CS 4510 Automata and Complexity 11/15/2023

Lecture 19: In and around PSPACE

Lecturer: Abrahim Ladha *Scribe(s): Rahul*

So far, we proved a few theorems in and around NP. We proved the Cook-Levin theorem, that SAT was NP-complete. We also proved Ladner's theorem, that if $\mathsf{P} \neq \mathsf{NP}$ then there exists languages $\notin \mathsf{P}$, $\in \mathsf{NP}$ and not NP-complete. Today's lecture will be on space, that "other" resource. Space is a very different resource than time. After an algorithm finishes running, you get the space back. You can never get the time back. This makes space both a less interesting and more interesting resource to study since it uses techniques and tricks which would not work for time. They are less applicable, but interesting in their own right. For example, performing super-exponential search to use one less unit of space.

1 Space as a Resource

Recall $\mathsf{TIME}(f(n)), \mathsf{SPACE}(f(n))$ are the classes of languages decidable in $f(n)$ time or space, respectively. We prove the following containment chain:

Theorem 1.

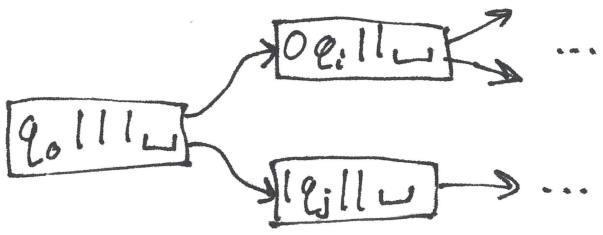
$$\mathsf{TIME}(f(n)) \subseteq \mathsf{SPACE}(f(n)) \subseteq \mathsf{TIME}(2^{O(f(n))})$$

Proof. Consider a language decidable in $f(n)$ time. There exists a Turing machine which takes $f(n)$ steps to decide this language on inputs of length n. At each step, it may use at most one new cell of the tape. So a machine which uses $f(n)$ time can use no more than $f(n)$ space. The first containment then follows.

We show a stronger result to prove the second containment.

$$\mathsf{SPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(2^{O(f(n))})$$

The first containment follows from the generalization of non-determinism. We can now show the second containment in a creative way. Given a language decidable by a non-deterministic Turing Machine in $f(n)$ space, we want to show this language is decidable deterministically in $2^{O(f(n))}$ time. We will do so by graph search! For some specific N, w, let the configuration graph G be a directed graph such that each node corresponds to a configuration of N on w. Note that if N runs in $f(n)$ space, then this graph is not infinite. There exists a bound of the possible number of vertices. Also notice that since as defined, N must halt on all inputs, this graph does not contain a cycle, it is a rooted tre.



Note that we do not count the input as a part of the space used. In some models, the input is on a separate read-only tape. Since our machine N is non-deterministic, our graph may have a arity greater than one. We may assume it has arity no more than two. In order to show $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(2^{O(f(n))})$ we give a deterministic algorithm which simulates N on w in time $2^{O(f(n))}$. First using N, w build the configuration graph. Then we perform BFS from the start configuration C_o to an accepting one C_a. BFS is linear time in the size of input. This graph has worst case $2^{O(f(n))}$ nodes, as that is the number of possible configurations of an $f(n)$ space machine. It also takes that long to build the graph, so we see this is a $2^{O(f(n))}$ deterministic algorithm so $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{TIME}(2^{O(f(n))})$.

□

2 Savitch's Theorem

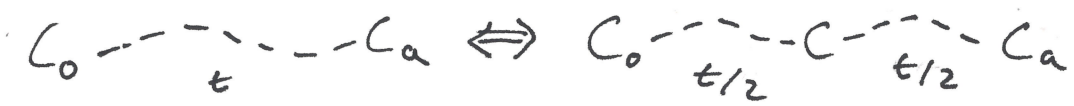
Our main result today:

Theorem 2 (Savitch)**.** For $f(n) \geq \log n$, $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f^2(n))$

Let us first interpret this result. We somehow are able to "de-nondeterminisfy" a complexity class with only polynomial overhead in the resource used. Could such a technique apply to P vs NP? Probably not, or someone would have found it by now. So although we only get polynomial space cost, we can infer we probably will get a super-polynomial, maybe exponential time cost. Our deterministic algorithm may only use $f^2(n)$ space, but it should probably use $2^{f(n)}$ time to perform this simulation.

A second immediate remark is that since polynomials are closed under composition, multiplication, we see $\mathsf{NPSPACE} = \mathsf{PSPACE}$. The study of space, already looks very different than the study of time. This should be an analogous problem to P vs NP. Unlike that problem, this result is unexpected, and we have been able to solve it. Nondeterminism does not have a strong effect on space complexity.

Proof. Let N be nondeterministic Turing machine which uses $f(n)$ space. We give a deterministic simulator M which uses no more than $f^2(n)$ space. Rather than some naive simulation strategy, we are going to simulate using divide and conquer! If C_o is the start configuration, C_a is the accept configuration[1], and C is some other, intermediary configuration, notice that $C_o \overset{*}{\vdash} C_a$ in t steps if $C_o \overset{*}{\vdash} C$ in $t/2$ steps and $C \overset{*}{\vdash} C_a$ in $t/2$ steps for some t.



This will be our divide and conquer recurrence. We brute force search for some C and perform our recurrence in this way. Importantly, our recursive calls are run sequentially and reuse space.

[1]Without loss of generality, we may suppose that there is a unique accepting configuration. We could modify N so that it clears the tape and resets the tape head before accepting.

Algorithm 1 $M(N, w)$ Deterministic simulator of N on w

C_0 = start configuration of N on w
C_a = accepting configuration of N
d = chosen such that N has no more than $2^{df(n)}$ configurations
$YIELDS(C_0, C_a, 2^{df(n)})$

Algorithm 2 $YIELDS(C_i, C_j, t)$

```
if C_i = C_j then
    return true
end if
if t = 1 then
    if C_i ⊢ C_j in one step by δ of N then
        return true
    end if
else t > 1
    for configuration C of N of size f(n) do
        YIELDS(C_i, C, t/2)
        YIELDS(C, C_j, t/2)
        return true if both calls return true
    end for
end if
return false
```

It certainly is correct. M is a deterministic simulator of N, so it decides the same language. Now onto the analysis. If N uses $f(n)$ space, we hope to show M simulates N in no more than $f^2(n)$ space. Space is reusable, unlike time. You may recall using the master theorem, which counts all branches to measure the time complexity of an algorithm. Since space is reusable, we need not do all this provided we execute the recursive calls sequentially. The space used by such a divide and conquer algorithm is simply the recursion depth multiplied by the size of a stack frame.

For each recursive call, a stack frame containing $\langle C_i, C_j, t\rangle$ is stored. Since N uses $f(n)$ space, we see that $|C_i| = |C_j| = O(f(n))$. Also notice that with $t = 2^{df(n)}$, the size of t in the worst case is $\log t = O(f(n))$, so the size of a stack frame is simply $O(f(n))$.

Each level of the recursion divides t in half, so the depth of our recursion tree is $\log t = O(f(n))$. Since each level of our recursion tree takes $O(f(n))$ space and our recursion has $O(f(n))$ depth we observe the total space used is $O(f(n)) \cdot O(f(n)) = O(f^2(n))$ □

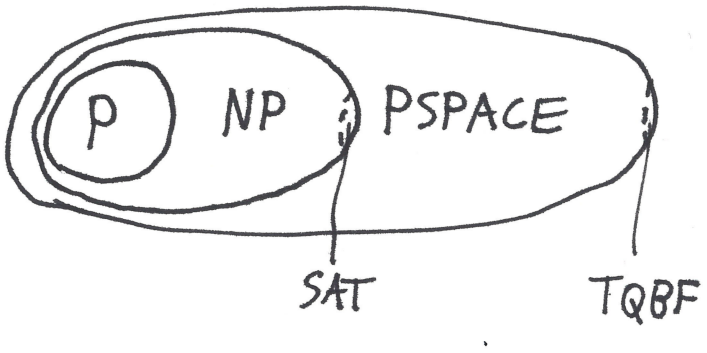
There are some restrictions on $f(n)$. First, unmentioned, is that we may assume it is space-constructible, that M can compute $f(n)$ within $O(f(n))$ space. Most obvious functions have this property, but some crazy ones do not. Second is that $f(n) \geq \log(n)$. This is required as storing a configuration of N on w takes $\log(n2^{O(f(n))}) = \log n + f(n)$. A final remark, Hartmanis came up with a similar idea but to prove a theorem about context-free languages. Savitch was the one to notice the technique could apply to space

complexity[2].

3 PSPACE-completeness

Recall that SAT is NP-complete, a boolean formula might look like $(x_1 \vee x_2 \vee x_3)$. This is not a boolean formula so much as it is a logical formula! We just hide the quantifiers. We say a boolean formula is satisfiable if there *exists* a satisfying assignment. We could simply quantify over the assignment, like $\exists x_1 \exists x_2 \exists x_3 (x_1 \vee x_2 \vee x_3)$.

What if we allow for universal quantifiers? Like $\forall x_1 \forall x_2 \exists x_3 (x_1 \vee x_2 \vee x_3)$? This is called TQBF: True Quantifies Boolean Formula. TQBF $= \{ \phi \mid \phi$ is a true quantified boolean formula $\}$. Turns out that as SAT is NP-Complete, TQBF PSPACE-complete. The intuition is that since TQBF is a generalization of SAT, it should be harder than SAT.



Definition 3.1. A language B is said to be PSPACE-complete if $\forall L \in$ PSPACE that $L \leq_p B$ and $B \in$ PSPACE.

It is a similar definition to NP-completeness. You may ask why do we still use the notion of polytime reduction and not a polyspace reduction. We would like the definition we use of transformation to be far easier than the problems themselves. The reductions should not have any resource to do any solving, only transforming. Analogously, all problems in P are P-complete if you consider your reduction to be the polytime one.

Theorem 3. TQBF is PSPACE-complete

Proof. First, we show TQBF $\in$ PSPACE by giving a PSPACE algorithm. Consider the formula $Q_1 x_1 Q_2 x_2 ... Q_n x_n [\Phi(x_1, x_2, ..., x_n)]$ where each Q_i could be either $\exists, \forall$. We give a recursive polynomial space algorithm A to determine if it is true.

Importantly, the recursive calls are done sequentially to reuse space. The recursion depth is the number of quantifiers, n. For each stack frame, you only need to store at most one bit, the answer. So this algorithm is in fact linear space and we conclude TQBF $\in$ PSPACE.

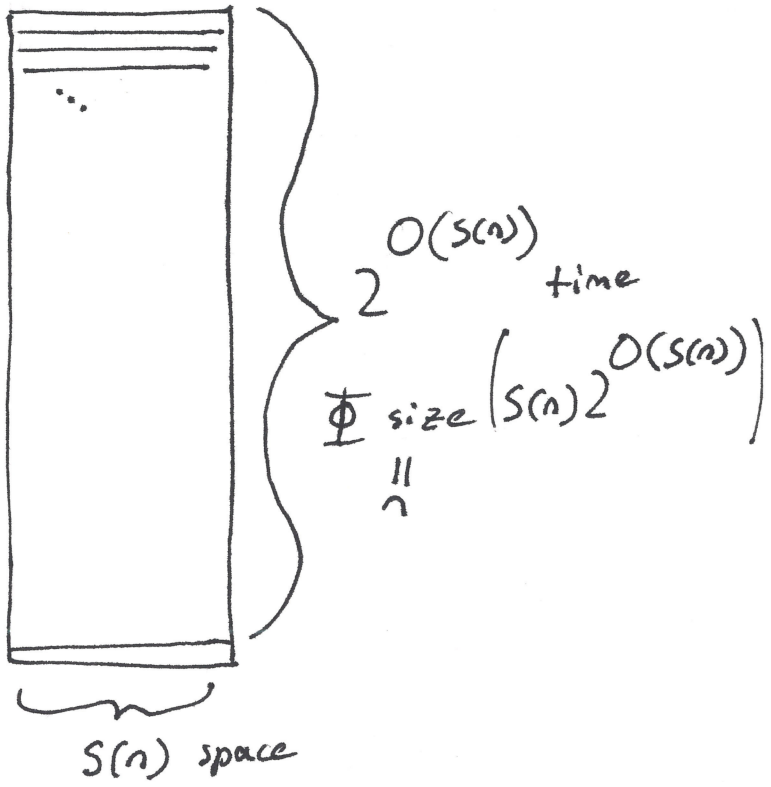
Next we prove it is PSPACE-hard. Let M be a machine which uses only polynomial space $S(n)$. We give a quantified boolean formula Φ to simulate M on input w such that M

[2]See this post by Lipton for some fascinating history of the theorem `https://rjlipton.wpcomstaging.com/2009/04/05/savitchs-theorem`

Algorithm 3 $\mathrm{A}(Q_1x_1...Q_nx_n[\Phi(x_1,...,x_n)]$

if no quantifiers **then**
  evaluate Φ and accept/reject appropriately
end if
if $Q_1 = \exists$ **then**
  return $\mathrm{A}(Q_2x_2...Q_nx_n[\Phi(0,...,x_n)]) \vee \mathrm{A}(Q_2x_2...Q_nx_n[\Phi(1,...,x_n)])$
end if
if $Q_1 = \forall$ **then**
  return $\mathrm{A}(Q_2x_2...Q_nx_n[\Phi(0,...,x_n)]) \wedge \mathrm{A}(Q_2x_2...Q_nx_n[\Phi(1,...,x_n)])$
end if

accepts $w \iff \Phi \in$ TQBF. One first idea is to repeat the strategy used in the Cook-Levin theorem.



Our table with only have polynomial width, but could possibly have super exponential height. A polyspace machine may use exponential time. This formula would then be exponentially sized meaning our reduction would not take polytime. It would take too long for the reduction to even write the formula down. Instead, we proceed similar to Savitch's theorem.

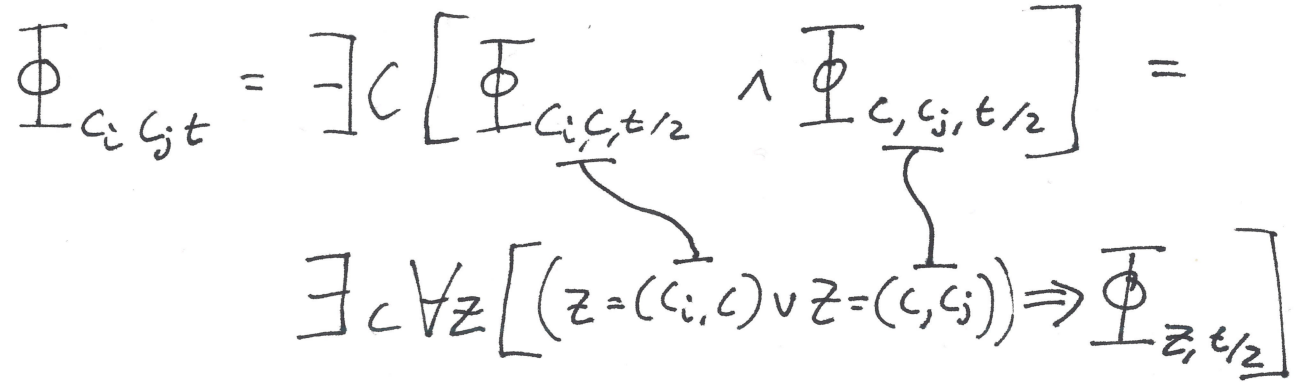
We recursively define a quantified boolean formula. First suppose there exists a CNF formula with no quantifiers $\Phi_{C_1,C_2,1}$ which is true if and only if C_1, C_2 are configurations of the PSPACE machine M and either $C_1 = C_2$ or $C_1 \vdash C_2$ after one step of δ of M. Such a

formula exists by a similar construction in the Cook-Levin theorem, and is of size $O(S(n))$.

Now we may recursively define our formula as

$$\Phi_{C_i,C_j,t} = \exists C[\Phi_{C_i,C,t/2} \wedge \Phi_{C,C_j,t/2}]$$

Note that by "$\exists C$", we really mean that C is encoded into several variables and we quantify like $\exists c_{12}$.... As long as they are all the same quantifier this generalization is fine. Each level does cut t in half, but doubles the size of the formula. The formula size is approximately $t = 2^{O(S(n))}$, too big for a polytime reduction to write down. We can make a smaller formula by using a universal quantifier to fold the formula in half.



Note that equality and implication are not allowed in CNF form, but the formula could obviously be rewritten into a CNF, making it harder to read. Each time we half t, we only add a linear sized amount to our formula, so the size of the formula is $(\log t)S(n) = O(S^2(n))$, which is a polynomial. So our reduction here can take polynomial time.

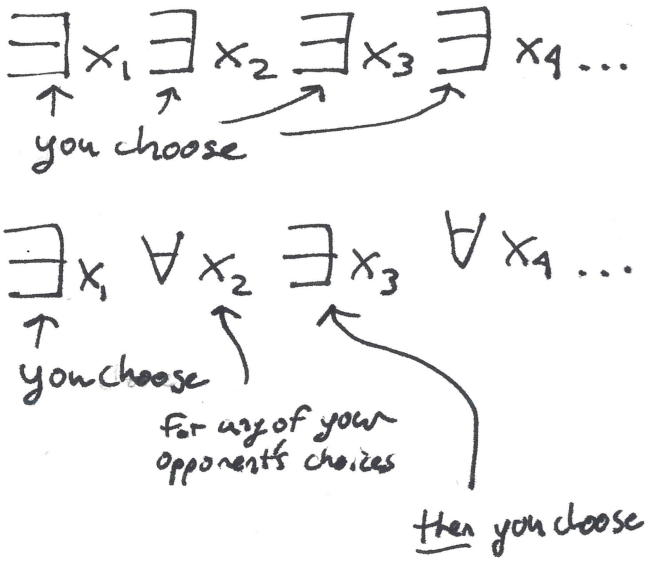
Since TQBF $\in$ PSPACE and TQBF is PSPACE-hard we see that TQBF is PSPACE-complete.

□

4 Puzzles and Games

Notice SAT has structure like most puzzles. A puzzle is a single-player device in which you make a sequence of decisions to reach some goal. Intuitively, $\exists x_1, \exists x_2, ...$ is your sequence of decisions. Many puzzles are NP-complete since they can encode this structure.

Notice TQBF has structure like two player games of perfect information. Consider a TQBF with quantifiers like $\exists\exists\forall\forall\exists\exists$... you can reformulate this into a TQBF with quantifiers which only alternate, like $\exists\forall\exists\forall$... You can turn two of the same kind of quantifier into one as $\exists x_1 \exists x_2 \equiv \exists(x_1, x_2)$. Having a TQBF with alternating quantifiers looks like a game! It is a literal minimax. You make a choice, then for all possible moves the opponent could make, then you make a choice, then the opponent, and so on.



Most two player games, under appropriate restrictions and generalizations, are PSPACE-complete. Chess, checkers, Go, tic-tac-toe and more. Some appropriate restrictions would be that the game require perfect information (no shadowed areas of the map), be generalized in some way[3] and a polynomial bound on the depth of number of moves. Without this bound many of these games are actually EXPTIME-complete although their proofs are less general.

Because of how we can interpret TQBF vs SAT, we can also intuitively say that games are harder than puzzles. We may characterize PSPACE by the problem of determining who is winning in some two-player game. There is no short certificate to convince a verifier one person is winning over another unless NP = PSPACE, and we don't think that'll happen.

[3]Recall that chess is played on a fixed game with a fixed number of pieces. There is no way to measure its complexity as a function of some n, as its technically a finite game. Generalized chess is proven to be PSPACE-complete if you generalize the board size as a function of n.