

5.7 APSPACE = EXP

(\Rightarrow) Suppose $L \in APSPACE$, and let M be a poly-space ATM deciding L . We build a TM M' that decides L by computing the configuration graph of M on a given input x , and then marking nodes with “ACCEPT” as specified in the definition of an ATM to decide whether M accepts x . Since M uses polynomial space, the size of its configuration graph is at most singly exponential, and so M' runs in singly exponential time. Thus $L \in EXP$.

(\Leftarrow) Suppose $L \in EXP$, and let M be a two-tape exp-time TM deciding L . We will make use of the locality of M 's computation to shrink our space usage to polynomial. Given an input x , let $H(i, j, a, q)$ be a predicate that's true iff after M runs on x for i steps, its tape head (more specifically, work/output tape head) is at position j , the character a is on the tape at position j , and M is in state q . Let $N(i, j, a, q)$ be true iff after M runs on x for i steps, character a is on the tape at position j , M is in state q , and the tape head is *not* at position j . Note that since M runs in time $O(2^{poly(n)})$, the step i and cell position j can be written in $O(poly(n))$ space. Thus these predicates only take up polynomial space. Now we come up with recursive formulas to compute H and N . Notice that if $H(i, j, a, q)$ holds then at step $i - 1$, either the tape head was still at cell j and didn't move, or the tape head was at cell $j + 1$ and moved left, or it was at cell $i - 1$ and moved right. Similarly, if $N(i, j, a, q)$ holds then at step $i - 1$, either the tape head was at cell j and moved away, or it was at cell $j + 1$ and didn't move left, or it was at cell $j - 1$ and didn't move right, or it wasn't at any of cells $j - 1, j, j + 1$. This reasoning leads to the following formulas for H and N when $i > 1$:

$$\begin{aligned}
 H(i, j, a, q) &:= \exists b, q'. \\
 &(\delta(q', b) = (q, a, S) \wedge H(i - 1, j, b, q')) \\
 \vee (\delta(q', b) = (q, -, L) \wedge H(i - 1, j + 1, b, q') \wedge N(i - 1, j, a, q')) \\
 \vee (\delta(q', b) = (q, -, R) \wedge H(i - 1, j - 1, b, q') \wedge N(i - 1, j, a, q')) \\
 N(i, j, a, q) &:= \exists b, c, q'. \\
 &(\delta(q', b) = (q, a, R|L) \wedge H(i - 1, j, b, q')) \\
 \vee (\delta(q', b) = (q, -, R|S) \wedge H(i - 1, j + 1, b, q') \wedge N(i - 1, j, a, q')) \\
 \vee (\delta(q', b) = (q, -, L|S) \wedge H(i - 1, j - 1, b, q') \wedge N(i - 1, j, a, q')) \\
 \vee (N(i - 1, j - 1, b, q') \wedge N(i - 1, j, a, q') \wedge N(i - 1, j + 1, c, q'))
 \end{aligned}$$

We leave it as a simple but tedious exercise for the reader to formally prove the validity of these formulas. Also note that we technically should be keeping track of the input tape head as well and have input bits as part of our transitions, but this would make the formula somewhat unreadable and is completely trivial to do, so we're omitting it. Now we can use the above formulas to obtain a poly-space ATM M' deciding $L(M)$ as follows: on input

x accept iff $H(2^{p(n)}, 1, 1, q_{halt})$ holds. To check whether it holds, recurse on the formulas above, using \exists states to choose b, c, q' , and the disjunct to recurse on, and using \forall states to simultaneously recurse on all the conjuncts in a particular disjunct. Once we reach $i = 1$, we just look at the definition of M to decide what to return. We can reuse all space with each recursive call since we don't need any kind of stack, so this algorithm runs in poly-space. Thus $L \in APSPACE$.

Alternate Solution: According to Wikipedia, it is known that finding a winning strategy for the game of checkers on an $n \times n$ board is EXP-complete. Thus if we can show that CHECKERS $\in APSPACE$, then we immediately get $EXP \subseteq APSPACE$. It's actually quite easy to see that CHECKERS $\in APSPACE$; player 1 gets \exists states and player 2 gets \forall states. It's pretty clear that we can encode the current board state and legal moves in polynomial space, so we can indeed decide this problem with a poly-space ATM. Note that we can't say CHECKERS $\in PSPACE$ by the same logic because the number of moves (i.e. quantifier alternations) can be exponential.

5.9 EXACT INDSET

a.) $(G, k) \in EI \iff \forall S \subseteq V, \exists S' \subseteq V$ s.t. (S' is an ind set, $|S'| = k$, and (S is an ind set $\Rightarrow |S| \leq |S'|$)). Thus $EI \in \Pi_2^p$.

b.)

$$(G, k) \in EI \iff (G, k) \in INDSET \wedge (G, k + 1) \notin INDSET$$

Letting $L = \{(G, k) | G \text{ doesn't have an ind set of size } k + 1\}$, we see from the above that $EI = INDSET \cap L$. It's clear that $INDSET \in NP$ and $L \in coNP$; hence $EI \in DP$.

c.) Pick any $L \in DP \Rightarrow L = L_1 \cap L_2$, where $L_1 \in NP, L_2 \in coNP$. Let ϕ_1 be the formula obtained from reducing L_1 to $3SAT$, and ϕ_2 the formula obtained from reducing L_2 to $\overline{3SAT}$. Now consider the reduction from $3SAT$ to $INDSET$ described in Figure 2.5 on pg 52 of the textbook. Augment the reduction by adding $k - 1$ nodes and add edges from each of them to every other node in the graph. Then the reduction will always give a maximum ind set of size k if the formula of k clauses is satisfiable, and a maximum ind set of size $k - 1$ if it's not. Let (G_1, k_1) and (G_2, k_2) be the results of applying this reduction to ϕ_1 and ϕ_2 , respectively. Then we have: $x \in L_1 \iff \phi_1 \in 3SAT \iff (G_1, k_1) \in EI$ and $x \in L_2 \iff \phi_2 \notin 3SAT \iff (G_2, k_2) \notin INDSET \iff (G_2, k_2 - 1) \in EI$. Now define $G = G_1 \times G_2$, where $V(G) = V(G_1) \times V(G_2)$, and there's an edge in G from (u_1, u_2) to (v_1, v_2)

iff there's an edge from u_1 to v_1 in G_1 or an edge from u_2 to v_2 in G_2 . Then it's not hard to see that $(G_1, a) \in EI \wedge (G_2, b) \in EI \iff (G, ab) \in EI$. Hence $\forall x, x \in L \iff x \in L_1 \wedge x \in L_2 \iff (G_1, k_1) \in EI \wedge (G_2, k_2 - 1) \in EI \iff (G, k_1(k_2 - 1)) \in EI$, i.e. $L \leq_p EI$.

5.11 SUCCINT SET-COVER

SUCCINT SET-COVER is defined as: given $S = \{\phi_1, \dots, \phi_m\}$ and k , $x \in \text{SUCCINT SET-COVER} \iff \exists S' \subseteq \{1, \dots, m\}$ s.t. \forall assignments to boolean variables, $|S'| \leq k \wedge \bigvee_{i \in S'} \phi_i$ evaluates to 1. Hence $\text{SUCCINT SET-COVER} \in \Sigma_2^p$.

6.3 Decidable Language in $P_{/poly} - P$

Define B and U_B as in the proof of the Baker-Gill-Solovay theorem. We know from Problem 3.3 on the previous problem set that we can make B decidable in exponential time. Thus U_B can also be decidable, since we can just check whether each string of a given length is in B . Furthermore, we proved in the course of the theorem that $U_B \notin P^B$, which clearly implies that $U_B \notin P$. Finally, U_B is a unary language, so it's trivially in $P_{/poly}$.

6.4 Theorem 6.15

(\Rightarrow) Suppose L has a logspace-uniform circuit family $\{C_n\}$ of polynomial size. Then for any input x of length n , we can determine whether $x \in L$ in polynomial time by computing circuit C_n (poly time since the circuit family is logspace uniform and poly size) and then directly simulating it on x . Hence $L \in P$.

(\Leftarrow) Suppose $L \in P$, and let M be a TM deciding L in poly time. Use the standard construction from the Cook-Levin theorem to get an oblivious TM M' with $L(M) = L(M')$. In Problem 4.6 on the previous problem set, we saw that the Cook-Levin reduction can be made into an implicitly logspace computable function. Thus, for any length n , we can compute any particular bit of circuit C_n such that $\forall x \in \{0, 1\}^n, C_n(x) = L(x)$, as described in the proof of Theorem 6.6 (which essentially just reiterates the Cook-Levin reduction). Furthermore, since M' runs in poly time, the circuit C_n , which checks all of M' 's snapshots on input x , is poly size. Hence L has a logspace-uniform circuit family of polynomial size.

6.9 Mahaney

Let L be an NP-complete sparse language, and let R be a poly-time reduc-

tion from SAT to L . Notice that, given a length n , R maps the exponential number of strings of length at most n to a polynomial number of poly-length strings. Thus we can solve SAT in polynomial time using essentially the same algorithm as we used in Problem 2.30 on the rst problem set. The algorithm was:

Algorithm 1: *sat*

Input: ϕ and an array of known results A (initialized to unknown before the first call)

1. if $n = 0$ then return ϕ (which must be either true or false)
2. if $A[|R(\phi)|]$ is not equal to unknown then return $A[|R(\phi)|]$
3. if $sat(\phi(0, x_2, \dots, x_n), A) = \text{true}$ or $sat(\phi(1, x_2, \dots, x_n), A) = \text{true}$ then
4. set $A[|R(\phi)|] = \text{true}$
5. return true
6. else
7. set $A[|R(\phi)|] = \text{false}$
8. return false

By the same argument we used for 2.30, this algorithm does indeed solve SAT in polynomial time as long as we can store and lookup values in the array A in poly time. This is actually a minor issue since there are an exponential number of possible indices into the array; however, the issue is easily resolved by using a hash table or linked list rather than a standard array. Since $SAT \in P$ and everything in NP reduces to SAT , we conclude that $P = NP$.