Recitation 10: Space Complexity

In this recitation, we'll review the definitions of the space complexity classes we've seen so far and the intuition behind the proof of Savitch's theorem, and then we'll show that generalized geography can be solved in PSPACE.

Space complexity classes

Recall that $\mathsf{SPACE}(s(n))$ is defined as the set of languages decided by a *deterministic* Turing machine that uses only $O(s(n))$ tape cells. Likewise, we defined $\mathsf{NSPACE}(s(n))$ to be the set of languages decided by an *nondeterministic* Turing machine that uses only $O(s(n))$ tape cells in *every* thread of nondeterminism (similar to how we defined NTIME).

Definition 1.

$$\mathsf{PSPACE} = \bigcup_k \mathsf{SPACE}(n^k)$$
$$\mathsf{NPSPACE} = \bigcup_k \mathsf{NSPACE}(n^k)$$

These definitions use the standard single-tape Turing machine model that we've used so far in this class. However, this breaks down when we consider *sublinear* (i.e., $o(n)$) space complexity bounds; for example, $\mathsf{SPACE}(\log n)$ would fail to make sense since a single-tape Turing machine bounded to using only $\log n$ cells won't even have enough space to store its entire input string.[1]

To remedy this, we shift our definitions of SPACE and NSPACE to use a different model of computation: a two-tape Turing machine with a read-only input tape and a limited-capacity read/write work tape (Figure 1 shows the case where the work tape is limited to $O(\log n)$ cells). We then define $\mathsf{SPACE}(s(n))$ as the set of languages decidable by such a machine where the size of the work tape is $O(s(n))$; this allows us to have a well-defined notion of space complexity even when $s(n) < n$.

It turns out that the updated definitions of PSPACE and NPSPACE (and any other $\Omega(n)$ space complexity class) agree with our old ones,

[1] Strictly speaking, such a machine would be bounded to $k \log n$ cells for some constant k (independent of n). But no matter the value of k, for sufficiently large n we will have $k \log n < n$.

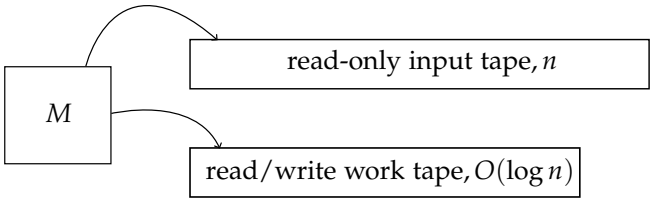

Figure 1: 2-tape Turing machine that is restricted to space $O(\log n)$.

so this definition serves as a natural extension of our notion of space complexity to the sublinear setting. To wit, we can now use this to define the classes L and NL:

Definition 2.

$$\mathsf{L} = \mathsf{SPACE}(\log n)$$
$$\mathsf{NL} = \mathsf{NSPACE}(\log n)$$

Seeing how we're defining sublinear space bounds, a natural question might be why we don't have something similar for time? Turns out we do! Note that with a sublinear time bound, the machine doesn't have enough time to read the input. So when considering sublinear time we're generally talking about *approximation* algorithms.

We can think of L as the class of problems that can be solved using only a constant number of pointers (that reference a part of the input) and counters (counting up to a quantity that is polynomial in the input length), since it takes $\log n$ bits to store an index between 0 and n. In particular, recall that the language

$$PATH = \{\langle G, s, t\rangle : G \text{ is a directed graph with a path from } s \text{ to } t\}$$

is in NL, since an NL machine can simply guess a sequence of edges starting from s and only needs to store the most recently visited node and the target t.

You may ask why these complexity classes are interesting — given the parallelism between L/NL and P/NP, results concerning L and NL might bring us closer to solving the $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$ problem. (For example, as you'll see in lecture, we know that $\mathsf{NL} = \mathsf{coNL}$, but $\mathsf{NP} \stackrel{?}{=} \mathsf{coNP}$ is still an open question.)

Savitch's theorem

Given this, let us revisit Savitch's theorem:

Theorem 3 (Savitch's theorem)**.** Let $f : \mathbb{N} \to \mathbb{R}^+$ be a function such that $f(n) \geq \log n$ for all n. Then, $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f^2(n))$.

In this class, the notation $f^2(n)$ always refers to $f(n)^2$, not $f(f(n))$.

Corollary 4. $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

This is a pretty surprising result! The naïve way of simulating nondeterminism would give us an exponential blowup in both space and time complexity, but it turns out that by being a little clever, we can

reduce the space blowup to be only quadratic. (In the time case, it's an open question whether we can do any better than exponential.)

The proof of Savitch's theorem involved considering a recursive algorithm for determining if a given configuration of our nondeterministic Turing machine yields a given end configuration within a certain number of steps; by repeatedly bisecting the search space, we were able to come up with an algorithm that requires only $O(f^2(n))$ space.

More details on that proof can be found in Section 8.1 of the textbook; instead of reviewing that, let's consider a simpler analogue of this theorem to build intuition about how the algorithm works.

Proposition 5. *PATH* $\in \mathsf{SPACE}(\log^2 n)$.

This is of course a simple corollary of Savitch's theorem (since *PATH* $\in$ NL), but let us try to prove it directly; the argument proceeds quite similarly to that of Savitch's theorem itself.

Proof. Consider the following algorithm for determining if a directed graph $G = (V, E)$ has a path from s to t that is of length $\leq k$.

> CANREACH$_G$ = "On input s, t, k:
>
> 1. If $k = 1$, then check if $s = t$ or $(s, t) \in E$. If either check passes, then **accept**; otherwise, **reject**.
> 2. If $k > 1$, then for each $v \in V$:
>   i. Recursively call CANREACH$_G$ on $(s, v, k/2)$ and $(v, t, k/2)$. If both calls accept, then **accept**.
> 3. **Reject**."

The correctness of this algorithm follows from the fact that every path of length k can be split into its first half and its second half.[2] We can determine the space complexity by setting bounds on how much space is taken before each recursive call as well as the maximum depth of the recursion. The first is simply the inputs to the recursive call, which take a total of $O(\log |V| + \log k)$ space to store; as for the maximum recursion depth, that will be $\log k$ since we divide k by 2 on each recursive call and end up on the base case when $k = 1$.

So, CANREACH$_G$ runs in space $O((\log |V| + \log k) \cdot \log k)$. We're not quite done yet; we now need a Turing machine that is given G as input and determines if G has a path of arbitrary length from s to t.

Since any path from s to t can be reduced to eliminate cycles, it suffices to decide whether G has a path from s to t that does not repeat any vertices; i.e., we can set $k = |V|$. This gives us the following machine:

[2] This assumes that k is even for simplicity, but we could just as easily relax such an assumption by using $\lfloor k/2 \rfloor$ in the first call and $\lceil k/2 \rceil$ in the second call.

M = "On input $\langle G, s, t \rangle$:

1. Let $G = (V, E)$.
2. Run CANREACH$_G$ on $(s, t, |V|)$. If it accepts, **accept**; if it rejects, **reject**."

By our analysis from earlier, this only requires space $O(\log^2 n)$. □

The CANYIELD algorithm in the proof of Savitch's theorem uses the same core idea; hopefully this sheds some light on how that works.

Generalized geography

Geography is a game where two players take turns naming locations. Each location named must start with the last letter of the previous location, and locations cannot be repeated. So if Alice and Bob are playing the game and Alice says "Boston," then Bob can say "Naples," Alice can then say "Shanghai," so on and so forth. A player loses when they are unable to respond, which means that the other player wins.

This game can be modeled as a directed graph! Each vertex in the graph corresponds to the name of a location. For two locations ℓ_i and ℓ_j, if ℓ_j starts with the same letter that ℓ_i ends with, then there is a directed edge $\ell_i \to \ell_j$. Alice and Bob take turns selecting vertices in the graph to produce a path. If a player ever ends up in a state where they cannot extend the path without repeating a previously selected vertex, then that player loses and the other player wins.

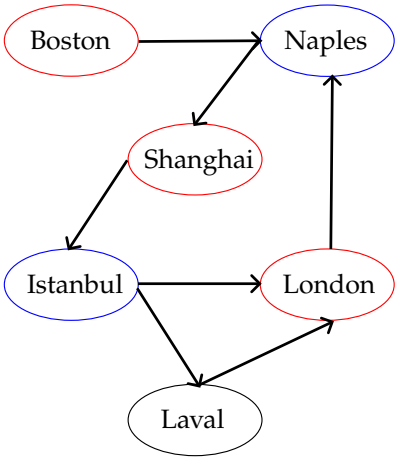

Figure 2: A game of geography modeled as a directed graph, with players Alice and Bob take turns selecting vertices. Alice is represented as red and Bob is represented as blue.

The game played is: Boston, Naples, Shanghai, Istanbul, London. After Alice selects London, Bob loses because he cannot select any other vertices.

The game of *generalized geography* (*GG*) has the same rules, except Alice and Bob are now playing on an *arbitrary* directed graph G with

a fixed starting vertex s. We say that a player has a *winning strategy* starting from s if they can always win even if the other player plays optimally. In other words, Alice has a winning strategy if regardless of what Bob does, Alice can always counter Bob's moves to force a win. We can write this as a language:

$$GG = \{\langle G, s\rangle : \text{Alice has a winning strategy starting from } s\}.$$

Proposition 6. $GG \in$ PSPACE.

The main idea is to reframe "Alice has a winning strategy" as the equivalent statement "There exists a move that Alice can make such that Bob does not have a winning strategy after that move."

Proof. Consider the following recursive algorithm for GG:

> $M =$ "On input $\langle G, s\rangle$:
>
> 1. If s has no outgoing edges, then Alice cannot win, so **reject**.
> 2. For every vertex v such that there exists an edge $s \to v$ in G:
>   i. Remove all edges that include s in G to form G'.
>   ii. Recursively call M on $\langle G', v\rangle$.
> 3. If all of the calls accept, **Reject**. Otherwise (if at least one call rejects), **Accept**.

Players take turns selecting vertices in the game, so the recursive call at step 2.ii shifts the perspective from Alice to Bob. The value returned from each call answers whether *Bob has a winning strategy given that Alice selected vertex v*. If all of the calls accept, that means Alice cannot select any vertex to prevent Bob from having a winning strategy, so she does not have a winning strategy. If at least one call rejects, that means there is a way for Alice to select a vertex to prevent Bob from having a winning strategy, so she has a winning strategy.

To show that GG is in PSPACE, M must run in polynomial space. This algorithm is a depth first search, so the space used depends on what is stored on the recursion stack. Since we remove one vertex before each recursive call, the maximum depth of recursion is $|V|$. Before making each call, the only information we need to store is one vertex v,[3] which takes $\log|V|$ space, so M requires $O(|V|\log|V|)$ space, which is indeed polynomial in the input size. □

Theorem 7. GG is PSPACE-hard.

Proof. We proved this during lecture by giving a polynomial time reduction from the PSPACE-complete problem *FORMULA-GAME*. □

Corollary 8. GG is PSPACE-complete.

It's also worth thinking about this game in terms of quantifiers. There exists a first move for Alice, such that whatever the first move of Bob, Alice can counter it, whatever the second move of Bob, Alice can counter it, ... all the way until Alice counters Bob's last move. In general, a poly-time solvable problem behind polynomially many quantifiers is in PSPACE; that's the essence of what *TQBF* $\in$ PSPACE (or *FORMULA-GAME* $\in$ PSPACE) tells us.

[3] This is because we don't have to actually construct G' — it suffices to store the set of visited nodes and reference that whenever we reach step 1 in the algorithm. This detail doesn't make a huge difference for the proof, though; if we stored the entire graph G' at each step, then the space complexity would come out to $O(n^2)$ rather than $O(n\log n)$.

Aside: why $\leq_P$?

A natural question to ask at this point in the course is why the definition of PSPACE-completeness uses the seemingly arbitrary notion of poly-time reducibility rather than something else (perhaps a notion of poly-space reducibility?).

When we call a language A "complete" for a complexity class C, this is not merely a property of the language and the complexity class; the idea of "completeness" is always relative to some weaker model of computation as compared to C. This is evident in how we motivated NP-completeness as a concept: we care that *SAT* is NP-complete since that means that a polynomial-time algorithm for *SAT* would allow us to solve any NP problem with only polynomial overhead (which means we can stay in P).

Similarly, if we had a polynomial-time algorithm for a PSPACE-complete problem such as *TQBF*, then we want to have the property that this gives us a polynomial-time algorithm for every problem in PSPACE, since we restrict the reductions to run in polynomial time.

In particular, if we were to use polynomial-space reductions, then not only would this lead to unhelpful circular reasoning (to use a poly-time *TQBF* algorithm to solve other PSPACE problems, we first need a way to run the poly-space reduction in poly time), but it also ends up implying that almost every problem in PSPACE is PSPACE-complete under this definition!

Soon, you'll see a weaker form of reduction in class, called a logspace reduction ($\leq_L$). It is certainly possible to define PSPACE-completeness in terms of $\leq_L$ instead of $\leq_P$, and this would indeed give rise to a stronger notion of what it means for a problem to be PSPACE-complete.

Ultimately, the choice of which weaker model of computation to use is dictated by which one we are most interested in for our purposes. For NP, of course, the choice of P is pretty clear, but in principle we could go with something weaker.

In particular, if a language A is known to be PSPACE-complete with respect to $\leq_L$, then $A \in L$ would imply that $L = PSPACE$, something that would not follow if we only knew that A is PSPACE-complete with respect to $\leq_P$. It turns out, though, that we already know that $L \subsetneq PSPACE$ (you'll see this later in the course!), so the practical utility of PSPACE-completeness under $\leq_L$ is limited.