CS 221 Computational Complexity, Lecture 6 Feb 8, 2018

Alternation, Time, Space; Fortnow's theorem

Instructor: Madhu Sudan *Scribe: Patrick Guo*

1 Overview

We introduce the concept of alternation, a generalization of nondeterminism, and prove the following results relating alternating time and space to more familiar complexity classes:

- ATIME(poly) = PSPACE
- ASPACE(log) = P

This will then allow us to show the following time-space tradeoff for SAT:

- Fortnow's Theorem: $SAT \in L \implies SAT \notin \text{TIME}(n^{1+o(1)})$

2 Alternating Algorithms

Definition 1. *An Alternating Algorithm is an algorithm which allows the use of the quantifiers* $\forall$ *and* $\exists$ *in addition to all the usual programming features*

We can represent the computation of an alternating algorithm by a tree: regular deterministic steps move the state of our machine to exactly one child state, but our alternating algorithm introduces the additional states $\forall$ and $\exists$ which fork the computation into multiple branches: a universal $\forall$ node accepts if all branches accept (think AND) and the existential $\exists$ node accepts if at least one branch accepts (think OR). From this definition it is clear that NP, coNP $\subset$ ATIME(poly) (since an NP-algorithm is a polynomial time algorithm with an existential $\exists$ operator, and its complement a polynomial time algorithm with a universal $\forall$ operator).

We are interested in the time and space used by alternating algorithms, which are formalized as follows: time is the depth of the computation tree, and space is the maximum space used over all computations to the leaves. Then, for notation, we have

- ATIME($t(n)$): what you can do with alternating algorithm in time $t(n)$
- ASPACE($s(n)$): what you can do with alternating algorithm in space $s(n)$
- To restrict by $a(n)$ alternations, we will use the notation $\text{ATIME}_{a(n)}(t(n))$

With our notation we can write statements like ATIME_1(poly) = NP $\cup$ coNP (note: number of alternations counts just the places where quantifiers change, e.g. $\exists\forall\exists\exists\forall\forall\forall$ is just 4 alternations, since we can 'merge' consecutive identical quantifiers)

One way to really understand alternating algorithms is to think of them as two-player games between an existential $\exists$ player and a universal $\forall$ player who make choices at respective nodes. This gives us the following metaphors: ATIME(poly) $\cong$ Go (where you dont remove pebbles), ASPACE(poly) $\cong$ Chess.

ATIME(poly) - the "go" problem - starting with x configuration on an n sized go board, who is the winner? $\exists$ and $\forall$ alternate turns choosing computation branches (i.e. placing pebbles). The existential player chooses configurations that he believes leads to an accept state, and the universal player looks for a counterexample in one of the resulting branches. This is subject to the following rules:

1. moves poly time verifiable for legality
2. at the end, the winner is poly time computable
3. total number of moves is poly(n)

By contrast, ASPACE(poly) - the "chess" problem - has the same first two conditions, but the total number of moves is not limited by poly(n) (eg think 8 queens on a chessboard); instead, the total number of moves can be exponential. Hence, Go and Chess give us a good way to think about in ATIME(poly) and ASPACE(poly)

Exercise 2. *Generalize "go" and "chess" into games that are ATIME(poly)-complete and ASPACE(poly)-complete, and prove their completeness*

In more familiar terms, "go"/"chess" are PSPACE-complete/EXP-time-complete, respectively, and in the following sections we prove these relations between alternating time/space and deterministic space/time

2.1 ATIME(poly) = PSPACE

Specifically, we show

Theorem 3. $SPACE(t(n)) \subseteq ATIME(t^2(n)) \subseteq SPACE(t^2(n))$

Proof. $SPACE(t(n)) \subseteq ATIME(t^2(n))$: this is a Savitch-style proof. With $t(n)$ space, there are $2^{t(n)}$ maximum deterministic steps (one for each unique configuration) to get from the initial s_0 state to a possible accepting state s_f. An alternating algorithm can simulate this computation by having the existential player guess the middle computation s_1 between the initial state and the final state. The universal player then responds with which computation $s_0 \to s_1$ or $s_1 \to s_f$ he needs proven, and then the existential player returns a guess for the middle configuration in that computation, and so on. Since each step halves the number of steps in the computation left to prove, this process requires $\log(2^{t(n)}) = t(n)$ guesses from the existential player, and each guess takes $t(n)$ time to write the guessed configuration, thus giving a runtime of $ATIME(t^2(n))$.

$ATIME(t^2(n)) \subseteq SPACE(t^2(n))$: an alternating algorithm with runtime $O(t^2(n))$ has tree depth $O(t^2(n))$ and we can deterministically simulate this tree of $\exists$ (or's) and $\forall$ (and's) with space $O(depth) = O(t^2(n))$ by using a depth-first search to determine which nodes of the tree accept. At each branching node, we just need to store which nondeterministic choice was made and whether the node was $\exists$ or $\forall$, so the the alternating algorithm can be simulated with deterministic space $O(t^2(n))$, the depth of the tree (and thus maximum number of nodes DFS needs to store these values for at any one time).

$\square$

2.2 ASPACE(log) = P

This follows from

Theorem 4. $TIME(2^{s(n)}) \subseteq ASPACE(s(n)) \subseteq TIME(2^{O(s(n))})$

$TIME(2^{s(n)}) \subseteq ASPACE(s(n))$: we will use locality of algorithms. Looking at the Turing machine table of the computation of some algorithm running in $TIME(2^{s(n)})$, we construct a game that determines whether or not the algorithm accepts. The universal player challenges the value at $CELL(i, j)$ (the jth bit of the machine in step i of the computation), and the existential player gives values of $\{CELL(i-1, t)\}_{t \in [\pm c]}$ (for some c constant by locality of algorithms) that resulted in the bit found at $CELL(i, j)$. Then, the universal player challenges one of the new cells the existential player gave, and so on, until we reach the initial configuration that can be verified. The only memory we need to store at any one time is $i, j, CELL(i, j)$, which takes $O(\log(2^{s(n)})) = O(s(n))$ bits, thus $TIME(2^{s(n)}) \subseteq ASPACE(s(n))$.

$SPACE(s(n)) \subseteq TIME(2^{O(s(n))})$: Given alternating algorithm with space $s(n)$, we can build a directed graph in time $2^{s(n)}$ where vertices are configurations (state of algorithm, memory) and edges (u, v) imply can go from state u to v in one step. Now we can add a counter to algorithm - if the counter reaches $> 2^{s(n)}$, reject, since this means we have started looping through configuration.

Thus, the tree for the computation of this alternating algorithm has depth at most $2^{s(n)}$, and since there are $2^{s(n)}$ configurations total, by merging together vertices on the same row that represent the same configuration, we have a graph with at most $2^{2s(n)}$ vertices, and moreover note that after merging identical vertices we will have a directed acyclic graph, so we can just do a topological sort and traverse the graph to simulate its computation, which takes $O(size) = O(2^{O(s(n))})$ time, thus we have $SPACE(s(n)) \subseteq TIME(2^{O(s(n))})$.

3 Fortnow's Theorem ('98)

Theorem 5. *$SAT \in L \implies SAT \notin$ TIME($n^{1+o(1)}$)*

Intuitively, the tradeoff is as follows: if $SAT \in TIME(n^{1+\varepsilon})$, then this means non-determinism is not powerful, and thus co-nondeterminism is not powerful either, so the quantifiers $\exists, \forall$ are both not powerful, and in general, alternation is not powerful. On the other hand, if $SAT \in L$, then TIME(t(n)) $\approx$ SPACE(log(t(n))), so computations can be made to take small space, then can do savitch-style alternation proof in small space, thus showing alternation is powerful.

For a sketch of the proof, we apply a stronger Cook's theorem that states $Ntime(t(n)) \leq SAT$ of length $t(n)\log t(n)$. Now, supposing $SAT \in L$, this means $(N)TIME(t(n)) \subseteq SPACE(c\log t(n))$, and from a Savitch-style proof we can show that $SPACE(c\log t(n)) \subseteq ATIME_a(t(n)^{c/a})$. For example, with $a = 2$, there are $t(n)^c$ possible configurations, and the existential player guesses middle configurations $s_1, \cdots, s_{t(n)^{c/2}-1}$ to the universal player, chopping up the possible configurations into chunks of $t(n)^{c/2}$, and the universal player then replies with one computation $s_i \to s_{i+1}$ that needs to be proved, which, after using up our 2 allotted alternations, takes $O(t(n)^{c/2})$ runtime, for a total of $ATIME_2(t(n)^{c/2})$.

Exercise 6. *Show for $a > 2$ that $SPACE(c\log t(n)) \subseteq ATIME_a(t(n)^{c/a})$*

Putting these results together, it follows that if $SAT \in L$, then $TIME(t(n)) \subseteq ATIME_a(t(n)^{c/a})$

Now, suppose $SAT \in Time(n^{1+\varepsilon})$. It follows that $\exists ATIME_1(f(n)) \leq Time(f(n)^{1+\varepsilon})$ where we use $\exists ATIME_1(f(n))$ to denote languages in $ATIME_1(f(n))$ whose alternation begins with $\exists$. Then, we can take the universal $\forall$ on both sides, and complement, then take the existential $\exists$ on both sides, and repeatedly apply this process, we get $ATIME_a(f(n)) \subseteq TIME(f(n)^{(1+\varepsilon)^a}) \approx TIME(f(n)^{1+2a\varepsilon})$ where the approximation comes from taking ε small.

Exercise 7. *Complete the details for the step $SAT \in Time(n^{1+\varepsilon}) \implies ATIME_a(f(n)) \subseteq TIME(f(n)^{(1+\varepsilon)^a})$*

Taking $f(n) = t(n)^{c/a}$, this gives $ATIME_a(t(n)^{c/a}) \subseteq TIME(t(n)^{c/a+2\varepsilon c})$, then choosing large a (e.g. $3c$) will give us $ATIME_a(t(n)^{c/a}) \subseteq TIME(t(n)^{c/2})$ but $TIME(t(n)^c) \subseteq ATIME_a(t(n)^{c/a})$ followed from $SAT \in L$, contradicting the time hierarchy theorem. Hence we cannot have both $SAT \in L$ and $SAT \in TIME(n^{1+o(1)})$.

We believe many stronger statements about SAT ($SAT \notin L$, $SAT \notin Time(n^{1+o(1)})$), but they are hard to show. Alternations appeared at first glance to be unrelated to such questions, but with such a notion Fortnow was able to show that at least we cannot be simultaneously wrong about SAT.