CS 4510 Automata and Complexity 4/13/2023

Lecture 19: Savitch's Theorem

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

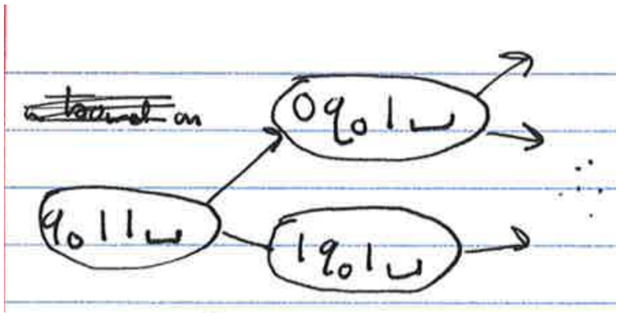
Recall $\mathsf{TIME}(f(n)), \mathsf{SPACE}(f(n))$ are the classes of languages decidable in $f(n)$ time or space, respectively. We prove the following containment chain:

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

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.



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 an algorithm which runs in $2^{O(f(n))}$ deterministic time. 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

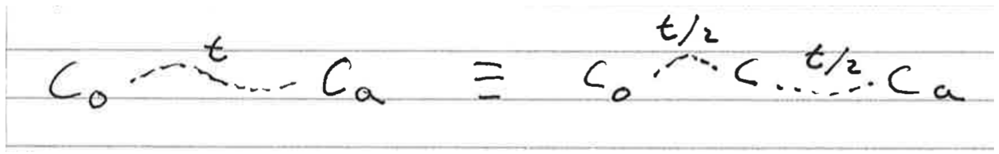
Our main result today:

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

with some conditions on f. Let us first interpret this result. We somehow are able to "de-nondeterminisfy" something 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.

Now onto the proof. Rather than some naive strategy, we are going to use divide and conquer. We want to simulate a nondeterministic Turing machine N which uses $f(n)$ space, deterministically using no more than $f^2(n)$ space. If C_o is the start configuration, C_a is the accept configuration, and C is some other 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.

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. For each recursive call, a stack frame containing C_i, C_j, t is stored. Each level of recursion uses $O(f(n))$ space, as a worst case. C_i, C_j are of size $O(f(n))$ since N uses $f(n)$ space. Each level divides $t = 2^{df(n)}$ in half. It may help if you recall anything about the Master theorem, or even geometric series. Here we won't measure

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 $C_i \vdash C_j$ in one step by $\delta$ 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
```

time, but space. The depth of our recursion tree is $\log t = log(2^{df(n)}) = O(f(n))$. 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 $O(f(n)) \cdot O(f(n)) = O(f^2(n))$

I mentioned that there were some restrictions on $f(n)$. First 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 was super-linear, that $f(n) \geq n$. This can be improved to $f(n) \geq log(n)$ with some automata specification. A final remark, Hartmanis came up with a similar idea but to prove a theorem about context-free languages[1].

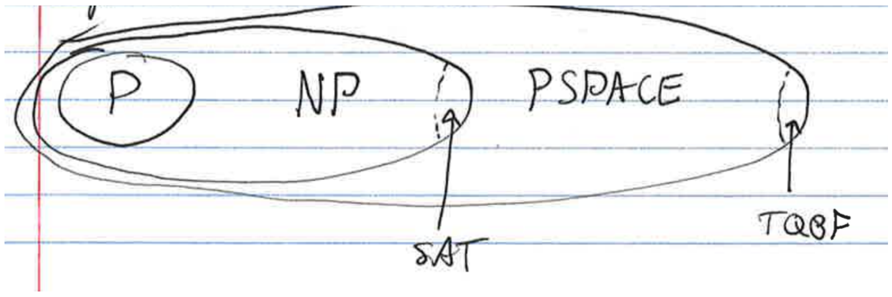
3 PSPACE-completeness

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

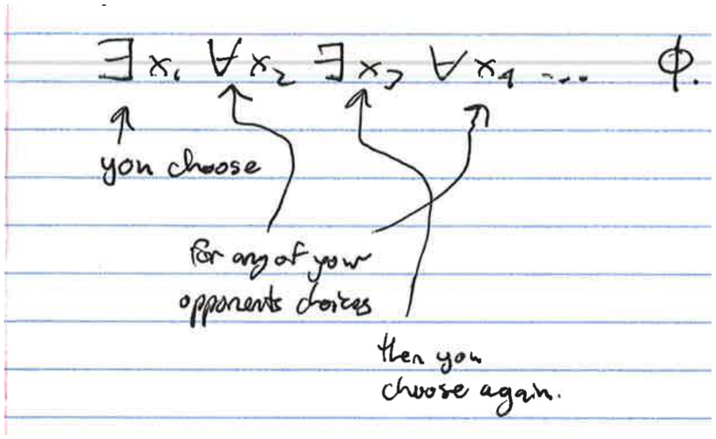
[1]See this post by Lipton for some fascinating history of the theorem `https://rjlipton.wpcomstaging.com/2009/04/05/savitchs-theorem`

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.



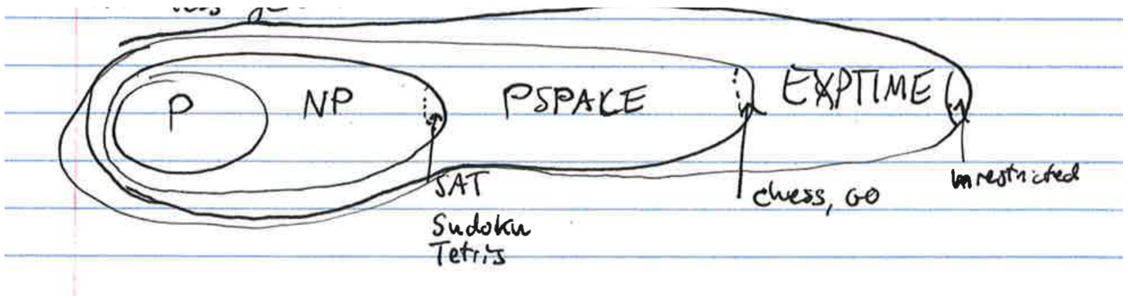
Notice SAT has structure like most puzzles. A puzzle is a single-player device in which you make a sequence of decisions to reach some goal. Intuitively, $\exists x_1, \exists x_2, ...$ is your sequence of decisions. Many puzzles are NP-complete since they can encode this structure.

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 $\exists\forall\exists\forall$... With a little abuse of types, you 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.


$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \dots \phi.$
you choose
for any of your opponents choices
then you choose again.

Most two player games, under appropriate restrictions and generalizations, are PSPACE-complete. Chess, checkers, Go and more. Some appropriate restrictions would be that the game require perfect information (no shadowed areas of the map), be generalized in some way[2] 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.

[2]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.



Because of how we can interpret TQBF vs SAT, we can also intuitively say that games are harder than puzzles. It should require a proof that TQBF is PSPACE-complete, like we did for SAT, but we don't have enough time.