P \underset{A,x}{\vdash} R$, then u_P is an OR (respectively, AND) gate with inputs u_Q and u_R.
- If P is a configuration in state q_{read} with ia on its index tape, then u_P is an input vertex with value
$$\begin{cases} 1 & \text{if } x_i = a \\ 0 & \text{if } x_i \neq a \end{cases}.$$
D can determine whether $x_i = a$, since i is written on its worktape as part of P, and x is on its input tape.
- If P is a final (nonfinal) halting configuration, then u_P is an input vertex with value 1 (respectively, 0).
- If P is the initial configuration, then u_P is the output gate.

Note that C is acyclic, since A is constructed so that it never repeats any configuration along a single computation path. Note also why we need A to be one-read.

CORRECTNESS: By induction on $\mathrm{depth}(u_P)$, P is an accepting configuration if and only if $\mathrm{val}(u_P) = 1$.

Basis ($\mathrm{depth}(u_P) = 0$): Then u_P is an input vertex. By construction, $\mathrm{val}(u_P) = 1$ if and only if P is either a final configuration, or a configuration containing q_{read} that is about to accept.

Induction ($\mathrm{depth}(u_P) > 0$): Suppose that P is an existential (universal) configuration with $P \underset{A,x}{\vdash} Q$ and $P \underset{A,x}{\vdash} R$. Then P is accepting if and only if either Q or R is accepting (respectively, both Q and R are accepting) which, by the induction hypothesis, is true if and only if $\mathrm{val}(u_Q) = 1$ or (respectively, and) $\mathrm{val}(u_R) = 1$, which is true if and only if $\mathrm{val}(u_P) = 1$.

Analysis: D uses $O(\log n)$ space to enumerate each of A's configurations P, to find the configurations Q such that $P \underset{A,x}{\vdash} Q$ in order to output the edge (u_Q, u_P), and to count to i in order to find x_i on the input tape. This is all possible since A's configurations only have length $O(\log n)$, and $i \leq n$. □

Let $CV = \{(C, b_1, b_2, \ldots, b_k) \mid C$ is a circuit with k inputs and with gates labeled from {AND, OR, NOT} that outputs 1 on input $(b_1, b_2, \ldots, b_k)\}$.

Corollary 7.12 (Ladner [27]): CV is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{P}$.

Notice that the fact that monotone circuits are provably weaker than general circuits does not contradict the fact that $CV \leq_m^{\mathcal{L}} MCV$: the latter says that it is as hard to predict the output of a monotone circuit as it is to predict the output of a general circuit.

7.5. Relating Circuits to $\mathcal{P}$

In this section we present a slightly different view of the relationship between circuits and $\mathcal{P}$. Rather than viewing circuits as the subject of a problem to be solved by a polynomial time deterministic Turing machine, circuits will be viewed as computing devices themselves, and we will be interested in how their power relates to the power of Turing machines. We begin by discussing circuits as computing devices with associated complexity measures.

Definition 7.13: Let C be a circuit with n inputs and 1 output. The *language accepted by C* is $L(C) = \{x \in \{0,1\}^n \mid C \text{ outputs 1 on input } x\}$.

Note that $L(C) \subseteq \{0,1\}^n$ is always finite. Therefore, in defining circuit complexity classes below, we must consider an infinite family of circuits, one for each input size n.

Definition 7.14: $\mathrm{SIZE}(S(n)) = \{L \mid$ for all n, $L \cap \{0,1\}^n$ is accepted by some circuit C_n with n inputs, bounded fanin, and size $S(n)\}$.

Definition 7.15: $\mathrm{DEPTH}(D(n)) = \{L \mid$ for all n, $L \cap \{0,1\}^n$ is accepted by some circuit C_n with n inputs, bounded fanin, and depth $D(n)\}$.

Definition 7.16: $\mathcal{PSIZE} = \bigcup_{c>0} \mathrm{SIZE}(n^c)$.

If $n = 1$, size=1 suffices.

Theorem 7.17: $\mathcal{P} \subseteq \mathcal{PSIZE}$.

Proof: It suffices to simulate an alternating Turing machine that runs in $O(\log n)$ space by a family of polynomial size circuits C_n, by Theorem 4.9. This is what was done in proving MCV is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{P}$, in Theorem 7.11, except:

- if P is a configuration in state q_{read} with ia on its index tape, then u_P is
  - an input vertex labeled x_i, if $a = 1$,
  - the negation $\neg x_i$ of an input vertex, if $a = 0$, and
  - the constant 0, if $a \notin \{0, 1\}$.

□

The state of the art in refining Theorem 7.17 is given in the following theorem:

Theorem 7.18 (Pippenger and Fischer [35]): $\text{DTIME}(T(n)) \subseteq \text{SIZE}(O(T(n) \log T(n)))$.

Open Problem 7.19: Improve Theorem 7.18.

Ruzzo [39] showed a tighter relationship between alternating Turing machines and circuits than Theorem 7.17. He first observed that the same simulation demonstrates that an alternating Turing machine running in time $T(n)$ and space $S(n)$ (simultaneously) can be simulated by a family of circuits C_n of depth $O(T(n))$ and size $2^{O(S(n))}$ simultaneously. Because of the nonuniformity in the definition of SIZE and DEPTH (i.e., C_n has, in general, no resemblance to C_{n+1}), it is impossible for the converse to hold, that is, for a single alternating Turing machine to simulate an infinite family of circuits efficiently. The following exercise makes this point as conclusively as one would like:

Exercise 7.20: Prove that SIZE(1) contains nonrecursive sets. (Hint: define a nonrecursive language that, for all nonnegative integers n, either contains every string or no string in $\{0, 1\}^n$.)

To overcome this obstacle, Ruzzo defined "uniform" families of circuits, and showed that alternating time $O(T(n))$ and space $O(S(n))$ is *equivalent* to uniform circuit depth $O(T(n))$ and size $2^{O(S(n))}$ simultaneously. In this sense, alternating Turing machines are a good model of parallel algorithms, with time measuring parallel time and space measuring the logarithm of the number of processors.

Definition 7.21: For any $k \geq 1$, $\mathcal{NC}^k = \{L \mid L$ is accepted by some alternating Turing machine in time $O(\log^k n)$ and space $O(\log n)$ simultaneously$\}$.

Definition 7.22: $\mathcal{NC} = \bigcup_{k \geq 1} \mathcal{NC}^k$.

By Ruzzo's Theorem, $\mathcal{NC}$ is the class of languages that can be accepted by extremely fast parallel algorithms using a polynomial amount of hardware.

Here is how $\mathcal{NC}$ fits in with previously studied complexity classes:

$$\mathcal{NC}^1 \subseteq \mathcal{L} \subseteq \mathcal{NL} \subseteq \mathcal{NC}^2 \subseteq \mathcal{NC} \subseteq \mathcal{P} \cap \mathcal{POLYLOG}.$$

The first containment follows from Theorem 4.11, the third from the proof of Theorem 4.14 (where only $O(\log n)$ space is needed to retain three configurations), $\mathcal{NC} \subseteq \mathcal{P}$ from Theorem 4.7, and $\mathcal{NC} \subseteq \mathcal{POLYLOG}$ again from Theorem 4.11.

If any language that is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{P}$ is in $\mathcal{NC}$, then $\mathcal{P} = \mathcal{NC}$. Thus, assuming $\mathcal{P} \neq \mathcal{NC}$, these hard problems are "inherently sequential". This provides another strong motivation for the study of completeness for $\mathcal{P}$.

7.6. Other Problems Complete for $\mathcal{P}$

To indicate the variety of natural problems that are known to be $\leq_m^{\mathcal{L}}$–complete for $\mathcal{P}$, seven of them are described below. The source for these and other problems is a compendium by Greenlaw, Hoover, and Ruzzo [13], which describes approximately 100 such problems. The proof of the first of these seven will be given in Section 7.7, but the remaining proofs are omitted.

1. Linear Programming [7, 24]

   INPUTS: $n \times d$ integer matrix A, vector b of n integers.

   PROBLEM: Is there a vector x of d rational numbers such that $Ax \leq b$?

   The ordinary linear programming problem, which asks to maximize the inner product $c \cdot x$ subject to $Ax \leq b$, is transformed most naturally into a language recognition problem by adding one more constraint $c \cdot x \geq k$, where c and k are also part of the input. This extra constraint can then be incorporated as a new row of A and b.

2. Maximum Flow [12]

   INPUTS: Directed graph G with a nonnegative integer capacity c_e for each edge e, and two designated vertices s and t.

   PROBLEM: Is the maximum "flow" from s to t odd?

   The *flow* f_e on any edge e is a nonnegative number that satisfies the following conditions:

   - $f_e \leq c_e$ for each edge e, and
   - the total flow into vertex v equals the total flow out of vertex v, for all $v \notin \{s, t\}$.

   The *flow from s to t* is the total net flow into t. The question of whether the maximum flow from s to t is at least a designated integer k is also $\leq_m^{\mathcal{L}}$–complete for $\mathcal{P}$ [28].

3. First Fit Decreasing Bin-Packing [1]

   INPUTS: Rational numbers $v_1, v_2, \ldots, v_n \in [0, 1]$, integers i and b.

   PROBLEM: Is v_i put into bin b by the First Fit Decreasing heuristic?

   First Fit Decreasing is the following heuristic: consider each v_i in decreasing order, inserting it into the least numbered bin in which it still fits.

4. Depth First Search [37]

   INPUTS: Adjacency lists for a graph G, and three distinguished vertices s, u, and v.

   PROBLEM: Is u visited before v by the Depth First Search algorithm, assuming it starts at vertex s and handles adjacencies in the order dictated by the adjacency lists?

5. Unit Resolution [20]

   INPUTS: A Boolean formula F in conjunctive normal form.

   PROBLEM: Can the empty clause $\Box$ be deduced from F by unit resolution?

   As an example of unit resolution, if two of the clauses in F are $A = \neg y$ and $B = x \vee y \vee \neg z$, then unit resolution adds the clause $x \vee \neg z$ to F.

Deducing the empty clause is a way of proving unsatisfiability of F in resolution. If instead one wanted to prove that F implies some clause E, then $\neg E$ could be added to F and proved unsatisfiable.

6. Unification [8]

   INPUTS: Two terms s and t, each composed of variables and function symbols.

   PROBLEM: Are there substitutions of terms for the variables in s and t that cause them to become equal?

   As an example, the terms $s = f(x, g(x, y))$ and $t = f(g(w, w), z)$ can be unified by setting $x = g(w, w)$ and $z = g(g(w, w), y)$.

7. Context-Free Grammar Emptiness

   INPUTS: Context-free grammar G.

   PROBLEM: Is $L(G) = \emptyset$?

Citation purposely omitted: to be used as homework

7.7. Linear Programming

Theorem 7.23 (Dobkin, Lipton, and Reiss [7], Khachian [24]): Linear Programming is $\leq_m^{\mathcal{L}}$-complete for $\mathcal{P}$.

Proof:

1. Linear Programming $\in \mathcal{P}$: This is difficult and was open for many years. It was finally proved by Khachian [24].

2. $MCV \leq_m^{\mathcal{L}}$ Linear Programming: This proof is due to Cook [13].

CONSTRUCTION: Given a monotone circuit C and its k input values $b_1, b_2, \ldots, b_k$, we must output A and b such that $Ax \leq b$ has a solution x if and only if C outputs 1 on inputs $b_1, b_2, \ldots, b_k$.

For each vertex u of C, there is a corresponding variable x_u and some constraints as follows:

- For each input vertex u with value b, output the inequalities representing
  - ▷ $x_u = b$ (actually represented by two inequalities $x_u \leq b$ and $-x_u \leq -b$).
- For each OR gate u with inputs v and w, output the inequalities
  - ▷ $0 \leq x_u \leq 1$,
  - ▷ $x_v \leq x_u$ (i.e., if $\text{val}(v) = 1$, then $\text{val}(u) = 1$),
  - ▷ $x_w \leq x_u$ (i.e., if $\text{val}(w) = 1$, then $\text{val}(u) = 1$), and
  - ▷ $x_u - x_v - x_w \leq 0$ (i.e., if $\text{val}(v) = 0$ and $\text{val}(w) = 0$, then $\text{val}(u) = 0$).
- For each AND gate u with inputs v and w, output the inequalities
  - ▷ $0 \leq x_u \leq 1$,
  - ▷ $x_u \leq x_v$ (i.e., if $\text{val}(v) = 0$, then $\text{val}(u) = 0$),

- ▷ $x_u \le x_w$ (i.e., if val$(w) = 0$, then val$(u) = 0$), and
- ▷ $x_v + x_w - x_u \le 1$ (i.e., if val$(v) = 1$ and val$(w) = 1$, then val$(u) = 1$).

- If u is the output of C, output the inequalities representing
  - ▷ $x_u = 1$.

CORRECTNESS: By induction on depth(u), the only possible solution for x_u is val(u). Hence, $Ax \le b$ has a solution if and only if C outputs 1.

ANALYSIS: Space $O(\log n)$ suffices to hold u, v, w, and the label of a single input vertex. □

7.8. Problems in $\mathcal{P}$ Not Known to be Complete

Open Problem 7.24: Each of the following problems is known to be in $\mathcal{P}$, but not known to be either complete for $\mathcal{P}$ or to be in $\mathcal{POLYLOG}$. (Of course, they need not necessarily be either.)

1. Greatest Common Divisor

   INPUTS: n-bit integers x and y.

   PROBLEM: Compute $\gcd(x, y)$.

   This problem is known to be in $\mathcal{P}$ by Euclid's algorithm. The problem is still open even if we just want to determine if x and y are relatively prime, i.e., $\gcd(x, y) = 1$.

2. Modular Exponentiation

   INPUTS: n-bit integers a, e, and m.

   PROBLEM: Compute $a^e \bmod m$.

   This may not seem entirely natural, but there are other important problems that are reducible to this. These include probabilistic primality checking, RSA encryption and decryption [38], computing inverses modulo primes, and computing square roots modulo certain primes.

3. Edge Weighted Matching

   INPUTS: Graph G with positive integer edge weights.

   PROBLEM: Find a matching of maximum weight.

4. Stable Marriage

   INPUTS: n men and n women each with a complete ordered list of marital preferences.

   PROBLEM: Find n marriages such that there do not exist a man and a woman who each prefer the other over his or her own spouse.

5. Comparator Circuit Value

   INPUTS: Circuit C of comparators (each of which takes two inputs x and y and outputs exactly one copy each of $\max(x, y)$ and $\min(x, y)$), vector $(b_1, b_2, \ldots, b_k) \in \{0, 1\}^k$, and integer i.

   PROBLEM: Is the ith output of C on input $(b_1, b_2, \ldots, b_k)$ a one?

It is interesting to note that problems 3, 4, and 5 are all equivalent (i.e., reducible to each other) and that there are no known reductions between problems 1 and 2.

7.9. Exercises

1. Show that $CV \in \mathcal{P}$.
2. Do Exercise 7.20.
3. Prove that $\mathcal{NC}$ is closed under $\leq_m^{\mathcal{L}}$.
4. Let *CFGempty* be the set of context-free grammars G such that $L(G) = \emptyset$. Prove that *CFGempty* is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{P}$.

Chapter 8

Nondeterministic Polynomial Time

8.1. Satisfiability of Propositional Formulas

In this section we present a new proof that SAT is $\mathcal{NP}$-complete. Given the machinery that we have developed, this is much simpler than the one originally given by Cook [4].

Definition 8.1: A circuit is in *normal form* if and only if the NOT gates have as inputs only circuit inputs.

Exercise 8.2: Show that any circuit can be put into normal form with an increase of 1 in depth and doubling in size.

Definition 8.3: $CSAT = \{C \mid C \text{ is a circuit in normal form with } L(C) \neq \emptyset\}$.

Theorem 8.4: $CSAT$ is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{NP}$.

Proof:

1. $CSAT \in \mathcal{NP}$: Given a circuit C with k inputs, guess input values $(b_1, b_2, \ldots, b_k) \in \{0,1\}^k$ and accept if and only if C outputs 1 on input $(b_1, b_2, \ldots, b_k)$. The circuit evaluation is done in polynomial time as in Theorem 7.11.

2. $CSAT$ is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{NP}$: Let N be a nondeterministic Turing machine that accepts a language L in polynomial time $p(n)$. Assume without loss of generality that each configuration of N has at most two immediate successors. N can be simulated by a nondeterministic Turing machine N' that behaves as follows. On input x, N' computes $p(n)$ (which can be done deterministically), existentially chooses $y \in \{0,1\}^{p(n)}$, and writes $\#y$ after x on the input tape. N' then returns its head to the left end of x and simulates some deterministic Turing machine D on input $x\#y$, where D simulates N, but consumes another bit of y whenever it needs to simulate a nondeterministic choice. By Theorem 4.9, there is an alternating Turing machine A that uses space $O(\log n)$ and accepts the same language as D.

Construction: We must construct a deterministic Turing machine M that, on input x, outputs some circuit C that is in $CSAT$ if and only if there exists a y such that A accepts $x\#y$. M first computes $p(n)$, and then proceeds exactly as in Theorems 7.11 and 7.17, combining the input conventions of those two proofs as follows:

- If P is a configuration in state q_{read} with ia on its index tape, then:
  - if $i \leq |x| + 1$, then u_P is a constant gate with value
    - 1, if $a = (x\#)_i$, and
    - 0, otherwise;
  - if $i > |x| + 1$, then u_P is
    - an input vertex labeled $y_{i-|x|-1}$, if $a = 1$,
    - the negation $\neg y_{i-|x|-1}$ of an input vertex, if $a = 0$, and
    - the constant 0, if $a \notin \{0, 1\}$.

CORRECTNESS:

N accepts x	if and only if	there exists a y of length $p(n)$ such that A accepts $x\#y$
	if and only if	there exists a y of length $p(n)$ such that C outputs 1 on input y
	if and only if	$C \in CSAT$.

The second equivalence is proved in the correctness proof of Theorem 7.11.

ANALYSIS: M can compute the binary representation of $p(n)$ in space $O(\log n)$. Once this is done, the remainder of the analysis is the same as in Theorem 7.11. □

Given Theorem 8.4, it is reasonably straightforward to prove that SAT is complete for $\mathcal{NP}$. In fact, Cook [4] considered the following much more restrictive version of SAT:

Definition 8.5: A propositional formula is in *conjunctive normal form* if and only if it is the conjunction of "clauses", each of which is the disjunction of "literals", each of which is either a propositional variable or its negation.

Definition 8.6: *3SAT* is the set of satisfiable propositional formulas in conjunctive normal form with at most three literals per clause.

Theorem 8.7 (Cook [4]): *3SAT* is $\leq_m^{\mathcal{L}}$-complete for $\mathcal{NP}$.

Proof:

1. *3SAT* $\in \mathcal{NP}$: Given a formula F with k variables, nondeterministically choose a truth assignment $A \in \{0, 1\}^k$ and accept if and only if A satisfies F. This formula evaluation can be done deterministically in polynomial time (in fact, in ATIME$(\log n)$ [2]).

2. $CSAT \leq_m^{\mathcal{L}} 3SAT$:

CONSTRUCTION: Given a circuit C, the reduction outputs a formula F with one variable x_w for each vertex w of C. F is the conjunction of the following:

- For each OR gate w with inputs u and v, output the clauses

$$(\neg x_u \vee x_w) \wedge (\neg x_v \vee x_w) \wedge (\neg x_w \vee x_u \vee x_v).$$

- For each AND gate w with inputs u and v, output the clauses

$$(\neg x_u \vee \neg x_v \vee x_w) \wedge (\neg x_w \vee x_u) \wedge (\neg x_w \vee x_v).$$

- For each NOT gate w with input u, output the clauses

$$(x_u \vee x_w) \wedge (\neg x_u \vee \neg x_w).$$

- If w is the output gate, output the singleton clause (x_w).

Correctness: We must prove that $C \in CSAT$ if and only if $F \in 3SAT$.

"only if" clause: Suppose C outputs 1 on input $(b_1, b_2, \ldots b_k)$. Then F has a satisfying assignment $A : \{x_w\} \rightarrow \{0, 1\}$, where $A(x_w) = \text{val}(w)$.

"if" clause: Suppose the truth assignment $A : \{x_w\} \rightarrow \{0, 1\}$ satisfies F. Then $\text{val}(w) = A(x_w)$ is consistent and has $\text{val}(w) = 1$ for the output gate w.

Analysis: Space $O(\log n)$ suffices to hold u, v, and w. □

For a detailed discussion of completeness for $\mathcal{NP}$ and a comprehensive list of the diverse problems that are complete for $\mathcal{NP}$, see the book by Garey and Johnson [10].

8.2. Exercises

1. Do Exercise 8.2.

   (Hint: Because a single gate may be the input to more than one other gate, it is insufficient to apply deMorgan's laws in any naive way.)

Chapter 9

The Polynomial Hierarchy

9.1. Complementary Classes

Definition 9.1: $\mathcal{NP} = \bigcup_{c>0} \text{NTIME}(n^c)$.

Definition 9.2: $\mathcal{PSPACE} = \bigcup_{c>0} \text{DSPACE}(n^c) = \bigcup_{c>0} \text{NSPACE}(n^c) = \bigcup_{c>0} \text{ATIME}(n^c)$.

The last two equalities follow from Theorems 4.11 and 4.14.

We then have the following chain of containments:

$$\mathcal{L} \subseteq \mathcal{NL} \subseteq \mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PSPACE}.$$

We know that $\mathcal{NL} \neq \mathcal{PSPACE}$ from the nondeterministic space hierarchy theorem, but we don't know if any of the other containments is proper. This section is devoted to the world between $\mathcal{NP}$ and $\mathcal{PSPACE}$.

Open Problem 9.3: Does $\mathcal{P} = \mathcal{PSPACE}$?

Definition 9.4: If $\mathcal{C}$ is a set of languages over some alphabet Σ , (i.e., $\mathcal{C} \subseteq 2^{\Sigma^*}$), then

$$co\mathcal{C} = \{\Sigma^* - L \mid L \in \mathcal{C}\}.$$

$\mathcal{C}$ is closed under complementation if and only if $\mathcal{C} = co\mathcal{C}$. We already have some results concerning complementary classes, for instance, $\mathcal{P} = co\mathcal{P}$, $\mathcal{NL} = co\mathcal{NL}$ (Theorem 4.23), and $\mathcal{PSPACE} = co\mathcal{PSPACE}$. However, the question for $\mathcal{NP}$ is unresolved:

Open Problem 9.5: Is $\mathcal{NP} = co\mathcal{NP}$?

Notice that this is related, but not equivalent, to Open Problem 4.25 on Page 34.

Proposition 9.6: If $\mathcal{NP} \neq co\mathcal{NP}$ then $\mathcal{P} \neq \mathcal{NP}$.

Proof: $\mathcal{P} = co\mathcal{P}$. $\square$

Aside from the implications of the previous proposition, there are some natural languages in $co\mathcal{NP}$ that justify the study of this class. One such language is the set of propositional tautologies. The set of well formed formulas can be partitioned into those that are tautologies and those that are falsifiable; the latter can be checked in $\mathcal{NP}$ by guessing a truth assignment and checking that it makes the formula false. (The complement of the set of tautologies also includes all strings that are not well formed formulas, but these are easily recognized in polynomial time.) The question of whether $\mathcal{NP} = co\mathcal{NP}$ is the same as asking whether there is a proof system in which every tautology has a polynomial length proof. (See Cook and Reckhow [5] for more information.)

A useful equivalent characterization of $co\mathcal{NP}$ follows from DeMorgan's laws:

Proposition 9.7: $co\mathcal{NP}$ is the set of languages accepted by polynomial time bounded alternating Turing machines that have no existential states.

For instance, to show that the set of tautologies is in $co\mathcal{NP}$, an alternating Turing machine can universally choose all possible truth assignments and verify that each satisfies the input formula.

Another example of a natural problem in $co\mathcal{NP}$ is the set of prime numbers. In fact, this set is in $\mathcal{NP} \cap co\mathcal{NP}$. The fact that it is in $co\mathcal{NP}$ is easy: to determine that x is not prime, nondeterministically guess a proper factor and divide. The proof that the set of primes is in $\mathcal{NP}$ is due to Pratt [36] and is more difficult.

One might wonder if it is possible that $\mathcal{NP} \subseteq co\mathcal{NP}$ without the two classes being equal. The following general proposition shows that this cannot be the case.

Proposition 9.8: If $\mathcal{C} \subseteq co\mathcal{C}$ then $\mathcal{C} = co\mathcal{C}$.

Proof: For any $L \in co\mathcal{C}$, $\Sigma^* - L \in \mathcal{C} \subseteq co\mathcal{C}$. Hence, $L \in \mathcal{C}$. $\square$

9.2. Turing Reducibility

Informally, a language A might be considered to be reducible to a language B whenever there is an efficient algorithm that accepts A, given an efficient subroutine that accepts B. The definition of many-one reducibility given in Definition 5.2 is a very restricted version of this general notion. The more general notion of reducibility is formalized in the definition below of an "oracle Turing machine", where the "oracle" plays the role of the subroutine.

Definition 9.9: An *oracle Turing machine* M is defined as follows: M is a Turing machine with a distinguished query tape, and three distinguished states, $q_?$, q_Y, and q_N. The computation of M depends on some "oracle" language B. Whenever M enters state $q_?$, *in the next step* M will be in state q_Y (q_N) if the nonblank portion of the query tape contains a string that is (respectively, is not) in B. Let M^B denote the machine M with oracle B, and $L(M^B)$ denote the language accepted by this machine.

Definition 9.10: $A \leq_T^{\mathcal{P}} B$ ("A is polynomial time Turing reducible to B") if and only if there is a *deterministic* oracle Turing machine M such that M^B runs in polynomial time and $A = L(M^B)$.

This is the type of reduction Cook [4] used in his original work on complete problems for $\mathcal{NP}$. In his subsequent work, Karp [22] popularized the more restrictive many-one reducibility, which was found to suffice for all the natural reductions among $\mathcal{NP}$-complete problems. For these reasons, polynomial time Turing reducibility is often called "Cook reducibility", and polynomial time many-one reducibility is often called "Karp reducibility". (The latter, denoted $\leq_m^{\mathcal{P}}$, is defined exactly as $\leq_m^{\mathcal{L}}$, except that the reduction is allowed polynomial time.)

A natural question is whether polynomial time Turing reducibility is more powerful than polynomial time many-one reducibility. To shed some light on this question, consider the following example:

Example 9.11: Let SAT ($UNSAT$) be the set of satisfiable (respectively, unsatisfiable) propositional formulas. Then $UNSAT \leq_T^{\mathcal{P}} SAT$ by copying the input formula to the query tape, entering $q_?$, and making q_N the only final state. In contrast, if $UNSAT \leq_m^{\mathcal{P}} SAT$, then $\mathcal{NP} = co\mathcal{NP}$, because $UNSAT$ is $\leq_m^{\mathcal{P}}$–complete for $co\mathcal{NP}$, $\leq_m^{\mathcal{P}}$ is transitive, and $\mathcal{NP}$ is closed under $\leq_m^{\mathcal{P}}$ (all left as exercises).

Open Problem 9.12: Is $\mathcal{NP}$ closed under $\leq_T^{\mathcal{P}}$? If $\mathcal{NP} \neq co\mathcal{NP}$, then Example 9.11 shows that the answer is "no".

Definition 9.13: For any language B, let $\mathcal{P}^B = \{L(M^B) \mid M$ is a deterministic oracle Turing machine such that M^B runs in polynomial time$\} = \{A \mid A \leq_T^{\mathcal{P}} B\}$.

For any language B, let $\mathcal{NP}^B = \{L(M^B) \mid M$ is a nondeterministic oracle Turing machine such that M^B runs in polynomial time$\}$.

Definition 9.14: For any set $\mathcal{C}$ of languages, let

$$\begin{aligned} \mathcal{P}^{\mathcal{C}} &= \bigcup_{B \in \mathcal{C}} \mathcal{P}^B, \text{ and} \\ \mathcal{NP}^{\mathcal{C}} &= \bigcup_{B \in \mathcal{C}} \mathcal{NP}^B. \end{aligned}$$

The following examples will help make these ideas more concrete.

Example 9.15: Does $\mathcal{P}^{\mathcal{NP}} = \mathcal{NP}$? $\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$ because $B \in P^B$ for any language B: simply copy the input onto the query tape and call the oracle. By negating the oracle's answer, it is also true that $co\mathcal{NP} \subseteq \mathcal{P}^{\mathcal{NP}}$. From this, the answer to the question must be "no", unless $\mathcal{NP}=co\mathcal{NP}$. In fact, this question is just a restatement of Open Problem 9.12.

Example 9.16: Does $\mathcal{NP}^{\mathcal{P}} = \mathcal{NP}$? Yes, since we can nondeterministically simulate a specific nondeterministic Turing machine N with an oracle for a language $B \in \mathcal{P}$, pausing in the simulation to answer oracle queries by simulating the oracle and possibly complementing its answer. Since N^B can only make polynomially many queries to B, each on inputs of only polynomial length, and since $B \in \mathcal{P}$, they can all be answered in polynomial time.

9.3. The Polynomial Hierarchy

The polynomial hierarchy was defined by Meyer and Stockmeyer [30], with further properties explored by Stockmeyer [43] and Wrathall [48]. The individual classes of the polynomial hierarchy are the Δ, Σ, and Π sets recursively defined as follows:

Definition 9.17:

$$\Delta_0^P = \Sigma_0^P = \Pi_0^P = \mathcal{P}$$

and, for all nonnegative integers k,

$$\begin{aligned}
\Delta_{k+1}^P &= \mathcal{P}^{\Sigma_k^P}, \\
\Sigma_{k+1}^P &= \mathcal{NP}^{\Sigma_k^P}, \\
\Pi_{k+1}^P &= co\Sigma_{k+1}^P, \text{ and} \\
\mathcal{PH} &= \bigcup_{k \geq 0} \Sigma_k^P.
\end{aligned}$$

Example 9.18: As concrete examples of these definitions, consider the first few levels of the polynomial hierarchy:

$$\begin{aligned}
&\Sigma_1^P = \mathcal{NP}^{\Sigma_0^P} = \mathcal{NP}^{\mathcal{P}} = \mathcal{NP}. \\
&\Pi_1^P = co\Sigma_1^P = co\mathcal{NP}. \\
&\Delta_1^P = \mathcal{P}^{\Sigma_0^P} = \mathcal{P}^{\mathcal{P}} = \mathcal{P}. \\
&\Delta_2^P = \mathcal{P}^{\Sigma_1^P} = \mathcal{P}^{\mathcal{NP}} = \{A \mid A \leq_T^{\mathcal{P}} SAT\}.
\end{aligned}$$

Open Problem 9.19: Does $\mathcal{PH} = \mathcal{PSPACE}$? We will see shortly that $\mathcal{PH} \subseteq \mathcal{PSPACE}$.

Open Problem 9.20: Does $\Sigma_k^P = \Pi_k^P$ or $\Sigma_{k-1}^P = \Sigma_k^P$, for any $k \geq 1$? We will see shortly that a positive answer to either of these questions would have farther-reaching consequences for the polynomial hierarchy. A negative answer to either question would show that $\mathcal{P} \neq \mathcal{PSPACE}$, resolving a major open question.

Proposition 9.21: For all $k \geq 0$,

$$\Sigma_k^P \cup \Pi_k^P \subseteq \Delta_{k+1}^P \subseteq \Sigma_{k+1}^P \cap \Pi_{k+1}^P.$$

(See Figure 9.1.)

Proof:

1. $\Sigma_k^P \subseteq \Delta_{k+1}^P = \mathcal{P}^{\Sigma_k^P}$, because $B \in \mathcal{P}^B$ for any language B.
2. $\Delta_{k+1}^P = \mathcal{P}^{\Sigma_k^P} \subseteq \mathcal{NP}^{\Sigma_k^P} = \Sigma_{k+1}^P$, because $\mathcal{P}^B \subseteq \mathcal{NP}^B$ for any language B.
3. $\Pi_k^P \subseteq \Delta_{k+1}^P \subseteq \Pi_{k+1}^P$ because $\Delta_{k+1}^P = co\Delta_{k+1}^P$.

$\square$

Open Problem 9.22: Are any of the containments in Proposition 9.21 proper?

$$
\begin{array}{ccc}
 & \mathcal{PH} & \\
 & \vdots & \\
 & \Delta_{k+1}^P & \\
/ & & \backslash \\
\Pi_k^P & & \Sigma_k^P \\
\backslash & & / \\
 & \Delta_k^P & \\
 & \vdots & \\
 & \Delta_2^P & \\
/ & & \backslash \\
\Pi_1^P = co\mathcal{NP} & & \Sigma_1^P = \mathcal{NP} \\
\backslash & & / \\
 & \Delta_1^P = \mathcal{P} &
\end{array}
$$

Figure 9.1: The Internal Structure of the Polynomial Hierarchy

9.4. A Sample Problem in $\mathcal{PH}$

In this section the polynomial hierarchy is illustrated further by discussing a sample problem in $\mathcal{PH}$ that does not appear to be in $\mathcal{NP} \cup co\mathcal{NP}$.

Example 9.23 (Papadimitriou [33]): Let *UNIQUE-SAT* be the set of propositional formulas with exactly one satisfying assignment. Although *SAT* is in $\mathcal{NP}$, it does not seem that *UNIQUE-SAT* is in $\mathcal{NP}$. While a nondeterministic Turing machine can find a satisfying assignment for F, it is not clear how it can check that the assignment it found is the only satisfying assignment for F, since this seems to be a universal statement (i.e., *all* other assignments do not satisfy F).

Although not a nondeterministic algorithm, this does give rise to an alternating Turing machine that accepts *UNIQUE-SAT* in polynomial time with a single alternation from existential to universal states. Letting A and B be truth assignments to the variables of F,

$$\mathit{UNIQUE\text{-}SAT} = \{F \mid (\exists A)(\forall B)((A \text{ satisfies } F) \wedge ((B \neq A) \Rightarrow B \text{ does not satisfy } F))\}.$$

The alternating Turing machine existentially chooses a satisfying truth assignment A and universally verifies that no other truth assignment B satisfies F. By Theorem 9.26 below, this implies that *UNIQUE-SAT* is in Σ_2^P.

In fact, *UNIQUE-SAT* is in $\Delta_2^P = \mathcal{P}^{\mathcal{NP}}$: there is a polynomial time deterministic Turing machine M with an oracle for *SAT* that accepts *UNIQUE-SAT*. On input F with variables

$x_1, x_2, \ldots, x_k$, M executes the algorithm in Figure 9.2. Note the many invocations of the oracle for SAT.

if $F \notin SAT$ **then reject**;
comment: Construct a satisfying assignment $A_1 \wedge A_2 \wedge \cdots \wedge A_k$ by "self-reducibility";
for i **from** 1 **to** k **do**
  if $(F \wedge A_1 \wedge A_2 \wedge \cdots \wedge A_{i-1} \wedge x_i) \in SAT$
    then $A_i \leftarrow x_i$
    else $A_i \leftarrow \neg x_i$;
if $(F \wedge \neg(A_1 \wedge A_2 \wedge \cdots \wedge A_k)) \in SAT$
  then reject **comment:** there is at least one other satisfying assignment;
  else accept.

Figure 9.2: Algorithm Demonstrating That $UNIQUE\text{-}SAT \in \Delta_2^P$

9.5. Characterizing Σ_k^P and Π_k^P by Fixed Alternations

We next consider alternating Turing machines with a fixed number of alternations between existential and universal states. These machines provide an alternative characterization of the complexity classes that comprise the polynomial hierarchy.

Definition 9.24: A Σ_{k+1}^P machine is a polynomial time alternating Turing machine that starts in an existential state and, on any path of the computation tree, alternates between existential and universal states at most k times.

Definition 9.25: A Π_{k+1}^P machine is a polynomial time alternating Turing machine that starts in a universal state and, on any path of the computation tree, alternates between universal and existential states at most k times.

Theorem 9.26 (Stockmeyer and Meyer [44, 43]): For any positive integer k,

$$\begin{aligned}\Sigma_k^P &= \{L \mid L \text{ is accepted by some } \Sigma_k^P \text{ machine}\}, \text{ and}\\ \Pi_k^P &= \{L \mid L \text{ is accepted by some } \Pi_k^P \text{ machine}\}.\end{aligned}$$

Proof: The proof is by induction on k.

Basis ($k = 1$):

$$\begin{aligned}\Sigma_1^P = \mathcal{NP} &= \{L \mid L \text{ is accepted by some } \Sigma_1^P \text{ machine}\}, \text{ and}\\ \Pi_1^P = co\mathcal{NP} &= \{L \mid L \text{ is accepted by some } \Pi_1^P \text{ machine}\}.\end{aligned}$$

Induction ($k \geq 1$):

1. $\Sigma_{k+1}^P \subseteq \{L \mid L \text{ is accepted by some } \Sigma_{k+1}^P \text{ machine}\}$:

By definition, $\Sigma_{k+1}^P = \mathcal{NP}^{\Sigma_k^P}$. Let $L \in \Sigma_{k+1}^P$ be accepted by a nondeterministic oracle Turing machine M that runs in some polynomial $p_1(n)$ time with an oracle for $B \in \Sigma_k^P$. We want to simulate M by a Σ_{k+1}^P machine A. The problem is that M may query its oracle a polynomial number of times, and simulating each of these queries requires some alternations.

We will show how to simulate M so that all oracle calls along any computation path are postponed until the end of that computation path. A simulates M, except that when M calls the oracle with query y_i, A guesses $b_i \in \{Y, N\}$ and records the pair (y_i, b_i) on a separate worktape. A then enters state q_{b_i} and continues the simulation of M. If M rejects, A rejects. If M accepts, then A must verify all guesses b_i before accepting.

The series of guesses left on the worktape are pairs of the form (y_i, b_i) and correspond to verifying that $y_i \in B$ if $b_i = Y$, and $y_i \in \Sigma^* - B$ if $b_i = N$. By the induction hypothesis, B is accepted by some Σ_k^P machine M_B and $\Sigma^* - B$ is accepted by some Π_k^P machine M_{Σ^*-B}, each running in some polynomial $p_2(n)$ time. It would be simple if A could now universally verify these guesses, but those with $b_i = Y$ would introduce one too many alternations. Instead, A first makes one pass over the worktape containing the pairs (y_i, b_i) and does the following:

- If $b_i = N$, A replaces the pair (y_i, b_i) by the initial configuration of M_{Σ^*-B} on input y_i.
- If $b_i = Y$, A simulates M_B on input y_i until M_B reaches its first universal configuration Q, and replaces the pair (y_i, b_i) by Q.

Until this point, A has used no universal configurations. A now universally verifies that each configuration remaining on the tape is accepting by simulating M_B or M_{Σ^*-B}, as appropriate. This takes at most another $k-1$ alternations, since these are configurations accepted by Π_k^P machines.

The running time of this simulation is $O((p_1(n))^2 + p_2(p_1(n)))$, which is also polynomial, as follows. M may query its oracle at most $p_1(n)$ times on queries y_i each of length at most $p_1(n)$. For each, it may take time $O(p_1(n))$ to record y_i, and time $O(p_2(p_1(n)))$ to simulate M_B or M_{Σ^*-B} on input y_i.

2. $\Sigma_{k+1}^P \supseteq \{L \mid L \text{ is accepted by some } \Sigma_{k+1}^P \text{ machine}\}$:

Let L be accepted by some Σ_{k+1}^P machine A. Let $B = \{Q \mid Q$ is a universal configuration of A that leads to acceptance in at most $k-1$ alternations$\}$. B is accepted by a Π_k^P machine that simulates A starting at Q so, by the induction hypothesis, $B \in \Pi_k^P$.

We construct a nondeterministic oracle Turing machine M such that $L = L(M^B)$. M simulates A until A enters its first universal configuration Q, and then queries its oracle to determine if $Q \in B$, accepting if and only if this is so. The running time of M is less than the running time of A, so it is certainly polynomial.

3. $\Pi_{k+1}^P = \{L \mid L \text{ is accepted by some } \Pi_{k+1}^P \text{ machine}\}$:

$$\begin{aligned}
\Pi_{k+1}^P &= co\Sigma_{k+1}^P \\
&= \{\Sigma^* - L \mid L \text{ is accepted by some } \Sigma_{k+1}^P \text{ machine}\} \\
&= \{L \mid L \text{ is accepted by some } \Pi_{k+1}^P \text{ machine}\}.
\end{aligned}$$

The last equality follows by interchanging existential and universal states, and applying deMorgan's laws. □

One corollary of Theorem 9.26 provides a useful normal form for nondeterministic oracle Turing machines:

Corollary 9.27: $L \in \Sigma_{k+1}^P$ if and only if $L = L(M^B)$, where $B \in \Pi_k^P$ and M is a nondeterministic oracle Turing machine that runs in polynomial time, queries B once, and accepts if and only if the answer is yes.

$L \in \Pi_{k+1}^P$ if and only if $L = L(M^B)$, where $B \in \Sigma_k^P$ and M is a "co-nondeterministic" oracle Turing machine (i.e., no existential states) that runs in polynomial time, queries B once, and accepts if and only if the answer is yes.

Proof: This restricted form of oracle Turing machine is exactly what arises in part 2 of the proof of Theorem 9.26. □

Note the similarity between this normal form for nondeterministic oracle Turing machines and many-one reductions. The only difference is that in Corollary 9.27 the reduction is nondeterministic.

Corollary 9.28: $\mathcal{PH} \subseteq \mathcal{PSPACE}$.

Proof: $\mathcal{PH}$ is the subset of $\mathcal{PSPACE}$ obtained by fixing the number of alternations of a polynomial time alternating Turing machine. □

9.6. How to Collapse the Polynomial Hierarchy

Theorem 9.29 (Stockmeyer [43]): If $\Sigma_k^P = \Pi_k^P$ for any $k \geq 1$, then $\Sigma_j^P = \Pi_j^P = \Sigma_k^P$ for all $j \geq k$.

Proof: The proof is by induction on j.

Basis ($j = k$): Vacuous.

Induction ($j \geq k$): Assume by the induction hypothesis that $\Sigma_j^P = \Pi_j^P = \Sigma_k^P$.

1. $\Sigma_{j+1}^P \subseteq \Sigma_k^P$: Let $L \in \Sigma_{j+1}^P$. By Corollary 9.27, $L = L(M^B)$ where $B \in \Pi_j^P$, and M is a polynomial time nondeterministic oracle Turing machine that makes one query "$y \in B$?", and accepts if and only if the answer is yes. By the induction hypothesis, $B \in \Sigma_k^P$, so B is accepted by some Σ_k^P machine A by Theorem 9.26. Then L is also accepted by some Σ_k^P machine, as follows. Simulate M until it enters state $q_?$ with y on its query tape, and then simulate A on y. Since M is nondeterministic and A is a Σ_k^P machine, there are only $k-1$ alternations. Since M and A each run in polynomial time and y has polynomial length, the total time is polynomial. Therefore, by Theorem 9.26 again, $L \in \Sigma_k^P$.

2. $\Pi_{j+1}^P \subseteq \Sigma_k^P$: $\Pi_{j+1}^P = co\Sigma_{j+1}^P \subseteq co\Sigma_k^P = \Pi_k^P = \Sigma_k^P$. □

The following corollary shows that separating $\mathcal{PH}$ from $\mathcal{P}$ would be sufficient to prove $\mathcal{P} \neq \mathcal{NP}$.

Corollary 9.30: $\mathcal{P} \neq \mathcal{PH}$ if and only if $\mathcal{P} \neq \mathcal{NP}$.

Proof:

"if" clause: $\mathcal{P} \subseteq \mathcal{NP} \subseteq \mathcal{PH}$, so $\mathcal{P} \neq \mathcal{NP}$ implies $\mathcal{P} \neq \mathcal{PH}$.

"only if" clause: If $\mathcal{P} = \mathcal{NP}$ then $\Sigma_1^P = \mathcal{NP} = \mathcal{P} = co\mathcal{P} = co\mathcal{NP} = \Pi_1^P$. By Theorem 9.29, then, $\mathcal{PH} = \Sigma_1^P = \mathcal{NP} = \mathcal{P}$. □

9.7. Relating Circuits to $\mathcal{NP}$

Having shown in Theorem 7.17 that $\mathcal{P} \subseteq \mathcal{PSIZE}$, and in Exercise 7.20 that $\mathcal{P} \neq \mathcal{PSIZE}$, a natural question to ask is whether $\mathcal{NP} \subseteq \mathcal{PSIZE}$. In this section we will see that if the answer is positive, then the polynomial hierarchy collapses to $\Sigma_2^P \cap \Pi_2^P$. Therefore, such a containment seems unlikely.

Definition 9.31: $\mathit{SAT\text{-}CIRCUIT} = \{C \mid C$ has m inputs and accepts the set of length m encodings of satisfiable formulas$\}$.

Definition 9.32: If F is a propositional formula and $A_1, A_2, \ldots, A_i$ are literals of distinct variables of F, then $F \mid (A_1, A_2, \ldots, A_i)$ denotes F with the substitutions $A_1 = A_2 = \cdots = A_i = 1$ and $\neg A_1 = \neg A_2 = \cdots = \neg A_i = 0$.

Any standard binary encoding of formulas will suffice for Definition 9.31, provided that the encoding of $F \mid (A_1, A_2, \ldots, A_i)$ is no longer than the encoding of F. For example, it suffices if the constants 0 and 1 have shorter encodings than any literal. The method must also be capable of padding out such shorter encodings to m bits.

Lemma 9.33: $\mathit{SAT\text{-}CIRCUIT} \in co\mathcal{NP}$.

Proof: To avoid later confusion, note that three different levels of computing devices are involved in the proof: first there is the $co\mathcal{NP}$ machine whose input is the encoding of a circuit; this circuit itself has as input the encoding of a formula, which in turn has as input a truth assignment to its variables.

CONSTRUCTION: On input C, a circuit with m inputs, construct a $co\mathcal{NP}$ machine M that executes the algorithm in Figure 9.3.

CORRECTNESS: The only part of the correctness that is not immediate from the construction is the self-reducibility segment. If $A_1 = A_2 = \cdots = A_k = 1$ satisfies F, then C has correctly accepted F as satisfiable, and M should accept. Assume, then, that $A_1 = A_2 = \cdots = A_k = 1$ does not satisfy F, yet $C \in \mathit{SAT\text{-}CIRCUIT}$. Let i be any value such that $F \mid (A_1, A_2, \ldots, A_{i-1})$ is satisfiable, but $F \mid (A_1, A_2, \ldots, A_{i-1}, A_i)$ is not; there must be such an i, since C (correctly) output 1 on input F, but $F \mid (A_1, A_2, \ldots, A_k)$ is false. If $A_i = x_i$, then C erroneously stated that $F \mid (A_1, A_2, \ldots, A_{i-1}, A_i)$ is satisfiable. If $A_i = \neg x_i$, then both $F \mid (A_1, A_2, \ldots, A_{i-1}, x_i)$ and $F \mid (A_1, A_2, \ldots, A_{i-1}, \neg x_i)$ are unsatisfiable, so that C erroneously stated that $F \mid (A_1, A_2, \ldots, A_{i-1})$ is satisfiable. In either case, $C \notin \mathit{SAT\text{-}CIRCUIT}$, so M should reject.

ANALYSIS: There are five lines in Figure 9.3 marked with asterisks, in which M must solve a circuit value or formula value problem. By Theorem 7.11, each can be done deterministically in polynomial time. □

Theorem 9.34 (Karp and Lipton [23]): If $\mathcal{NP} \subseteq \mathcal{PSIZE}$ then $\mathcal{PH} = \Sigma_2^P \cap \Pi_2^P$.

Proof: Suppose some family of circuits C_n of polynomial size $p_1(n)$ accepts SAT. We will show that $\Pi_2^P \subseteq \Sigma_2^P$, and the theorem then follows from Proposition 9.8 and Theorem 9.29.

universally choose $F \in \{0,1\}^m$;
comment: verify that C behaves correctly on every input F ;
if F does not encode a propositional formula with some number k of variables
  then
    if C outputs 0 on input F *
      then accept
      else reject
  else
    if C outputs 0 on input F *
      then begin
        universally choose truth assignment $A \in \{0,1\}^k$;
        if A falsifies F *
          then accept
          else reject
      end
      else begin
        comment: use self-reducibility to verify that F is satisfiable ;
        for i **from** 1 **to** k **do**
          if C outputs 1 on input $F \mid (A_1, A_2, \ldots, A_{i-1}, x_i)$ *
            then $A_i \leftarrow x_i$
            else $A_i \leftarrow \neg x_i$;
        if $A_1 = A_2 = \cdots = A_k = 1$ satisfies F *
          then accept
          else reject
      end .

Figure 9.3: A $co\mathcal{NP}$ Algorithm That Accepts *SAT-CIRCUIT*

CONSTRUCTION: Let $L \in \Pi_2^P$. By Corollary 9.27, $L = L(M^{SAT})$, where M is a "co-nondeterministic" oracle Turing machine (i.e., no existential states) that runs in polynomial time $p_2(n)$ and makes one call on an oracle for *SAT*. Construct a Σ_2^P machine A that accepts L as described below, and then apply Theorem 9.26.

On input x, A does the following:

existentially choose the encoding of a circuit C ;
universally choose $b \in \{0,1\}$;
  case b **of**
    0: **if** $C \in$ *SAT-CIRCUIT* **then accept else reject**;
    1: simulate M on input x using C to answer the oracle query ;
       if C does not have enough inputs to accommodate the query **then reject**;
       if M accepts x **then accept else reject**
  end .

Correctness: If M^{SAT} accepts x, then some choice of C will be in *SAT-CIRCUIT*, and will have enough inputs to accommodate M's query. For this choice of C, A will accept x. Conversely, if A accepts x, then $C \in$ *SAT-CIRCUIT* and hence answers the oracle query correctly. Hence, M^{SAT} accepts x.

Analysis: By Lemma 9.33, *SAT-CIRCUIT* $\in co\mathcal{NP}$, so there is only one alternation. Let $p_3(n)$ be the running time of the $co\mathcal{NP}$ machine of Lemma 9.33, and $p_4(n)$ be the time to solve the circuit value problem on inputs of length n. M can only create a query of length $p_2(n)$, so it suffices if C has $p_2(n)$ inputs and, by hypothesis, $p_1(p_2(n))$ size. Such a circuit has an encoding of length $p(n) = O(p_1(p_2(n)) \log p_1(p_2(n)))$. Therefore, A runs in time $O(p(n)+p_3(p(n))+p_2(n)+p_4(p(n)))$, a polynomial. $\square$

9.8. Exercises

1. If $\mathcal{C}$ is a set of languages over some alphabet Σ and L is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{C}$, prove that $\Sigma^* - L$ is $\leq_m^{\mathcal{L}}$–complete for $co\mathcal{C}$.

2. Prove that $\leq_m^{\mathcal{P}}$ is transitive.

3. Prove that $\leq_T^{\mathcal{P}}$ is transitive.

4. Prove that $\mathcal{NP}$ is closed under $\leq_m^{\mathcal{P}}$.

5. Prove that the following statements are equivalent:

   (a) $\mathcal{NP} = co\mathcal{NP}$.

   (b) Some language that is $\leq_m^{\mathcal{P}}$–complete for $\mathcal{NP}$ is in $co\mathcal{NP}$.

   (c) Some language that is $\leq_T^{\mathcal{P}}$–complete for $\mathcal{NP}$ is in $co\mathcal{NP}$.

6. What is wrong with the following proof that $\mathcal{P} \neq \mathcal{NP}$?

   For any two languages A and B in $\mathcal{P}$, A is reducible to B by a polynomial time reduction, since the reduction can solve membership in A without even using B. But $SAT \not\leq_m^{\mathcal{P}} \emptyset$, where $\emptyset$ is the empty set, since, for any satisfiable formula F and any reduction f, it is not true that $F \in SAT$ if and only if $f(F) \in \emptyset$. Since $\emptyset$ is obviously in $\mathcal{P}$, $SAT \notin \mathcal{P}$.

Chapter 10

Polynomial Space

10.1. Problems Complete for $\mathcal{PSPACE}$

In this section we consider a problem analogous to *SAT* that is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{PSPACE}$. The proofs are almost identical to those in Section 8.1.

Definition 10.1: A *quantified Boolean formula* has the form

$$(\exists y_1)(\forall y_2)(\exists y_3)\cdots(\forall y_{2k})\ F(y_1, y_2, \ldots, y_{2k}),$$

where $F(y_1, y_2, \ldots, y_{2k})$ is a propositional formula with variables $y_1, y_2, \ldots, y_{2k}$. The quantification is over $y_i \in \{0,1\}$, for all $1 \leq i \leq 2k$.

Definition 10.2: *QBF* is the set of true quantified Boolean formulas, where the propositional part F is in conjunctive normal form with at most three literals per clause.

Notice that *3SAT* is the same as *QBF*, except that all quantifiers are existential in *3SAT*. This analogy suggests that *QBF* should be complete for alternating polynomial time, i.e., $\mathcal{PSPACE}$.

Theorem 10.3 (Stockmeyer and Meyer [44, 43]): *QBF* is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{PSPACE}$.

Proof:

1. $QBF \in \mathcal{PSPACE}$: It suffices to show that $QBF \in \mathrm{ATIME}\left(n^{O(1)}\right)$. Given a quantified Boolean formula Q with $2k$ variables, use the alternation to mimic the quantifiers of Q while choosing and recording a truth assignment $A \in \{0,1\}^{2k}$. Deterministically evaluate Q at this assignment A, and accept if and only if Q evaluates to true.

2. *QBF* is $\leq_m^{\mathcal{L}}$–hard for $\mathcal{PSPACE}$: Let A be an alternating Turing machine that runs in polynomial time. The reduction of $L(A)$ to *QBF* is exactly as in Theorems 8.4 and 8.7, with the following changes:

1. In the proof of Theorem 8.4, assume without loss of generality that A alternates between existential and universal states each step, adding dummy states if necessary. A is then put

in the normal form of alternating when writing y, and then simulating some deterministic Turing machine D on input $x\#y$. The language to which $L(A)$ is reduced is the "quantified circuit" problem:

$$\{C \mid (\exists b_1)(\forall b_2)(\exists b_3)\cdots(\forall b_{2k})(C \text{ outputs } 1 \text{ on input } (b_1, b_2, \ldots, b_{2k}))\}.$$

2. In the reduction of Theorem 8.7, add a quantifier $(\exists x_w)$ to the end of the quantifier list presented in the quantified circuit input, for each gate w. In order to preserve the alternation of quantifiers, add dummy universally quantified variables.

□

Using similar techniques, Meyer and Stockmeyer [30, 43] presented analogous problems complete for each Σ_k^P and Π_k^P.

The richest source of natural problems complete for $\mathcal{PSPACE}$ has come from two-person games, and was exposed by Schaefer [42]. For a large variety of games, he showed that the problem of deciding if Player 1 has a winning strategy, starting from some given configuration of the game, is $\leq_m^{\mathcal{L}}$-complete for $\mathcal{PSPACE}$. To see why this might be expected, notice that the alternation inherent in these questions may be expressed informally as follows: Is there a move for Player 1 such that, for all next moves by Player 2, there exists a next move by Player 1 such that ... the result is a configuration in which Player 1 has won the game?

An example of one of Schaefer's games is "generalized geography", which is played on a directed graph G with a distinguished start vertex s, as follows. Player 1 begins with s as the current vertex. The players take turns removing any edge (u, v) emanating from the current vertex u, after which v becomes the current vertex. The first player with no move remaining loses.

This game generalizes the children's game of "geography", in which there is a vertex for each of the 26 letters of the alphabet, and an edge (u, v) for every country of the world that begins with the letter u and ends with the letter v.

Definition 10.4: *GEO* is the set of pairs (G, s) such that Player 1 has a winning generalized geography strategy on G starting at s.

Theorem 10.5 (Schaefer [42]): *GEO* is $\leq_m^{\mathcal{L}}$-complete for $\mathcal{PSPACE}$.

Finally, the existence of problems that are complete for $\mathcal{PSPACE}$ makes it very unlikely that $\mathcal{PSPACE} = \mathcal{PH}$:

Theorem 10.6: If $\mathcal{PSPACE} = \mathcal{PH}$, then $\mathcal{PH} = \Sigma_k^P$ for some constant k.

Proof: If $\mathcal{PSPACE} = \mathcal{PH}$, then $QBF \in \Sigma_k^P$ for some k. Since QBF is $\leq_m^{\mathcal{L}}$-hard for $\mathcal{PSPACE}$ and Σ_k^P is closed under $\leq_m^{\mathcal{L}}$, $\mathcal{PH} = \mathcal{PSPACE} \subseteq \Sigma_k^P$. □

10.2. A Lower Bound for Problems Complete for $\mathcal{PSPACE}$

One feature of problems complete for $\mathcal{PSPACE}$ is that we can prove an interesting unconditional lower bound on their space complexity. "Unconditional" here means that the theorem does not rely on any unproven hypothesis such as $\mathcal{P} \neq \mathcal{NP}$.

Theorem 10.7: If L is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{PSPACE}$ then, for some $\delta > 0$, $L \notin \mathrm{DSPACE}(n^\delta)$.

Proof: By Theorem 3.8, there is some language $A \in \mathcal{PSPACE} - \mathrm{DSPACE}(n)$. Since L is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{PSPACE}$, there is some deterministic Turing machine F that reduces A to L and runs in $O(\log n)$ space. Let f be the function computed by F. Because F runs in $O(\log n)$ space, there is a constant $k > 0$ such that $|f(x)| \leq |x|^k$ for all sufficiently long x.

Now suppose that $\delta > 0$ is any constant such that L is accepted by some deterministic Turing machine D that runs in space $O(n^\delta)$. By using the method in the proof of Lemma 5.3, the deterministic Turing machines F and D can be composed to produce a deterministic Turing machine that accepts A in space $O(\log|x| + \log|f(x)| + |f(x)|^\delta) = O(\log n + \log(n^k) + (n^k)^\delta) = O(n^{k\delta})$. Since, by the definition of A, $A \notin \mathrm{DSPACE}(n)$, it must be the case that $\delta > 1/k$; for any $\delta \leq 1/k$, then, $L \notin \mathrm{DSPACE}(n^\delta)$. □

A similar lower bound holds, in fact, for nondeterministic Turing machines:

Corollary 10.8: If L is $\leq_m^{\mathcal{L}}$–complete for $\mathcal{PSPACE}$ then, for some $\delta > 0$, $L \notin \mathrm{NSPACE}(n^\delta)$.

Proof: By Corollary 4.15, if $L \notin \mathrm{DSPACE}(n^\delta)$, then $L \notin \mathrm{NSPACE}(n^{\delta/2})$. □

Bibliography

[1] R. Anderson, E. Mayr, and M. Warmuth. Parallel approximation algorithms for bin packing. *Information and Computation*, 82(3):262–277, 1989.

[2] S. R. Buss. The Boolean formula value problem is in ALOGTIME. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 123–131, New York, NY, May 1987.

[3] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, Jan. 1981.

[4] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, OH, May 1971.

[5] S. A. Cook and R. A. Reckhow. On the lengths of proofs in the propositional calculus. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pages 135–148, Seattle, WA, Apr.-May 1974.

[6] S. A. Cook and R. Sethi. Storage requirements for deterministic polynomial time recognizable languages. *Journal of Computer and System Sciences*, 13(1):25–37, 1976.

[7] D. Dobkin, R. J. Lipton, and S. Reiss. Linear programming is log-space hard for *P*. *Information Processing Letters*, 8(2):96–97, Feb. 1979.

[8] C. Dwork, P. C. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.

[9] P. W. Dymond and M. Tompa. Speedups of deterministic machines by synchronous parallel machines. *Journal of Computer and System Sciences*, 30(2):149–161, Apr. 1985.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[11] L. M. Goldschlager. The monotone and planar circuit value problems are log space complete for *P*. *SIGACT News*, 9(2):25–29, Summer 1977.

[12] L. M. Goldschlager, R. A. Shaw, and J. Staples. The maximum flow problem is log space complete for *P*. *Theoretical Computer Science*, 21:105–111, 1982.

[13] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo. A compendium of problems complete for *P*. In progress.

[14] J. Hartmanis, P. M. Lewis, II, and R. E. Stearns. Hierarchies of memory limited computations. In *Conference Record on Switching Circuit Theory and Logical Design*, pages 179–190, Ann Arbor, MI, 1965.

[15] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306, 1965.

[16] F. C. Hennie. One-tape off-line Turing machine computations. *Information and Control*, 8(6):553–578, 1965.

[17] F. C. Hennie and R. E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the ACM*, 13(4):533–546, 1966.

[18] J. Hopcroft, W. J. Paul, and L. G. Valiant. On time versus space. *Journal of the ACM*, 24(2):332–337, Apr. 1977.

[19] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, Oct. 1988.

[20] N. D. Jones and W. T. Laaser. Complete problems for deterministic polynomial time. *Theoretical Computer Science*, 3(2):105–117, 1976.

[21] N. D. Jones, Y. E. Lien, and W. T. Laaser. New problems complete for nondeterministic log space. *Mathematical Systems Theory*, 10:1–17, 1976.

[22] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, New York, 1972.

[23] R. M. Karp and R. J. Lipton. Some connections between nonuniform and uniform complexity classes. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 302–309, Los Angeles, CA, Apr. 1980.

[24] L. G. Khachian. A polynomial time algorithm for linear programming. *Doklady Akademii Nauk SSSR, n.s.*, 244(5):1093–1096, 1979. English translation in *Soviet Math. Dokl. 20*, 191–194.

[25] D. E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, 8(2):18–24, Apr.–June 1976.

[26] S. Y. Kuroda. Classes of languages and linear-bounded automata. *Information and Control*, 7:207–233, 1964.

[27] R. E. Ladner. The circuit value problem is log space complete for *P*. *SIGACT News*, 7(1):18–20, Jan. 1975.

[28] T. Lengauer and K. W. Wagner. The correlation between the complexities of the nonhierarchical and hierarchical versions of graph problems. In *4th Annual Symposium on Theoretical Aspects of Computer Science*, pages 100–113, 1987.

[29] R. J. Lipton, L. Snyder, and Y. Zalcstein. The complexity of word and isomorphism problems for finite groups. In *Conference on Information Science and Systems*, pages 33–35. Johns Hopkins University, 1976.

[30] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular sets with squaring requires exponential space. In *13th Annual Symposium on Switching and Automata Theory*, pages 125–129, University of Maryland, Oct. 1972.

[31] B. Monien and I. H. Sudborough. On eliminating nondeterminism from Turing machines which use less than logarithm worktape space. In H. A. Maurer, editor, *Automata, Languages, and Programming: Sixth Colloquium*, volume 71 of *Lecture Notes in Computer Science*, pages 431–445, Graz, Austria, July 1979. Springer-Verlag.

[32] N. Nisan. $RL \subseteq SC$. Preprint, Hebrew University, Mar. 1991.

[33] C. H. Papadimitriou. On the complexity of unique solutions. In *23rd Annual Symposium on Foundations of Computer Science*, pages 14–20, Chicago, IL, Nov. 1982. IEEE.

[34] W. J. Paul, N. Pippenger, E. Szemerédi, and W. T. Trotter. On determinism versus non-determinism and related problems. In *24th Annual Symposium on Foundations of Computer Science*, pages 429–438, Tucson, AZ, Nov. 1983. IEEE.

[35] N. Pippenger and M. J. Fischer. Relations among complexity measures. *Journal of the ACM*, 26(2):361–381, Apr. 1979.

[36] V. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4:214–220, 1975.

[37] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20:229–234, 1985.

[38] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key crypto-systems. *Communications of the ACM*, 21:120–126, 1978.

[39] W. L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):365–383, June 1981.

[40] W. J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[41] W. J. Savitch. Nondeterministic $\log n$ space. In *Proceedings of the 8th Annual Princeton Conference on Information Sciences and Systems*, pages 21–23, Princeton University, 1974.

[42] T. J. Schaefer. On the complexity of some two-person perfect-information games. *Journal of Computer and System Sciences*, 16:185–225, 1978.

[43] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.

[44] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pages 1–9, Austin, TX, April-May 1973.

[45] R. Szelepcsényi. The method of forcing for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[46] M. Tompa. An extension of Savitch's theorem to small space bounds. *Information Processing Letters*, 12(2):106–108, Apr. 1981.

[47] M. Tompa. Two familiar transitive closure algorithms which admit no polynomial time, sublinear space implementations. *SIAM Journal on Computing*, 11(1):130–137, Feb. 1982.

[48] C. Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3:23–33, 1977.