Recitation 09: Space Complexity, Savitch’s Theorem, TQBF

In today’s recitation, we will discuss the notion of space complexity and develop some intuition for how it relates to the more common notion of time complexity. We will prove a key result, Savitch’s Theorem, which gives us the surprising result that nondeterministic space is within a polynomial factor of deterministic space, and thus NPSPACE = PSPACE. We will also discuss a canonical polynomial space example, TQBF, and develop a definition of polynomial space completeness, for which we’ll prove TQBF is a member. Finally, for those interested, there is a proof of MIN-FORMULA ∈ PSPACE, in the appendix.

Space Complexity

Much like time, space is a resource used by Turing machines over the course of their computation. Since Turing machines have just a finite control, for certain languages they will inevitably need to use the tape as a “workspace”. Thus, just as we defined time complexity over the running time, or number of steps a Turing machine takes on an input of length n, we now define space complexity similarly, for the amount of tape the machine uses.

**Definition 1.** If M is a Turing machine halting on all inputs, then the space complexity of M is the function \( f : \mathbb{N} \rightarrow \mathbb{N} \), such that \( f(n) \) is the maximum number of tape cells that M uses on any input of length n.

* If M is nondeterministic, \( f(n) \) must be the maximum number of tape cells used on any branch.

With this notion of space complexity, we can now define a class of languages based on the space required to decide them. Here we will use big-O notation - see the recitation 7 notes for a refresher, if needed.

**Definition 2.** Let \( f(n) \) be a function, \( f : \mathbb{N} \rightarrow \mathbb{N} \). \( \text{SPACE}(f(n)) \) is the set of languages, A, such that some deterministic Turing machine decides A in \( O(f(n)) \) space.

**Definition 3.** Let \( f(n) \) be a function, \( f : \mathbb{N} \rightarrow \mathbb{N} \). \( \text{NSPACE}(f(n)) \) is the
set of languages, $A$, such that some nondeterministic Turing machine decides $A$ in $O(f(n))$ space.

In short, $A \in \text{SPACE}(f(n))$ if there exists a deterministic decider, $M$ such that $L(M) = A$ and $M$ runs in $O(f(n))$ space, with space defined as in Definition 1. Additionally, $A \in \text{NSPACE}(f(n))$ if there exists a nondeterministic decider, $N$ such that $L(N) = A$ and $N$ runs in $O(f(n))$ space (on every branch).

One of the most interesting connections to analyze is that between space and time complexity. Let’s look at two key results:

1. A deterministic TM running in time $t$, can take at most $O(t)$ space.
2. A deterministic TM running in space $n$, can take at most $2^{O(n)}$ time.

It is important to understand why the above relations hold. For result 1, note that even the most space-wasteful TM cannot traverse more than $O(t)$ tape in $t$ steps, as our definition of “reasonable” models of computation presume a constant amount of work done per time-step. Result 2 might seem less intuitive, but it follows from the fact that, given a limited amount of space, a TM can only do so many things before it encounters a repeated configuration. In fact, we can analyze the exact number of unique configurations for a TM, $M$, using $n$ space. This value corresponds to the product of the number of unique states in $M$’s finite control, $|Q|$, the number of possible head positions, $n$, and the number of possible tape contents, $|\Gamma|^n$. Thus, a TM operating in $n$ space, can only run for $|Q| \times n \times |\Gamma|^n$ time, otherwise it would loop!

The above results seem to imply that space is somehow more powerful than time. Concretely, one can reuse space, but one cannot reuse time. Still, many results connecting space complexity and time complexity are yet-unproven, including whether $P \subseteq PSPACE$, which we will define in the next section.

**Polynomial Space**

Similarly to the classes $P$ and $NP$ for polynomial time on deterministic and nondeterministic TMs, respectively, we can define the analogous $PSPACE$ and $NPSPACE$ for space complexity.

**Definition 4.**

$$PSPACE = \bigcup_k \text{SPACE}(n^k).$$

**Definition 5.**

$$NPSPACE = \bigcup_k \text{NSPACE}(n^k).$$
Let’s do a quick illustrative example, showing $SAT \in \text{PSPACE}$ to get a better feel for polynomial space complexity.

**Example 1.** Show $NP \subseteq \text{PSPACE}$.

**Solution 1.** We begin by showing $SAT \in \text{PSPACE}$, we directly provide a Turing Machine using only $O(n)$ space. The deterministic TM, $M$, for input $\phi$, tries all possible truth assignments of the variables, reusing space on every assignment, and accepting if one works. Since we only need to store the current assignment, this algorithm is in $\text{SPACE}(n) \subseteq \text{PSPACE}$.

By the Cook-Levin Theorem, we know for all $A \in \text{NP}$, $A \leq_p SAT$, and a polynomial-time computable function can only use polynomial space, thus, for all $A \in \text{NP}$, $A \in \text{PSPACE}$, thus proving the desired result.

The above example, in addition to Savitch’s Theorem, which we will discuss in a later section, gives us the following image of the decidable languages:

![Figure 1: A schematic diagram of class membership, with a few key examples in each class.](image)

Regarding the above diagram, first note that the differentiation between language classes is only conjecture - it is yet unproven whether any of these containments are strict.

**Savitch’s Theorem**

**Theorem 1.** For any $f(n) \geq n$:

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

The above theorem is a potentially unintuitive result. In terms of time, it is conjectured that nondeterministic polynomial time is more
powerful than deterministic polynomial time, however the same does not seem to hold for space. Namely, anything that can be done in some $O(f(n))$ amount of space on a nondeterministic TM can be done using just a squared factor worth of overhead on a deterministic TM!

As a first attempt at proof, let’s try to simulate some nondeterministic TM, $N$, such that $L(N) \in \text{NSPACE}(f(n))$, via a deterministic TM, $M$. This simulation could operate as follows, updating each branch’s tape according to $N$’s rules:

| Branch 1 Tape | # | Branch 2 Tape | # | ⋯ |

We know that each branch can only use $O(f(n))$ space, however the issue with this approach arises in the number of potential branches. A nondeterministic TM using $O(f(n))$ space per branch, can run for maximally $2^{O(f(n))}$ time per branch. Since each timestep can have a corresponding branch split, this leads to a potentially exponential number of branches, hence $M$ will have used an exponential amount of space to store all of them!

In seeking a more efficient approach, we can look to a closely-related problem, called yieldability. In this problem setting, we want to determine whether a configuration, $c_1$ of a Turing machine, can yield a configuration $c_2$, within $t$ steps, using only valid transitions from the Turing machine’s transition function. We denote this $c_1 \xrightarrow{t} c_2$. Then, to simulate a nondeterministic machine, $N$, taking $O(f(n))$ space on input $w$, such that $|w| = n$, we just need to define a deterministic Turing machine, $M$ which accepts if and only if the starting configuration leads to the accepting configuration in the maximum allowed number of steps!

**Proof.** Let $N$ be an NTM deciding language $A$ using $O(f(n))$ space per branch. Now, we define DTM, $M$, deciding $A$ using $O(f^2(n))$ space. $M =$ “on input $w$:

1. Modify $N$ such that just before accepting, $N$ erases its tape and moves its head to the start position.

2. Define the following routine, $YIELD(c_i,c_j,t)$
   
   (a) If $t = 1$, if $c_i = c_j$ or $c_i$ yields $c_j$ in one step according to the rules of $N$, accept. Else, reject.

   (b) If $t > 1$, for each $c_{mid}$ in the set of potential configurations for $N$ using space $f(n)$, call $YIELD(c_i,c_{mid},\frac{t}{2})$ and $YIELD(c_{mid},c_j,\frac{t}{2})$. Accept if both accept, else continue.

   (c) If have not accepted yet, reject.

This problem setting is related to the LADDERDEA problem we saw in class. The key difference is that instead of changing one letter at each step from TRICK to TREAT, we now make one “step” according to a TM’s transition function.

This makes $c_{accept}$ unique, having the form $c_{accept} = q_{accept} \sqcup \sqcup \ldots$.

A key observation here is that the parent function does not get to call the second $YIELD$ recursive call until the first has returned! This ensures we can effectively reuse space.
3. Return the outcome of \( \text{YIELD} \left( c_{\text{start}}, c_{\text{accept}}, 2^{O(f(n))} \right) \)."

To analyze this algorithm, first let’s consider what information needs to be stored on the stack when the function \( \text{YIELD} \) calls itself. A reasonable choice of information to store are the values of \( c_i, c_j, t, \) and the choice (or index) of \( c_{\text{mid}} \). Note that each configuration represents a snapshot of the machine’s state and tape contents, thus it is bounded in length. Take \( c_{\text{start}} \) and some arbitrary \( c_{\text{ex}} \), for example:

\[
c_{\text{start}} = q_0 w_1 w_2 \ldots w_n \quad \text{Space } n+1
\]
\[
c_{\text{ex}} = w_1 w_2 q_7 \ldots \quad \text{Space } O(f(n))
\]

We know the size of the tape contents is bounded by \( O(f(n)) \), thus our configurations are bounded as well. Next, let’s consider the number of possible configurations we might encounter. We already know that an NTM using space \( O(f(n)) \) per branch, can only take time \( 2^{O(f(n))} \) for each branch. Thus, the maximum number of unique configurations that can be attained in a bounded space of this size is also \( 2^{O(f(n))} \).

Note that \( \text{YIELD} \) makes a recursive call which cuts the size of the subproblem in half at each step. Thus, after a sequence of calls, the stack might look as so:

```
| \( c_i \) | \( c_j \) | \( c_{\text{mid}} \) | \( t \) |
|\hline|
| \( c_i \) | \( c_{\text{mid}} \) | \( c_{\text{mid2}} \) | \( t/2 \) |
| \( c_i \) | \( c_{\text{mid2}} \) | \( c_{\text{mid3}} \) | \( t/4 \) |
| \( \circ \) \hspace{1cm} \( \circ \) \hspace{1cm} \( \circ \) |
| \( c_1 \) | \( c_2 \) | \( \Box \) | \( 1 \) |
```

The bottom of this recursive stack is the base case, and the height is \( \log_2 t = \log_2 2^{O(f(n))} = O(f(n)) \). Note that we can reuse the space on the stack to store the results of successive calls, as we are only ever traversing one chain of \( \log t \) function calls at a time! Thus, since we know the width and height of this stack are both \( O(f^2(n)) \), the amount of space needed by this deterministic decider for \( A \), is \( O(f^2(n)) \), thus \( A \in \text{SPACE}(f^2(n)) \). \( \square \)

It follows from Savitch’s Theorem that \( \text{PSPACE} = \text{NPSPACE} \), as every language decidable in \( n^k \) time on an NTM can be decided in \( n^{2k} \) time on a DTM, which are both polynomial in \( n \).

While there are exponentially many potential configurations, \( 2^{O(f(n))} \) of them, to be precise, an index for this number in binary would only need \( \log_2 2^{O(f(n))} \) bits, or simply \( O(f(n)) \) bits. If we enumerate in lexicographic, or some other order, we can easily translate indices to their corresponding \( c_{\text{mid}} \).

At first this might seem problematic, but the fact that we use a divide-and-conquer recursive approach ensures that we never materialize the entirety of the computation chain from \( c_{\text{start}} \) to \( c_{\text{accept}} \).

Figure 2: In order to keep track of the parent caller, \( \text{YIELD} \) must store some information about its state in an auxiliary portion of the tape, called the stack.
PSPACE Reductions

Definition 6. A language $B$ is PSPACE-complete if:

1. $B \in \text{PSPACE}$

2. For every $A \in \text{PSPACE}$, $A \leq_p B$

A natural question to ask is why we choose polynomial time computable functions for reductions in PSPACE. Consider a different choice: polynomial space computable functions. This might seem like a reasonable choice, however, given any two languages $A, B \in \text{PSPACE}$ (except the empty language and $\Sigma^*$), we can reduce $A$ to $B$ by just solving $A$!

Using a polynomial-time computable reduction alleviates this issue because it restricts the power of the function. A polynomial-time function, $f$, cannot simply run the decider for a PSPACE language, as the decider could take exponential time! Thus, the notion of completeness is best captured via a poly-time reduction.

TQBF

The TQBF problem is essentially an extension of the SAT problem, which allows the presence of quantifiers $\forall, \exists$ in the formula. These quantifiers correspond to the requirement that the boolean formula is satisfiable “for all” and “for any” of the quantified variables, respectively. Consider the following example:

Example 2. $\forall x \exists y \ [(x \lor y) \land (\overline{x} \lor \overline{y})]$. (For all assignments of $x$, i.e. $x = \text{True}$ and $x = \text{False}$, for every assignment of $y$, the statement is true.)

This TQBF instance happens to be satisfiable, but the example helps to illustrate the expressiveness of the language. Specifically, we can implicitly encode an “AND” statement in a “$\forall$” and we can implicitly encode an “OR” statement in a “$\exists$”. This will become relevant in the proof of PSPACE-completeness for TQBF.

Theorem 2. TQBF is PSPACE-complete.

Proof. We begin by giving an argument that TQBF $\in \text{PSPACE}$. The intuition is simply that we can recursively try all assignments of the variables and ensure that the quantifiers are satisfied. Concretely, we define a machine which operates as follows:

“on input $\langle \phi \rangle$:"

1. If $\phi$ has no quantifiers (only constants), evaluate $\phi$ and accept if True, reject if False.
2. If $\phi$ starts with an $\exists$ quantifier ($\phi = \exists x[\psi]$), evaluate $\psi$ with $x = True$ and $x = False$, recursively. Accept if either accepts.

3. If $\phi$ starts with an $\forall$ quantifier ($\phi = \forall x[\psi]$), evaluate $\psi$ with $x = True$ and $x = False$, recursively. Accept if both accept.”

This Turing machine uses $O(1)$ space per level, simply storing the truth value of the variable assigned. Furthermore, at each level a quantifier is removed, and thus the entire recursive stack is maximally $O(n)$ size. Thus, $\text{TQBF} \in \text{SPACE}(n) \subseteq \text{PSPACE}$. 

Now, for the more involved problem: showing $\text{TQBF}$ is PSPACE-hard. We present a reduction that is similar in nature to both the Cook-Levin Theorem, and Savitch’s Theorem. The goal is to show, for any $A \in \text{PSPACE}$, $A \leq_p \text{TQBF}$. Let $M$ be a Turing machine deciding $A$ in $O(n^k)$ space. We seek a polynomial-time computable function $f$, such that:

- If $w \in A$, then $f(w) = \phi_w \in \text{TQBF}$ is satisfiable.
- If $w \notin A$, then $f(w) = \phi_w \notin \text{TQBF}$ is not satisfiable.

To accomplish this, we turn to the construct used in the Cook-Levin Theorem: a way to represent a single configuration $c_i$ in a boolean formula. Let $\phi_{c_i,c_{i+1}}$ denote the formula which is true if and only if $c_i \rightarrow c_{i+1}$, according to $M$’s rules. Each cell in a configuration, $c_i$, is associated with a set of boolean variables representing potential tape symbols and states. Furthermore, each configuration has $O(n^k)$ cells. Thus, we can encode a boolean formula for $\phi_{c_i,c_{i+1}}$ as the formula $\phi_{cell}$ for each of the $O(n^k)$ cells in $c_i$ and $c_{i+1}$, as well as $\phi_{move}$, which ensures that the transition between adjacent cells is valid according to $M$’s rules (or that $c_i = c_{i+1}$). The entirety of this formula is $O(n^k)$ in length.

The above lays out how we might envision the base case of this formula, when $t = 1$. Now let’s deal with the case where $t > 1$. We proceed, recursively, quite like Savitch’s theorem. Let’s try an initial approach:

$$\phi_{c_i,c_{i+1},t} = \exists c_{mid} \left[ \phi_{c_i,c_{mid},t/2} \land \phi_{c_{mid},c_{i+1},t/2} \right].$$

Note that $c_{mid}$ is just an encoding for some configuration, in the sense that:

$$\exists c_{mid} \iff \exists x_1 \exists x_2 \ldots \exists x_T, \text{O(n^k)}.$$ 

This effectively allows us to encode an entire for-loop over an exponential space, in a polynomial amount of variables. Still, there is one issue. Note that this recursive formula has $O \left( \log 2^{O(n^k)} \right) = O(n^k)$ levels, as the maximum number of configurations is $2^{O(n^k)}$. However, at

Note that we only borrow from the Cook-Levin theorem. If we were to directly use the theorem’s construction to accept valid accepting tableaus of $M$ on $w$, the space needed to construct such a formula would be exponential, as $M$, which runs in space $O(n^k)$, can run for time $2^{O(n^k)}$. Thus, our computable function cannot compute this in polynomial time.

This reduction effectively gives us an exponentially-long SAT formula, as it only uses $\exists$. If we could construct a polynomial size SAT formula here, that would be a major result as it would show $NP = \text{PSPACE}$, which is unknown!
each level the size of our formula doubles, and thus the result is again exponential.

To deal with this issue, we introduce a $\forall$ clause. Instead of using a $\land$ to connect the clauses, we enforce that both are true via a $\forall$. This construct is inspired by the relationship discussed in example 2. We then simplify the formula as follows:

$$\phi_{c_i,c_j,t} = \exists c_{mid} \forall (c_A, c_B) \in \{(c_i,c_{mid}),(c_{mid},c_j)\} \left[ \phi_{c_A,c_B,t}^{c_{mid}} \right].$$

Another way to think of this formula is as:

$$\phi_{c_i,c_j,t} = \exists c_{mid} \forall c_A,c_B \left[ (c_A,c_B) = (c_i,c_{mid}) \lor (c_A,c_B) = (c_{mid},c_j) \right] \rightarrow \phi_{c_A,c_B,t}^{c_{mid}}.$$  

Now, each of the $O(n^k)$ levels of the recursion only adds $O(n^k)$ variables to the formula, as the recursive step only has one call to $\phi_{c_A,c_B,t}^{c_{mid}}$, instead of two calls as in the initial formulation. Thus, the total amount of variables in the resulting formula is $O(n^{2k})$, which is polynomial in $O(n^k)$, and is thus computable by $f$.

Appendix: MIN-FORMULA $\in$ PSPACE

MIN-FORMULA is the language of minimal-length boolean formulae. We say two boolean formulas are equivalent if they have the same set of variables and are true on the same set of assignments to those variables (they have the same truth table). A boolean formula is minimal if no shorter boolean formula is equivalent to it.

Example. MIN-FORMULA $\in$ PSPACE

Solution. We directly provide an algorithm to decide MIN-FORMULA using only polynomial space. Let $\phi$ be the input formula, with $|\phi| = n$. First, note that we can generate all possible shorter boolean formulas, using some fixed ordering, in polynomial space, as the number of unique boolean formulas over $\land$, $\lor$, and $\neg$ is $2^{O(n)}$, thus the size of the index needed in binary notation is just $O(n)$. Let’s assume $\psi$ is the proposed shorter boolean formula. Then, note that we can try all possible settings of variables in $O(n)$ space (just as we did for SAT $\in$ PSPACE), and ensure $\psi$ and $\phi$ return the same truth result! If we ever encounter a shorter boolean formula, $\psi$, such that the truth values are the same as those of $\phi$ for all variable settings, accept. If no formula is found, reject.

This algorithm operates in PSPACE, because to iterate over all smaller formulas and try all possible truth values, we can reuse the required $O(n)$ space. Thus, only using a polynomial amount of space in the length of the input, we can decide MIN-FORMULA, so MIN-FORMULA $\in$ PSPACE.