UMB CS622

Space Complexity

Wed, November 24, 2021
Announcements

• HW 9 due Sun 11:59pm EST
  • (after break)

• Happy Thanksgiving!
First: One More \textbf{NP}-Complete Problem

- \textit{SUBSET-SUM} = \{\langle S, t \rangle | S = \{x_1, \ldots, x_k\}, and for some \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, we have \Sigma y_i = t\}

  \begin{itemize}
    \item (reduce from \textit{3SAT})
  \end{itemize}

- \textit{VERTEX-COVER} = \{\langle G, k \rangle | G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}

  \begin{itemize}
    \item (reduce from \textit{3SAT})
  \end{itemize}
Theorem: *VERTEX-COVER* is NP-complete

- A vertex cover of a graph is ...
  - ... a subset of its nodes where every edge touches one of those nodes
THEOREM

If $B$ is NP-complete and $B \leq_P C$ for $C$ in NP, then $C$ is NP-complete.

3 steps to prove a language is NP-complete:
1. Show $C$ is in NP
2. Choose $B$, the NP-complete problem to reduce from
3. Show a poly time mapping reduction from $B$ to $C$
Theorem: VERTEX-COVER is NP-complete

3 steps to prove VERTEX-COVER is NP-complete:

1. Show VERTEX-COVER is in NP
2. Choose the NP-complete problem to reduce from: 3SAT
3. Show a poly time mapping reduction from 3SAT to VERTEX-COVER

To show poly time mapping reducibility:
1. create computable fn,
2. show that it runs in poly time,
3. then show forward direction of mapping red.,
4. and reverse direction (or contrapositive of forward direction)
Theorem: \textit{VERTEX-COVER} is NP-complete

- A vertex cover of a graph is ...
  - ... a subset of its nodes where every edge touches one of those nodes

Proof Sketch: Reduce \textit{3SAT} to \textit{VERTEX-COVER}

- The reduction maps:
- Variable $x_i \rightarrow$ 2 connected nodes
  - corresponding to the var and its negation, e.g.,
- Clause $\rightarrow$ 3 connected nodes
  - corresponding to its literals, e.g.,
- Additionally,
  - connect var and clause gadgets by ...
  - ... connecting nodes that correspond to the same literal
**VERTEX-COVER** example

\[ \phi = (x_1 \lor x_1 \lor x_2) \land (\overline{x_1} \lor \overline{x_2} \lor \overline{x_2}) \land (\overline{x_1} \lor x_2 \lor x_2) \]

**VERTEX-COVER** = \{ \langle G, k \rangle | G is an undirected graph that has a k-node vertex cover\}
\[ \phi = (x_1 \lor x_1 \lor x_2) \land (\overline{x_1} \lor \overline{x_2} \lor \overline{x_2}) \land (\overline{x_1} \lor x_2 \lor x_2) \]

**VERTEX-COVER example**

\[ \text{VERTEX-COVER} = \{ (G, k) | G \text{ is an undirected graph that has a } k \text{-node vertex cover} \} \]
**VERTEX-COVER** example

\[ \phi = (x_1 \lor x_1 \lor x_2) \land (\overline{x_1} \lor \overline{x_2} \lor \overline{x_2}) \land (\overline{x_1} \lor x_2 \lor \overline{x_2}) \]

Extra edges connecting variable and clause gadgets together
**VERTEX-COVER** example

- If formula has ...
  - $m = \#$ variables
  - $l = \#$ clauses
- Then graph has ...
  - # nodes $= 2 \times \# \text{vars} + 3 \times \# \text{clauses} = 2m + 3l$

$\Rightarrow$ If satisfying assignment, then there is a $k$-cover, where $k = m + 2l$

- Nodes in the cover are:
  - In each of $m$ var gadgets, choose 1 node corresponding to TRUE literal
  - For each of $l$ clause gadgets, ignore 1 TRUE literal and choose other 2
  - Since there is satisfying assignment, each clause has a TRUE literal
  - Total nodes in cover $= m + 2l$
**VERTEX-COVER** example

- If formula has ...
  - $m = \# \text{ variables}$
  - $l = \# \text{ clauses}$
- Then graph has ...
  - $\# \text{ nodes} = 2m + 3l$

$\Rightarrow$ If satisfying assignment, then there is a $k$-cover, where $k = m + 2l$

- Nodes in the cover are:
  - In each of $m$ var gadgets, choose 1 node corresponding to TRUE literal
  - For each of $l$ clause gadgets, ignore 1 TRUE literal and choose other 2
  - Since there is satisfying assignment, each clause has a TRUE literal
  - Total nodes in cover $= m + 2l$
**VERTEX-COVER** example

- If formula has ...
  - $m = \#\text{ variables}$
  - $l = \#\text{ clauses}$
- Then graph has ...
  - $\#\text{ nodes} = 2m + 3l$

$\iff$ If there is a $k = m + 2l$ cover,

- Then it can only be a $k$-cover as described on the last slide ...
  - 1 node (and only 1) from each of “var” gadgets
  - 2 nodes (and only 2) from each “clause” gadget
  - Any other set of $k$ nodes is not a cover

- Which means that input has satisfying assignment:
  - $x_i = \text{TRUE}$ if node $x_i$ is in cover, else $x_i = \text{FALSE}$
Last Time: **NP-Completeness**

**DEFINITION**

A language $B$ is **NP-complete** if it satisfies two conditions:

1. $B$ is in NP, and
2. every $A$ in NP is polynomial time reducible to $B$.

These are the “hardest” problems (in NP) to solve.
**NP-Completeness vs NP-Hardness**

**Definition**

A language $B$ is **NP-complete** if it satisfies two conditions:

1. $B$ is in NP, and
2. every $A$ in NP is polynomial time reducible to $B$.

“NP-Complete” $=$ in NP + “NP-Hard”

So a language can be NP-hard but not NP-complete!
Flashback: The Halting Problem

$HALT_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \}$

Thm: $HALT_{TM}$ is undecidable

Proof, by contradiction:

• Assume $HALT_{TM}$ has decider $R$; use it to create decider for $A_{TM}$:

  • ...

• But $A_{TM}$ is undecidable and has no decider!
Flashback: The Halting Problem

\[ HALT_{\text{TM}} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w \} \]

Thm: \( HALT_{\text{TM}} \) is undecidable

Proof, by contradiction:

- Assume \( HALT_{\text{TM}} \) has decider \( R \); use it to create decider for \( A_{\text{TM}} \):

\[
S = \text{“On input } \langle M, w \rangle, \text{ an encoding of a TM } M \text{ and a string } w:\n\
1. Run TM } R \text{ on input } \langle M, w \rangle. \\
2. If } R \text{ rejects, } \text{reject}.
3. If } R \text{ accepts, simulate } M \text{ on } w \text{ until it halts.}
4. If } M \text{ has accepted, } \text{accept}; \text{ if } M \text{ has rejected, } \text{reject}.”
\]

This means \( M \) loops on input \( w \)

This step always halts
Flashback: The Halting Problem

\[ \text{HALT}_\text{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w \} \]

Thm: \( \text{HALT}_\text{TM} \) is undecidable

Proof, by contradiction:

1. Assume \( \text{HALT}_\text{TM} \) has decider \( R \); use it to create decider for \( A_{\text{TM}} \):

   \[
   S = \text{“On input } \langle M, w \rangle, \text{ an encoding of a TM } M \text{ and a string } w:}
   \]
   \[1. \text{ Run TM } R \text{ on input } \langle M, w \rangle. \]
   \[2. \text{ If } R \text{ rejects, reject.} \]
   \[3. \text{ If } R \text{ accepts, simulate } M \text{ on } w \text{ until it halts.} \]
   \[4. \text{ If } M \text{ has accepted, accept; if } M \text{ has rejected, reject.”} \]

2. But \( A_{\text{TM}} \) is undecidable!
   1. i.e., this decider that we just created cannot exist! So \( \text{HALT}_\text{TM} \) is undecidable
The Halting Problem is $\textbf{NP}$-Hard

$HALT_{TM} = \{ \langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w \}$

Proof: Reduce $3SAT$ to the Halting Problem

(Why does this prove that the Halting Problem is $\textbf{NP}$-hard?)

Because $3SAT$ is $\textbf{NP}$-complete!
(\text{so every $\textbf{NP}$ problem is poly time reducible to $3SAT$})
The Halting Problem is \textbf{NP-Hard}

\[ \text{HALT}_\text{TM} = \{ \langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w \} \]

Computable function, from $3SAT \rightarrow \text{HALT}_\text{TM}$:

On input $\phi$, a formula in $3\text{cnf}$:

- Construct TM $M$

  $M = \text{on input } \phi$

  - Try all assignments
    - If any satisfy $\phi$, then accept
    - When all assignments have been tried, start over

- Output $\langle M, \phi \rangle$

  $\Rightarrow$ If $\phi$ has a satisfying assignment, then $M$ halts on $\phi$
  $\Leftarrow$ If $\phi$ has no satisfying assignment, then $M$ loops on $\phi$
Review:

DEFINITION
A language $B$ is **NP-complete** if it satisfies two conditions:

1. $B$ is in NP, and
2. every $A$ in NP is polynomial time reducible to $B$.

So a language can satisfy condition #2 but not condition #1

But can a language satisfy condition #1 but not condition #2?

Yes, every language in P ...

... unless $P = NP$

Can a non-P language satisfy condition #1 but not condition #2?

Yes ...

... but that implies $P \neq NP$, so it’s not known for sure
NP-Completeness vs NP-Hardness

NP-Complete

NP-Hard

P ≠ NP

P = NP

P = NP

≈ NP-Complete
On to Space ...
**Flashback:** Dynamic Programming Example

- Chomsky Grammar $G$:
  - $S \rightarrow AB \mid BC$
  - $A \rightarrow BA \mid a$
  - $B \rightarrow CC \mid b$
  - $C \rightarrow AB \mid a$

- Example string: **baaba**

- Store every **partial string** and their generating variables in a **table**

<table>
<thead>
<tr>
<th>Substring start char</th>
<th>Substring end char</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td>vars for “b”</td>
<td>vars for “ba”</td>
<td>vars for “baa”</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>vars for “a”</td>
<td>vars for “aa”</td>
<td>vars for “aab”</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*We are gaining time ...*

*... by spending more space!*
Space Complexity, Formally

**DEFINITION**

Let $M$ be a deterministic Turing machine that halts on all inputs. The *space complexity* of $M$ is the function $f: \mathbb{N} \rightarrow \mathbb{N}$, where $f(n)$ is the maximum number of tape cells that $M$ scans on any input of length $n$. If the space complexity of $M$ is $f(n)$, we also say that $M$ runs in space $f(n)$.

If $M$ is a nondeterministic Turing machine wherein all branches halt on all inputs, we define its space complexity $f(n)$ to be the maximum number of tape cells that $M$ scans on any branch of its computation for any input of length $n$. 

**TMs have a space complexity**
Space Complexity Classes

**Definition**

Let $f : \mathbb{N} \rightarrow \mathbb{R}^+$ be a function. The *space complexity classes*, $\text{SPACE}(f(n))$ and $\text{NSPACE}(f(n))$, are defined as follows.

\[
\text{SPACE}(f(n)) = \{ L \mid L \text{ is a language decided by an } O(f(n)) \text{ space deterministic Turing machine} \}.
\]

\[
\text{NSPACE}(f(n)) = \{ L \mid L \text{ is a language decided by an } O(f(n)) \text{ space nondeterministic Turing machine} \}.
\]

**Compare:**

Let $t : \mathbb{N} \rightarrow \mathbb{R}^+$ be a function. Define the *time complexity class*, $\text{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

\[
\text{NTIME}(t(n)) = \{ L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine} \}.
\]
Example: \textbf{SAT} Space Usage

$SAT = \{ \langle \phi \rangle | \phi \text{ is a satisfiable Boolean formula} \}$

$2^{O(m)}$ exponential time machine

$M_1 =$ “On input $\langle \phi \rangle$, where $\phi$ is a Boolean formula:

1. For each truth assignment to the variables $x_1, \ldots, x_m$ of $\phi$:
2. Evaluate $\phi$ on that truth assignment.
3. If $\phi$ ever evaluated to 1, accept; if not, reject.”

Each loop iteration requires $O(m)$ space

But the space is re-used on each loop! (nothing is stored from the last loop)

So the entire machine only needs $O(m)$ space!”
Example: Nondeterministic Space Usage

\[ ALL_{\text{NFA}} = \{ \langle A \rangle | \text{A is an NFA and } L(A) = \Sigma^* \} \]

Nondeterministic decider for \( ALL_{\text{NFA}} \):

\[ N = \text{“On input } \langle M \rangle, \text{ where } M \text{ is an NFA:} \]

1. Place a marker on the start state of the NFA.
2. Repeat \(2^q\) times, where \(q\) is the number of states of \(M\):
   3. Nondeterministically select an input symbol and change the positions of the markers on \(M\)’s states to simulate reading that symbol.
   4. Accept if stages 2 and 3 reveal some string that \(M\) rejects; that is, if at some point none of the markers lie on accept states of \(M\). Otherwise, reject.”

Additionally, need a counter to count to \(2^q\): requires \(\log(2^q) = q\) extra space.

Machine tracks “current” states of NFA:
\(q\) states = \(2^q\) possible combinations (so exponential time)

Each loop uses only \(O(q)\) space!

So the whole machine runs in (nondeterministic) linear \(O(q)\) space!
Flashback: TM Variations and Time

• If a multi-tape TM runs in: \( t(n) \) time
• Then an equivalent single-tape TM runs in: \( O(t^2(n)) \)
  • Quadratically slower

• If a non-deterministic TM runs in: \( t(n) \) time
• Then an equivalent deterministic TM runs in: \( 2^{O(t(n))} \)
  • Exponentially slower

What about space?
TM Variations and Space

**THEOREM**

**Savitch’s theorem**  
For any function $f : \mathbb{N} \rightarrow \mathbb{R}^+$, where $f(n) \geq n$,  
$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$.  

- If a **non-deterministic** TM runs in: $f(n)$ space  
- Then an equivalent **deterministic** TM runs in: $f^2(n)$ space  
  - **Exponentially Only Quadratically** slower!
Flashback: Nondet. TM $\rightarrow$ Deterministic TM

- Simulate NTM with Det. TM:
  - Number the nodes at each step
  - Deterministically check every tree path, in breadth-first order
    - 1
    - 1-1
    - 1-2
    - 1-1-1

$$b^t(n) = 2^{O(t(n))}$$
Flashback: NTM → Deterministic

3 tapes

Always has input, never changes: $O(n)$ space

Used to run each path (re-copy input here for each path): $O(n^k)$ space

Tracks which node we are on, e.g., 1-1-2, etc: $2^{O(n)}$ space??

D

0 0 1 0 □ ... input tape

x x # 0 1 x □ ... simulation tape

1 2 3 3 2 3 1 2 1 1 3 □ ... address tape
NTM→Deterministic TM: Space Version

Let $N$ be an NTM deciding language $A$ in space $f(n)$

• This means a single path could use $f(n)$ space

• That path could take $2^{O(f(n))}$ steps
  • (That’s the possible ways to fill the space)
  • Where each step could be a branch

• So naively tracking these branches requires $2^{O(f(n))}$ space!

• Instead, let’s “divide and conquer” to save space!
“Divide and Conquer” TM Config Sequences

• Want to check whether:

\[ 2^{O(f(n))} \text{ (possibly branching) steps} \]

This requires:

\[ \log(2^{O(f(n))}) = O(f(n)) \text{ splits} \]

Each split must remember 1 configuration

\[ "c_m" = O(f(n)) \text{ space} \]

• Instead, we check whether:

\[ 2^{O(f(n))}/2 \text{ steps} \]

So long as we save the intermediate config

\[ O(f(n)) \times O(f(n)) = O(f^2(n)) \text{ space} \]

• Keep dividing ...

\[ \text{Remembering these steps costs half the space …} \]

\[ \ldots \text{and we can reuse that space to check the second half} \]
Formally: A “Yielding” Algorithm

\[ \text{CANYIELD} = \text{“On input } c_1, c_2, \text{ and } t:\]

1. If \( t = 1 \), then test directly whether \( c_1 = c_2 \) or whether \( c_1 \) yields \( c_2 \) in one step according to the rules of \( N \). \text{Accept} if either test succeeds; \text{reject} if both fail.
2. If \( t > 1 \), then for each configuration \( c_m \) of \( N \) using space \( f(n) \):
3. Run \text{CANYIELD}(c_1, c_m, \frac{t}{2})
4. Run \text{CANYIELD}(c_m, c_2, \frac{t}{2})
5. If steps 3 and 4 both accept, then \text{accept}.
6. If haven’t yet accepted, \text{reject}.”

What’s the middle config? Try them all
(it doesn’t use any more space, per loop)

“divide and conquer”
Savitch’s Theorem: Proof

• Let $N$ be an NTM deciding language $A$ in space $f(n)$
• Construct equivalent deterministic TM $M$ using $O(f^2(n))$ space:

$$M = \text{“On input } w:\n 1. \text{ Output the result of } \text{CANYIELD}(c_{\text{start}}, c_{\text{accept}}, 2^{df(n)}).\text{“}$$

• $c_{\text{start}}$ = start configuration of $N$
• $c_{\text{accept}}$ = new accepting config where all $N$’s accepting configs go

Extra $d$ constant depends on size of tape alphabet
**PSPACE**

**DEFINITION**

**PSPACE** is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words,

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

**Definition**

\( \text{NPSPACE} \) is the class of languages that are decidable in polynomial space on a deterministic Turing machine. In other words,

\[
\text{NPSPACE} = \bigcup_{k} \text{SPACE}(n^k).
\]

Analogous to \( \text{P} \) and \( \text{NP} \) for time complexity
PSPACE vs NPSPACE

• **PSPACE**: langs decidable in poly space on deterministic TM

• **NPSPACE**: langs decidable in poly space on nondeterministic TM
Flashback: Does $P = NP$?

Proving $P \neq NP$ is hard because how do you prove an algorithm doesn't have a poly time algorithm? (in general it’s hard to prove that something doesn’t exist)
PSPACE vs NPSPACE

• **PSPACE**: langs decidable in poly space on deterministic TM

• **NPSPACE**: langs decidable in poly space on nondeterministic TM

**Theorem**: PSPACE = NPSPACE !!!

**Proof**: By Savitch’s Theorem!

**THEOREM**

**Savitch’s theorem**  For any function $f: \mathbb{N} \rightarrow \mathbb{R}^+$, where $f(n) \geq n$, \[ \text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)). \]
Space vs Time

- \( P \subseteq PSPACE \) and \( NP \subseteq NPSPACE \)
  - Because each step can use at most one extra tape cell
  - And space can be re-used

- \( PSPACE \subseteq \text{EXPTIME} \)
  - Because an \( f(n) \) space TM has \( 2^{O(f(n))} \) possible configurations
  - And a halting TM cannot repeat a configuration

- We already know \( P \subseteq NP \) and \( PSPACE = NPSPACE \) ... so:

\[
P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq \text{EXPTIME}
\]
Space vs Time: Conjecture

Researchers believe these are all completely contained within each other

But this is an open conjecture!

The only progress so far is: $P \subseteq \text{EXPTIME}$
(we will prove next week)

$P \subseteq NP \subseteq \text{PSPACE} = \text{NPSPACE} \subseteq \text{EXPTIME}$
No quiz 11/24!