Savitch's Theorem

Last time (before the exam) 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 $P \neq NP$ then there exist languages $\in P, \in NP$ and not NP-complete.

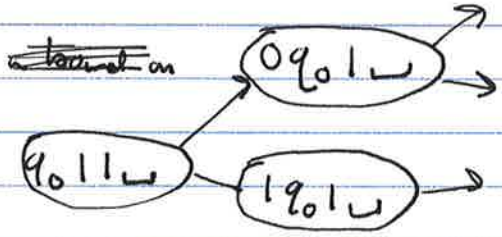
Todays lecture will be on space, that "other" resource. Space is a very different resource than time. After an algorithm finishes, you get the space back. You can never get the time back. This makes space a ~~more~~ less interesting resource to study. This also lets us use techniques and tricks which would not work for time.

Recall $TIME(f(n))$, $SPACE(f(n))$ are the classes of languages decidable in $f(n)$ time/space. Note the following chain

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

- Consider a language decidable in $f(n)$ time. There exists a Turing machine which takes $f(n)$ steps to decide this language. 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.

- We show a stronger result for the second one, with
$$SPACE(f(n)) \subseteq NSPACE(f(n)) \subseteq TIME(2^{O(f(n))})$$
The first containment following from the generalization of nondeterminism. We now show the second containment in a creative way. Given a language decidable by a nondeterministic 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 has a bound on the possible number of ~~edges~~ vertices.



Note that we do not count the input as part of the space used. Sometimes the input is on a separate write only tape. Since our machine N is nondeterministic, our graph may have arity greater than one. We may assume it has arity no more than two. In order to show $NSPACE(f(n)) \subseteq TIME(2^{O(f(n))})$ we give our $2^{O(f(n))}$ time deterministic algorithm. First using N, we build the configuration graph. Then we BFS from the start configuration C_0 to an accepting one C_a. BFS is linear time in the size of the input. This graph has worst case $2^{O(f(n))}$ nodes. It also takes this long to build it, so we see this is a time $2^{O(f(n))}$ deterministic algorithm. So $NSPACE(f(n)) \subseteq TIME(2^{O(f(n))})$.

Savitch's theorem, our main result today:

$$NSPACE(f(n)) \subseteq SPACE(f(n)^2)$$

with some conditions on f. Whoa hold on. Lets first interpret this result. We somehow are able to "de-nondeterminisfy" something with only polynomial overhead. Could such a technique apply to P vs NP? Probably not, or someone would have found it by now. So although we only get a polynomial space cost, we probably get a super-polynomial, ie, exponential time cost. Our deterministic algorithm may only use $f^2(n)$ space, but it should use $2^{f(n)}$ time.

A second immediate remark is that since polynomials are closed under composition, multiplication, we see $NPSPACE = PSPACE$. This world, the study of space, already looks very different. This is analogous to P vs NP, but unlike that problem, the result is unexpected, and we have been able to solve it. Now onto the proof.

Rather than some naive strategy, we are going to employ <u>divide and conquer.</u> We want to simulate a nondeterministic TM N which uses $f(n)$ space, deterministically using no more than $f^2(n)$ space. If C_0 is the start, C_a is the accept configurations and C is some other configuration, notice that $C_0 \vdash^* C_a$ in t steps if $C_0 \vdash^* C$ in $t/2$ steps and $C \vdash^* C_a$ in $t/2$ steps for some C.

$$C_0 \overset{t}{\cdots} C_a \equiv C_0 \overset{t/2}{\cdots} C \overset{t/2}{\cdots} C_a$$

This will be our divide-and-conquer recurrence. Importantly, our recursive calls are run sequentially and reuse space.

```
def YIELDS(Ci, Cj, t):
    if Ci = Cj: return true
    if Ci ⊢ Cj: (t=1) in one step by the rules of δ of N return true
    if t > 1:
        foreach C configuration of N using f(n) space:
            YIELDS(Ci, C, t/2)
            YIELDS(C, Cj, t/2)
                if both accept, accept
    reject
```

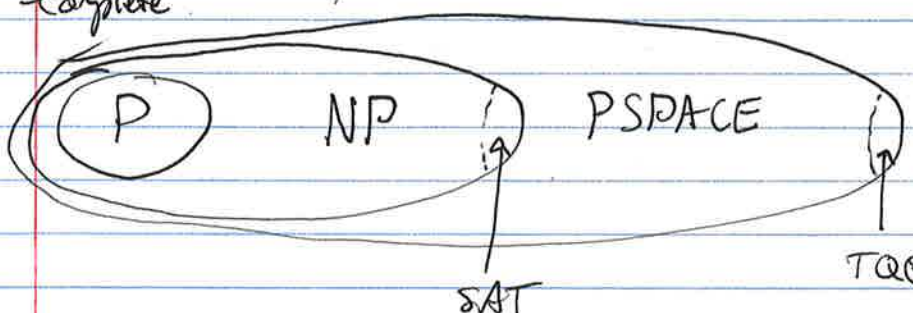
```
M on input w:
    C0 = start configuration of N
    Ca = accept configuration after tape head clears tape
    d = chosen so N has no more than 2^(df(n)) configurations
    return YIELDS(C0, Ca, 2^(df(n)))
```

It certainly is correct. M is a deterministic simulator of N, so it decides the same language. Now onto the analysis.

We hope to show M simulates N using 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 the recursion uses $O(f(n))$ space, as worst case, C_i, C_j are $O(f(n))$ since N uses $f(n)$ space. Each level of the recursion divides $t = 2^{df(n)}$ in half. If you recall anything about the Master theorem, or even geometric series, the depth of our recursion tree is $\log t = \log 2^{df(n)} = O(f(n))$. Since each level of the recursion 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 there were some restrictions on $f(n)$. First is that we may assume it is space-constructible. That M can compute $f(n)$ with $O(f(n))$ 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.

Recall SAT is NP-complete, a boolean formula is like $(x_1 \vee x_2 \vee x_3)$ right. This is not a boolean so much as it is a logical formula! We just hide the quantifiers, like $\exists x_1 \exists x_2 \exists x_3 (x_1 \vee x_2 \vee x_3)$. What if we allow for any quantifiers? Like $\forall x_1 \forall x_2 \exists x_3 (x_1 \vee x_2 \vee x_3)$? This is called TQBF True ~~quatified~~ quantified boolean formula. TQBF $= \{\phi \mid \phi$ is a true quantified boolean formula$\}$. Turns out how SAT is NP-complete, TQBF is PSPACE-complete.



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, $\exists x_1, \exists x_2, \ldots$ is you making your decisions. Most puzzles are then NP-complete.

Notice TQBF has structure like two player games of perfect information. Consider a TQBF like $\exists\exists\forall\forall\forall\exists\exists\ldots$ you can compress this into like $\exists\forall\exists\forall\ldots$ right?
$\exists x_1 \exists x_2 \equiv \exists (x_1, x_2)$ with a little abuse of types.
Playing a TQBF with alternating quantifiers looks like a game! Its quite literally minimax.

$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \quad \phi.$

↑ you choose

for any of your opponents choices

then you choose again.

Most two player games, under appropriate restrictions, are PSPACE-complete. Chess, checkers, Go, and more. Appropriate restrictions would be that the game require perfect information (no shadowed areas of the map) and a polynomial bound on the depth of the number of moves. Without this bound many of these games are actually EXPTIME-complete, although the proofs are less general their