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 $T(n) \geq \log_2 n$,

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

Proof: The four containments $\text{DTIME}(T(n)) \subseteq \text{NTIME}(T(n)) \subseteq \text{ATIME}(T(n))$ and $\text{DSPACE}(T(n)) \subseteq \text{NSPACE}(T(n)) \subseteq \text{ASPACE}(T(n))$ 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 $\text{DTIME}(T(n)) \subset \text{NTIME}(T(n))$. For $T(n) \geq n$, Theorem 3.12 shows that $\text{DTIME}(T(n)) \subset \text{DTIME}\left(2^{T(n)}\right)$. □

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 $\text{DTIME}(T(n)) \neq \text{NTIME}(T(n)) \neq \text{ATIME}(T(n))$ for all $T(n) < n$. Paul, Pippenger, Szemerédi, and Trotter [34] have shown that $\text{DTIME}(T(n)) \neq \text{NTIME}(T(n))$ 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. □

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. □

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. □

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 $P \vdash_{A,x} Q$. 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 $T(n) \geq n$,

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

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 $\not\flat$ 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** $(1 \le j \le n$ **and** $b = x_j)$ **or** $(((j < 0)$ **or** $(j > n))$ **and** $b = \not\flat)$
    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 $0 \le i \le t$ and $-2t \le j \le 3t$, 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, $T(n) \ge n$.

□

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 $T(n) \ge \log n$ purposely omitted: if A only writes small indices on its index

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 $P \vdash^{d}_{M,x} Q$ 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 $P \vdash^{\leq d}_{M,x} Q$ if and only if $P \vdash^{c}_{M,x} Q$ for some $c \leq d$.

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

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

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 $P \vdash^{\leq 2^k}_{N,x} Q$. Let R be a "midpoint" configuration in this computation, that is, $P \vdash^{\leq 2^{k-1}}_{N,x} R$ and $R \vdash^{\leq 2^{k-1}}_{N,x} Q$. 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, $P \vdash^{\leq 2^{k-1}}_{N,x} R$ and $R \vdash^{\leq 2^{k-1}}_{N,x} Q$. Thus, $P \vdash^{\leq 2^k}_{N,x} Q$.

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,

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

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

  We know $\text{DSPACE}(T(n)) \subseteq \text{NSPACE}(T(n)) \subseteq \text{DSPACE}((T(n))^2)$, 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 $\text{DSPACE}(S(n))$ 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 $\text{NSPACE}(n)$ closed under complementation? He was interested in this because the class of context-sensitive languages is exactly $\text{NSPACE}(n)$ 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 $P_0 \overset{\leq k}{\underset{M,x}{\vdash}} P$. Here are the details:

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