Chapter 3

Space Bounded Computation

Chapter Outline: Configuration graphs and the graph accessibility problem. Savitch's Theorem. Immerman-Szelepcsenyi Theorem.

3.1 Configuration Graphs

We begin this chapter by discussing configuration graphs. They are very useful in conceptualizing the formal process of computation, and often realize this formal process in terms of concrete graph theoretic questions. They are particularly useful in studying space bounded computations.

Let M be a non-deterministic TM and x be an input of length n. At any point during the computation, the *configuration* of M, which determines all the relevant future steps of the computation, can be fully described by specifying the current state, the contents of the tape, and the current position of the tape-head. If the machine M is time $T(n)$ bounded, then this information takes no more than $O(T(n))$ bits. For space bounded computation we use off-line TMs. Let M be a $S(n)$ space bounded machine. Note that for space bounded machine the input tape is read only and its content does not change during the computation. Then we can specify the *configuration* of M at any step by the current state ($O(1)$ bits), the position of the read-only head ($\log n$ bits), and the contents of the work tape together with its tape-head ($O(S(n))$ bits). If $S(n) = \Omega(\log n)$, as we will always assume in space bounded computations in these Lectures, we can describe a configuration using $O(S(n))$ bits.

Consider the configuration in which the tape content is the input x (if M is not an off-line TMs, then this x is followed by blank symbols of a combined length equal to the time bound), the position of tape-head is 0, and the current state is the start state of M. This is the (unique) *starting configuration*. Also naturally, any configuration in which the current state is a final state of M is called a *final configuration*. Technically TM computation is defined in terms of these configurations. One defines a binary relation on pairs of configurations:

$c \vdash c'$ iff in one step c can be followed by c', i.e., there is a legal move in one step from c to c'. For deterministic machines, there can be at most one next configuration after any c; for non-deterministic machines there can be more than one. The number of such next valid configurations is completely specified by the transition function δ, and thus there can be at most a constant number of them. This constant depends only on M, and not on the input x. We can extend these notions to Turing machines that use multiple tapes.

In either model the total number of possible configurations is $2^{O(f(n))}$, if f is the time or space bound. Given an input x, we can construct a directed graph with these configurations as nodes. We add an edge from configuration c to c', if $c \vdash c'$. We then mark the start and final configurations on this graph. We call this graph the *configuration graph* of $M(x)$. Notice that whether there is an edge from c to c' can be determined by examining the given configurations and the transition rules of the machine. Given any two configurations c and c', this can be checked out easily, in time linear in $|c| + |c'|$, and in space $O(\log(|c| + |c'|))$. It is easy to see

PROPOSITION 3.1 *Let M be a non-deterministic Turing machine and x be an input. M accepts x iff at least one final node is reachable from the start node in the configuration graph of $M(x)$.*

THEOREM 3.2 *For any function $f(n) = \Omega(\log n)$, $\mathrm{NSPACE}[f(n)] \subseteq \mathrm{DTIME}[2^{O(f(n))}]$*

Proof. Let M be a non-deterministic machine that uses $O(f(n))$ space and x be the input. To simulate the computation of M on x deterministically, we first construct the configuration graph G. We then check if there is path from the start configuration to any one of the final configuration. We can do this by depth first search. As the number of nodes is $2^{O(f(n))}$, we can do this in time $2^{O(f(n))}$. The correctness is ensured by Proposition 3.1. ♣

3.2 Savitch's Theorem

We next establish an important connection between non-deterministic and deterministic space. We shall show that any non-deterministic computation can be simulated in deterministic space with only a quadratic blow-up in space usage. All the formalism of computation and non-determinism aside, the crux of the matter is the following statement on a graph theoretic problem—The Graph Accessibility Problem (GAP).

THEOREM 3.3 (SAVITCH) *Given a directed graph $G = (V, E)$ over n nodes and two nodes s and t, we can test whether there is a path from s to t in $O((\log n)^2)$ deterministic space.*

Remark: Notice that the usual depth first search will use $\Omega(n)$ space. For each node visited, DFS has to mark that it has been visited. Thus it needs at least n bits of space. Of course DFS shows that GAP is in P.

Proof. Instead of DFS, we make the following trivial observation: There is a path from u to v with length at most t iff there is some w, and there are paths from u to w and w to v of length at most $\lceil t/2 \rceil$ and $\lfloor t/2 \rfloor$ respectively. So we will use a recursive "middle-first" search.

First of all, if t can be reached from s, then it can be done so using a path of length at most n (actually $n-1$ suffices; for any path we can remove any cycle, and thus removing all repeated appearances of any vertex along the path.) Then we apply the above observation. We go through all possible choices for a potential middle w. We check the existence of the required two paths of length $n/2$ recursively. Given two nodes x and y, and integer t (in binary), the following algorithm checks if there is a path of length at most t from u to v.

REACH(u, v, t)
    If $t = 1$ then
        If [$(u = v)$ or $(\langle u, v \rangle \in E)$] then *accept*
        else *reject*
    If $t > 1$ then
        For each node x in G,
            If [REACH(u, x, $t/2$) $\wedge$ REACH(x, v, $t/2$)] then *accept*
    If haven't yet accepted, *reject*.

We can check if t can be reached from s by calling REACH(s, t, n). One subtle issue is that n has to be a power of 2 for our recursion to work well. If n is not a power of 2, we call REACH(s, t, n'), where $n' = 2^{\lceil \log n \rceil}$.

The algorithm has a recursion depth of $\lceil \log n \rceil$. Thus the recursion stack has to store information about $\log n$ recursive calls. For each call, we need to store u, v, t and x used in that call. This needs $O(\log n)$ space. Thus the total space used is $O((\log n)^2)$. ♣

We use the above theorem to prove

THEOREM 3.4 (SAVITCH) *For any space constructible function $S(n) = \Omega(\log n)$,* NSPACE$[S(n)] \subseteq$ DSPACE$[(S(n))^2]$.

Proof. Let M be a non-deterministic machine that uses $O(S(n))$ space. We construct a deterministic machine M' that uses $O((S(n))^2)$ space to simulate M. Given input x, M' considers the configuration graph G of $M(x)$. It uses the above algorithm for GAP to test whether any one of the final configurations of G is reachable from the starting configuration. As there are $2^{O(S(n))}$ nodes in G, the space used by M' is $O((S(n))^2)$. One subtle issue is that M' cannot construct the whole G, as that would require exponential space. So, it runs the algorithm for GAP without explicitly constructing the graph. Whenever a query "Is $(x, y) \in E$?" is made, it answers it by checking if configuration y can be reached from x in one step. Space constructibility of $S(n)$ is required as we need to compute the number of nodes in G (for which we should know the value of $S(n)$). ♣

We next show that GAP is NL -complete (under log space reductions).

DEFINITION 3.5 *A language B is* NL*-complete if $B \in$ NL, and for any language $A \in$ NL, there is a deterministic Turing machine M that uses $O(\log n)$ space such that, for any x, $x \in A \iff M(x) \in B$.*

Here we have tacitly extended the model of space bounded TMs by allowing it to have a one-way write only output tape, which, like its one-way read only input tape, does not count toward the space bound. (Of course it is easy to show that, if it has space bound $S(n)$, the output length is bounded by $2^{O(S(n))}$, or else the computation will loop and not terminate.)

Exercise: Show that logspace reductions are closed under composition.

THEOREM 3.6 GAP *is* NL *-Complete.*

Since GAP is in P, and clearly P is closed under logspace reductions, a corollary is

COROLLARY 3.7 NL $\subseteq$ P.

Proof. It is easy to see that GAP $\in$ NL, as we simply guess a path (edge by edge and verify every step as we go) of length up to n.

Let $A \in$ NL via a machine M. We give a machine M' that reduces A to GAP . Given an input x, M' simply outputs the configuration graph of M on x. Clearly M' can be implemented in deterministic logspace. ♣

The spirit of this Theorem is the following: To discuss non-deterministic space bounded computation, instead of the abstract formalism of automata theory, without loss of generality, one can discuss the concrete problem GAP. NL is nothing but GAP.

3.3 Immerman-Szelepcsényi Theorem

In this section we prove the Immerman-Szelepcsényi theorem: NL is closed under complementation. Since GAP is NL-*Complete*, it is enough if we show that $\overline{\text{GAP}}$ is in NL .

Given a graph G and vertices s and t, we want to devise a non-deterministic algorithm in Logspace, such that there are some computational paths that end in "yes" iff the graph has no path from s to t (we call such a path an s-t path). This is a somewhat "unnatural" way of thinking about GAP . Usually, we phrase algorithms such that non-deterministic branches seek affirmative answers. In this case, however, if some branch says yes, then it has incontrovertible evidence that there is NOT an s-t path; and moreover if there is NOT an s-t path then some computational path should find such evidence.

While this "twist" is surprising, it is not contradictory, logically speaking. For example, one might try to find a "cut" in the graph separating s and t. Note that such a cut exists iff

there are no paths from s to t. However, we can't simply look for a cut, because (presumably) representing the cut would consume too much space.

Let G be a graph with n vertices, among which are two distinguished vertices s and t. The algorithm runs in two phases. In the first phase, we count the number of vertices reachable from the start vertex s. Define c_i to be the number of vertices reachable from s, within i steps. We incrementally compute $c_0, c_1, \ldots, c_{n-1}$. They are computed non-deterministically. From c_i to c_{i+1}, we will assume the correct c_i has been computed. Still, with c_i written on its tape, only some computational paths will find out the correct c_{i+1} and proceed. Consequently, only some computational paths will find out the correct c_{n-1} and proceed to the next phase. The unsuccessful paths would know that they are unsuccessful and die. In the second phase the value of c_{n-1} is used to check if t is not reachable from s. In the following algorithm, think of any surviving path as saying, "the fact I have not died means I have evidence". The pseudocode is given in Algorithm 1. The ideas used in the algorithm are described next.

We'll incrementally build up counts for $c_0, c_1, \ldots, c_{n-1}$. Finding c_0 is easy: only s is reachable from s in zero steps, and so $c_0 = 1$. c_1 will be the number of vertices for which there is a directed edge from s, to s itself. Suppose we have computed c_i correctly and stored it (only $O(\log n)$ bits are needed to store c_i). We describe how to compute c_{i+1} from c_i.

We undertake the following process for each v, in succession. Our "goal" for each v is to see if we can reach it in at most $i + 1$ steps, and if so we should increment c_{i+1} for it. The way we want to find this out is to "re-experience" the entire set of vertices counted in c_i, and for each such vertex u (which by definition are reachable from s in at most i steps), we check if (u, v) is an edge in G, or $u = v$.

Of course any computational path which correctly computed c_i has in the past encountered all these c_i vertices. However we don't have space to record all the names of these c_i vertices. Thus their identities are necessarily forgotten. So we go through all vertices, from 1 to n, and try to "re-experience" this list of vertices reachable within i steps.

So for any fixed v, we go through the following:

- For each u, we non-deterministically guess whether it was one of the c_i vertices reachable from s in $\leq i$ steps. For any u, if we guess "yes", then we try to validate our guess. We do this by non-deterministically guess a path of length at most i from s to u. If the validation succeeds, u is reachable from s in $\leq i$ steps. Here's the moment of truth from this u: if there is an edge from u to v in G, or $u = v$, we increment c_{i+1}. If the validation fails, i.e., this particular guessed path did not reach u, then our guess was wrong *somewhere*. It could be wrong either because u is not one of the c_i vertices reachable from s in i steps, or our guessed path is not a valid path (even though some such path might exist). In any case some guess along this path is wrong, so halt and die! If we guess that u is not reachable from s in i steps, do nothing and move on to the next u. In this whole process, we keep a count d of the number of vertices u for which we guessed "yes".

- Finally, after processing all u, check if $d = c_i$. (This is the "quality check": after all one can guess a lot of "no" and avoid all the validation and survive.) If $d = c_i$, we have accounted for all the c_i number of vertices that are reachable from s in i steps, since we have validated all the "yes" guesses. We conclude that all the "no" guesses were correct as well. Therefore, if a computational path lived thus far, and got $d = c_i$, then indeed we have "re-experienced" all the vertices reachable from s in i steps. It follows that we have incremented c_{i+1} for v along this computational path iff v is among the c_{i+1} vertices from s in $\leq i + 1$ steps. On the other hand, any path with $d \neq c_i$ simply halts and dies!

Algorithm 1: Shows $\overline{\text{GAP}} \in \text{NL}$
Input: $\langle G, s, t\rangle$: where, $G = (V, E)$ is a directed graph over n vertices, $s, t \in V$
Output: *accept* $\Longleftrightarrow$ there is NO path from s to t in G
IMMERMAN-SZELEPCSÉNYI(G, s, t)

```
(1)   c_0 = 1
(2)   for i = 0 to n − 2
(3)      c_{i+1} = 0
(4)      foreach node v ∈ V
(5)         d = 0
(6)         foreach node u ∈ V
(7)            Guess yes or no: if no, skip the next steps (8) to (12):
(8)               Guess a path of length i from s and if u is not among the
                  vertices on this path, then reject
(9)               d = d + 1
(10)              if [(u, v) ∈ E] ∨ [u = v]
(11)                 increment c_{i+1}
(12)                 goto 6 with next v
(13)        if d ≠ c_i then reject
(14)  d = 0
(15)  foreach node u ∈ V
(16)     Guess yes or no: if no, skip the next steps (17) to (19):
(17)        Guess a path of length n − 1 from s , and if u is not among the
            vertices on this path, then reject
(18)        if u = t then reject
(19)        d = d + 1
(20)  if d ≠ c_{n−1} then reject
(21)                  else accept
```

Note: a lot of non-deterministic paths will "die" in this computation! Guessing a u improperly for candidacy, for any v, will result in death as will guessing a u correctly, but

then following the wrong path. However, some sequence of guesses will run this gauntlet correctly all the way, and we'll have a proper count for c_{n-1}.

Now that we've computed c_{n-1}, we'll use it to answer the $\overline{\text{GAP}}$ question. The idea now is the same: "re-experience" all c_{n-1} vertices and see if t is among them. Thus, we iterate through all the vertices and (non-deterministically) try to account for the c_{n-1} vertices reachable from s once again.

For each node $v \in V$, we do the following: We guess whether there's a path of length $\leq n-1$ from s to v. If the guess is "no", move on to the next vertex. If the guess is "yes", we verify it by guessing a path of length $\leq n-1$ and check if v is encountered along the path. If the verification fails, die! If it succeeds, increment a counter d'. At the "moment of truth" when a vertex v was verified to be among the c_{n-1} vertices reachable from s, we check if it is t.

Finally, we accept $\langle G, s, t\rangle$ iff in the end $d' = c_{n-1}$ and t was never marked as one among the c_{n-1} vertices.

Note that, in the pseudocode above we have made some slight modifications. In line (12) after we confirmed that v should be counted in c_{i+1} we proceed to the next v' immediately. Also in line (18) we reject once we confirmed that t IS reachable. A subtle point: In the pseudocode it is no longer the case that a path with some incorrect guesses must die. Conceptually it is easier to think along the description above: All the guesses will be verified eventually one way or another, and any computational path that made an incorrect guess will eventually die. The whole process appears to be merely to re-experience again and again the set of vertices reachable within some number of steps, and we do the important task of computing the various counts c_i and whether t was among c_{n-1} almost as an after thought. ♣

3.4 Polynomial Space

We devote this section for a discussion of polynomial space bounded computations. Let us start with a comparison of deterministic and non-deterministic polynomial space bounded computations.

Recall that PSPACE consists of all languages computable by deterministic Turing machines that run in polynomial space. One can consider the analogous non-deterministic class, namely NPSPACE, which consists of all languages computable by non-deterministic Turing machines that run in polynomial space.

Unlike the realm of time bounded computations, where the question of P vs. NP is open, in the realm of space bounded computations, the analogous question of PSPACE vs. NPSPACE is easy to resolve. The following theorem is a corollary to Savitch's theorem.

THEOREM 3.8 *PSPACE = NPSPACE.*

Due to the above theorem, we will hardly talk about the notion of non-deterministic polynomial space bounded machines.

Our aim for the rest of the section is two-fold. We first exhibit a complete language for PSPACE. Then, we compare alternating Turing machines and PSPACE Turing machines.

3.4.1 QBF is PSPACE-Complete

In Section 2.2, we presented a canonical complete language (called A_{PSPACE}) for PSPACE. Here, we exhibit a "natural" PSPACE-complete language.

Definition 3.9 *Let f be a (quantifier free) boolean formula over n boolean variables $x_1, x_2, \ldots, x_n$. A quantified boolean formula F derived from f is given by,*

$$F = \exists x_1 \forall x_2, \ldots, Q x_n [f(x_1, x_2, \ldots, x_n)].$$

Here, $Q = \exists$ if n is odd and $Q = \forall$, if n is even. We say that F is valid, *if the above statement is logically true. Meaning, "there exists some choice of x_1 (namely, $x_1 = 1$ or $x_1 = 0$) such that, for all choices x_2, there exists some choice of x_3 such that, so on ..., such that, the formula f evaluates to true under the truth assignment $(x_1, x_2, \ldots, x_n)$.*

Definition 3.10 (QBF)

$$\text{QBF} = \{F | F \textit{ is a valid quantified boolean formula}\}$$

Example. Consider the quantified boolean formula

$$F = \exists x \forall y \exists z [(x \vee \neg y) \wedge (\neg y \vee z) \wedge (\neg x \vee y \vee \neg z)]$$

F is a valid formula. To see that, let $x = 1$, and for $y = 0$ we can set $z = 0$ and for for $y = 1$ we can set $z = 1$. Hence, $F \in$ QBF.

Note that even though we defined QBF to have strictly alternating quantifiers starting with an existential quantifier, these requirements are not essential. We can always append dummy variables to conform to this without increasing the size of the expression too much (at most double the size.)

Our goal is to show that QBF is PSPACE-Complete. As the first step, we prove that QBF is in PSPACE. We present a recursive procedure that solves QBF.

SolveQBF(F)
Execute one of the following cases.

1. Case 1 (F is a formula over zero variables):

- If F is the formula "1" then return(F is valid).
- If F is the formula "0" then return(F is not valid).

2. Case 2 (F starts with an existential quantifier): Let F start with "$\exists x$....". Define two quantified boolean formulas F_1 and F_2 by substituting $x = 0$ and $x = 1$ in F, respectively. Declare F to be valid if at least one of F_1 or F_2 is valid. We can check this by making two recursive calls SolveQBF(F_1) and SolveQBF(F_2).

3. Case 3 (F starts with an universal quantifier): Let F start with "$\forall x$....". Define two quantified boolean formulas F_1 and F_2 by substituting $x = 0$ and $x = 1$ in F, respectively. Declare F to be valid if both F_1 and F_2 are valid. We can check this by making two recursive calls SolveQBF(F_1) and SolveQBF(F_2) (after SolveQBF(F_1) use one bit the remember the result, and then reuse the space in the second call of SolveQBF).

Notice that the formulas F_1 and F_2 have one less variable than F. Thus, if we start with a formula over n variables, the depth of the recursive tree would be n. So the total space usage is polynomial. In short, we can simply do a tree-travesal of this binary tree of depth n defined by F in polynomial space (actually linear space). We have proved the following theorem.

Theorem 3.11 QBF*is in* PSPACE.

Our procedure for QBF is recursive. We often use recursion while writing space efficient algorithms. We saw another example of recursive procedure while proving Savitch's theorem. In general, the amount of space used by a recursive procedure would be $O(d \cdot s)$, where d is the depth of recursion levels and s is the local space usage per recursive level. It should be clear that one can unravel the recursion easily, by using stacks, and obtain a non-recursive procedure using the same amount of space $O(d \cdot s)$. But, it is often easier to write these procedures recursively.

We next show that QBF is PSPACE-hard. Let M be a machine with space bound $p(n)$ and x be an input. The computation of M on x is captured by the configuration graph G of size $2^{O(p(n))}$. Let c_0 and c_{acc} be the initial and accepting configurations of M. (We may wolog assume there is a unique accepting configurations of M.) Then, we want to check if there is a path of length at most $2^{O(p(n))}$ from c_0 to c_{acc}. We want to express the above question as a quantified boolean formula.

Let us do this in general among arbitrary two configurations. Let $c_1 \vdash_{2^k} c_2$ denote that there is a path of length at most 2^k from configuration c_1 to configuration c_2. We can use ideas from Savitch's theorem to express the above criterion as a quantified boolean formula. Recall that such a path exists, if and only if there is a configuration c' such that, there are paths of length at most 2^{k-1} from c_1 to c' and c' to c_2:

$$c_1 \vdash_{2^k} c_2 \iff \exists c'[(c_1 \vdash_{2^{k-1}} c') \wedge (c' \vdash_{2^{k-1}} c_2)]$$

We can recursively expand the expression in square brackets. But, such a method would end up in a formula of exponential size (because, the expression in square brackets involves two recursive calls). So, we play a small trick. We can express the criterion as follows. The criterion

$$(c_1 \vdash_{2^{k-1}} c') \wedge (c' \vdash_{2^{k-1}} c_2)$$

is equivalent to

$$\forall(\text{ configurations } u \text{ and } v)[(\ (u = c_1 \text{ and } v = c') \vee (u = c' \text{ and } v = c_2)\ ) \Longrightarrow (u \vdash_{2^{k-1}} v)].$$

The advantage of this transformation is that the later expression used only one syntactic expression of $u \vdash_{2^{k-1}} v$. We can expand it recursively. At the base case $c_1 \vdash_1 c_2$ for $k = 0$, it can be expressed in a quantifier free boolean expression as in Cook's Theorem. Thus for $c_1 \vdash_{2^{O(p(n))}} c_2$ we obtain in polynomial time a quantified boolean formula F such that the machine accepts x if and only if F is valid.

As configuration graph has $2^{O(p(n))}$ vertices, the final formula is of length $O(p(n))$. It uses only polynomial number of quantifiers. Technically, the final formula is not quite in the form required by our definition of quantified boolean formulas (Definition 2.24). But, F can be converted to the required form easily by using standard logical equalities. We have proved that following theorem.

THEOREM 3.12 *QBF is PSPACE-Complete.*

Theorem 3.11 gives us an easy mechanism to compare PH and PSPACE. Recall that, for any $k \geq 1$, QBF_k is Σ_k^p-Complete (Theorem 2.25). It is easy to see that QBF (with unbounded number of alternations) is a generalization of QBF_k (which can have only some constant number k alternations). Hence, as a corollary to Theorem 3.11, we have the following theorem (which is quite obvious at any rate when you think in terms of a tree traversal for a generic language in Σ_k^p defined by quantifiers, or by a k-alternating TM.)

THEOREM 3.13 *For any $k > 0$, $\Sigma_k^p \subseteq$ PSPACE. Thus,* PH $\subseteq$ PSPACE.

We can now extend Figure 2.1 to Figure 3.1.

3.4.2 APTIME = PSPACE

Our next goal is to connect PSPACE machines and alternating Turing machines (see Section 2.7). We already saw that alternating Turing machines with constant number of alternations accept precisely those languages in the various levels of the polynomial hierarchy (Theorem 2.26). Here we shall consider polynomial time ATMs with unbounded number of alternations.

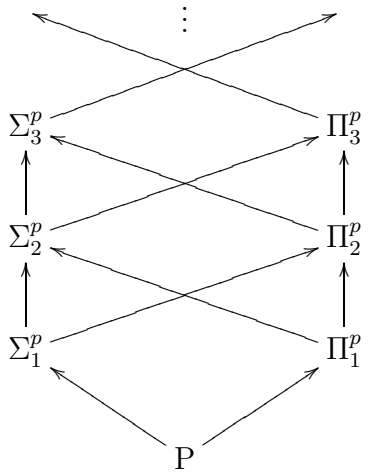

Figure 3.1: Polynomial Time Hierarchy and PSPACE

We say that a language is in the class APTIME, if it is accepted by an alternating Turing machine running in polynomial time. These machines are allowed to use arbitrary number of alternations. But, of course, as the machine runs in polynomial time, it can make at most a polynomial (in input length) number of alternations. We proceed to prove that alternating polynomial time and deterministic polynomial space are equal in power. In other words, we prove that APTIME=PSPACE.

First, let us show that PSPACE $\subseteq$ APTIME. One can give a proof using ideas from Savitch's theorem, similar to that of proving the PSPACE-hardness of QBF. But, as we have already shown that QBF is PSPACE-hard (Theorem 3.12), we will use it to prove the claim. Given a PSPACE machine M and an input x, our APTIME machine first deterministically computes a quantified boolean formula F using the reduction given by Theorem 3.12. So, the machine M accepts x if and only if F is valid. Validity of F can easily be tested in APTIME. Without loss of generality, assume that F is in the standard form given by Definition 3.9. Let F be over n boolean variables. The machine will start in existential mode and choose a value for x_1 (either 0 or 1). Then, it will enter universal mode and choose a value for x_2. In this way, we continue by alternating between existential and universal modes and choose values for $x_1, x_2, \ldots, x_n$. Finally, each computational path will have a full truth assignment in hand. The path will evaluate the underlying boolean formula on this truth assignment. If the evaluation returns "true" then this path will *accept*, else *reject*. From the definition of alternating Turing machines, it is clear that the machine accepts F if and only if F is valid. (One can think of a computation graph of our APTIME machine as a complete binary tree of depth n, where the kth level will have 2^k nodes labeled x_k. Nodes in odd level will be existential and those in even levels will be universal. Each of the 2^n leaves will correspond to a full truth assignment and the leaf would evaluate the underlying boolean formula on

that truth assignment.) We have proved the following claim.

THEOREM 3.14 PSPACE $\subseteq$ APTIME.

We next show that APTIME $\subseteq$ PSPACE. Essentially, we consider the configuration graph of the APTIME machine and do depth first search to find out if its starting configuration is labeled "accept". Our PSPACE procedure is similar to the recursive procedure used to show that QBF is in PSPACE (Theorem 3.11).

We now give a bit more details. Let M be an APTIME machine and x be an input. Each configuration can be stored in polynomial space. We write a procedure that takes as input a configuration c and outputs whether it is an accepting configuration. We use recursion and simply follow the definition of alternating Turing machines. It is easy when c is a leaf configuration. We simply check if c is an accepting state (as specified in the description of M). If so, c is an accepting configuration, else it is a rejecting configuration. Suppose the input c is a non-leaf configuration. Consider each of its children c', one by one. Recursively, find out c' is accepting or rejecting. If c is an existential configuration, then c is accepting if and only if at least one of its children is accepting. Similarly, if c is a universal configuration, then c is accepting if and only if all its children are accepting configurations.

Using the above procedure, we can find out if the starting configuration of M is accepting or rejecting. As M runs in polynomial time our procedure will need only polynomial number of levels of recursion. Each level needs only polynomial amount of space. Thus, the entire procedure runs in polynomial space. We have proved the following claim.

THEOREM 3.15 PSPACE $\subseteq$ APTIME

Combining Theorem 3.14 and Theorem 3.15, we have the following conclusion.

THEOREM 3.16 APTIME = PSPACE