CS 341: ALGORITHMS

Trevor Brown

trevor.brown@uwaterloo.ca

DC 2338, Office hour M3-4pm
THIS TIME

- **NP-completeness**
  - Subset Sum (finishing)
- NP-hardness
- Undecidable problems
- Back to NP-completeness
  - 0-1 Knapsack (in-class exercise)
  - TSP-Decision (**at-home exercise**)
FINISHING UP: REDUCING VERTEX-COVER TO SUBSET-SUM
(Proving Vertex-Cover $\leq_P$ Subset-Sum)
Let $I = (G, k)$, where $G = (V, E)$, $|V| = n$, $|E| = m$ and $1 \leq k \leq n$. Suppose $V = \{v_1, \ldots, v_n\}$ and $E = \{e_0, \ldots, e_{m-1}\}$. For $1 \leq i \leq n$, $0 \leq j \leq m - 1$, let $C = (c_{ij})$, where

$$c_{ij} = \begin{cases} 1 & \text{if } e_j \text{ is incident with } v_i \\ 0 & \text{otherwise.} \end{cases}$$

Define $n + m$ sizes and a target sum $W$ as follows:

$$a_i = 10^m + \sum_{j=0}^{m-1} c_{ij}10^j \quad (1 \leq i \leq n)$$

$$b_j = 10^j \quad (0 \leq j \leq m - 1)$$

$$W = k \cdot 10^m + \sum_{j=0}^{m-1} 2 \cdot 10^j$$

Then define $f(I) = (a_1, \ldots, a_n, b_0, \ldots, b_{m-1}, W)$.

This target weight asks for $k$ nodes and for all edges to be included twice.

Sort of like an adjacency matrix, but instead of storing which node-pairs are adjacent, store which edges are incident to each node.

Give a unique size of $10^j$ to each edge $e_j$.

Give a unique size to each node equal to $(10^m + \text{sizes of all edges incident to the node})$. 

Give a unique size to each edge.
Is there a 2-VC? Use subset sum to search for $W = 222222$

Looking for 2 nodes

All 5 edges counted twice

Node $v_2$ is not incident to edge $e_0$

Node $v_2$ is incident to edge $e_1$

Node $v_5$

Node $v_1$

Node $v_2$

Edge $e_0$

Edge $e_1$

Edge $e_4$

Sum of edge sizes incident to $v_1$, plus $10^m$

Note: no “carrying” can occur even if we sum everything

First digit of $W$ accurately captures # of nodes in the sum

Other digits are in [0,3]. An edge is definitely covered by a node if its digit is 2.

$W = 222222 = a_2 + a_3 + b_0 + b_1 + b_3 + b_4$

Subset sum looks for a subset of $\{a_1, a_2, a_3, a_4, a_5, b_0, b_1, b_2, b_3, b_4\}$ that sums to $W$

It finds $W = a_2 + a_3 + b_0 + b_1 + b_3 + b_4$

$a_2 + a_3 = 211211$

Edge $e_2$ counted twice, other edges once. Sum uses $b_0, b_1, b_3, b_4$ to get all to be counted twice.
Correctness of the Transformation

**Case 1:** Suppose $I$ is a yes-instance of Vertex-Cover. There is a vertex cover $V' \subseteq V$ such that $|V'| = k$. For $i = 1, 2$, let $E^i$ denote the edges having exactly $i$ vertices in $V'$. Then $E = E^1 \cup E^2$ because $V'$ is a vertex cover.

Let

$$A' = \{a_i : v_i \in V'\} \quad \text{and} \quad B' = \{b_j : e_j \in E^1\}.$$

The sum of the sizes in $A'$ is

$$k \cdot 10^m + \sum_{\{j : e_j \in E^1\}} 10^j + \sum_{\{j : e_j \in E^2\}} 2 \cdot 10^j.$$

The sum of the sizes in $B'$ is

$$\sum_{\{j : e_j \in E^1\}} 10^j.$$

Therefore the sum of all the chosen sizes is

$$k \cdot 10^m + \sum_{\{j : e_j \in E\}} 2 \cdot 10^j = k \cdot 10^m + \sum_{j=1}^{m} 2 \cdot 10^j = W.$$
Case 2: Suppose $f(I)$ is a yes-instance of Subset Sum.

- We show $I$ is a yes-instance of Vertex-Cover
- Since $f(I)$ is a yes-instance, there exists $A' \cup B'$ that sums to $W$
  - where $A'$ contains node sizes and $B'$ contains edge sizes
- Define $V' = \{v_i : a_i \in A'\}$. We claim $V'$ is a vertex cover of size $k$.
  - We must have $|V'| = k$ for the coefficient of $10^m$ to be $k$ (no carrying)
  - The coefficient of every other term $10^j$ ($j \leq m - 1$) must be 2
  - Suppose (for contra.) $V'$ does not cover some edge $e_j = (u, v)$
  - Then the coefficient of $10^j$ is zero for every $a_i \in A'$
  - But the coefficient of $10^j$ is 2, so a subset of $B'$ must sum to $2 \times 10^j$
  - But this is impossible (so $e_j$ is covered, so all edges are covered)
**Vertex Cover \( \leq_P \) Subset Sum**

Suppose \( I = (G, k) \), where \( G = (V, E) \), \(|V| = n\), \(|E| = m\) and \( 1 \leq k \leq n \).

Suppose \( V = \{v_1, \ldots, v_n\} \) and \( E = \{e_0, \ldots, e_{m-1}\} \). For \( 1 \leq i \leq n \), \( 0 \leq j \leq m - 1 \), let \( C = (c_{ij}) \), where

\[
c_{ij} = \begin{cases} 
1 & \text{if } e_j \text{ is incident with } v_i \\
0 & \text{otherwise.}
\end{cases}
\]

Define \( n + m \) sizes and a target sum \( W \) as follows:

\[
a_i = 10^m + \sum_{j=0}^{m-1} c_{ij} \cdot 10^j \quad (1 \leq i \leq n)
\]

\[
b_j = 10^j \quad (0 \leq j \leq m - 1)
\]

\[
W = k \cdot 10^m + \sum_{j=0}^{m-1} 2 \cdot 10^j
\]

Then define \( f(I) = (a_1, \ldots, a_n, b_0, \ldots, b_{m-1}, W) \).

Complexity of the transformation: Easy! Included for your notes.

Assume adjacency matrix and unit cost model for simplicity

Compute \( C \) with trivial algorithm in \( O(nm) \) time

Compute \( a_i \) by visiting all incident edges. Trivial algorithm yields \( O(m) \) time for each \( a_i \), totaling \( O(n + m) \) over all \( i \)

Trivial to compute all \( b_j \) in \( O(m) \) time

Trivial to compute \( W \) in \( O(m) \) time

Total \( O(nm) \) time. This is polynomial in the input graph size!
Every problem in NP reduces to

Let's take a break from NP-completeness to make sure we have time to cover NP-hardness and Undecidability
NP-HARDNESS

Problems that are \textbf{at least as hard} as NP-complete

(A quick definition)
NP-hard Problems

A problem \( \Pi \) is **NP-hard** if there exists a problem \( \Pi' \in \text{NPC} \) such that \( \Pi' \leq^T_p \Pi \).

Every NP-complete problem is automatically NP-hard, but there exist NP-hard problems that are not NP-complete.

Typical examples of NP-hard problems are optimization problems corresponding to NP-complete decision problems.

For example, \( \text{TSP-Decision} \leq^T_p \text{TSP-Optimization} \in \text{NPC} \), so \( \text{TSP-Optimization} \) is NP-hard.

**TSP-Optimal Value** is also NP-hard (and not in NP).

This version returns the **total weight** of an optimal Hamiltonian cycle.

Reduction from lecture 19

Returns an optimal Hamiltonian cycle
COMPLEXITY CLASS \textbf{EXPTIME}

A very brief overview

(Non-core material)
**EXPTIME** is the set of all decision problems that can be solved in exponential time. I.e., in time $O(2^{p(n)})$ where $p(n)$ is polynomial in the input size.

Observe that $NP \subseteq EXPTIME$

The idea is to generate all possible certificates of an appropriate length and check them for correctness using the given certificate verification algorithm. An example, for Hamiltonian Cycle, we could generate all $n!$ certificates and check each one in turn.

We do not know if there are problems in $NP$ that cannot be solved in polynomial time (because the $P = NP$ conjecture is not yet resolved). However, it is possible to prove that there exist problems in $EXPTIME \setminus P$. 
One such problem is the **Bounded Halting** problem. Here an instance $I = (A, x, t)$, where $A$ is a program, $x$ is an input to $A$, and $t$ is a positive integer (in binary). The question to be solved is if $A(x)$ halts after at most $t$ computation steps.

The **Bounded Halting** problem can be solved in time $O(t)$, but this is not a polynomial time algorithm because $\text{size}(I) = |A| + |x| + \log_2 t$.

Actually, it can be proven that **Bounded Halting** is EXPTIME-complete. This implies that it is in EXPTIME $\setminus$ P, since it is known that EXPTIME $\neq$ P.

$O(t)$ is exponential in $\log t$.

(And $\log t$ might be the largest term in the input size, in which case $O(t)$ would be **exponential in the input size**.)
UNDECIDABILITY
Problems that are impossible to solve
(Important material!)
UNDECIDABLE PROBLEMS

We say an algorithm $A$ “solves” a problem if, given any instance $I$, $A(I)$ has finite runtime and finds the correct answer.

A decision problem $\Pi$ is undecidable if there cannot exist an algorithm that solves $\Pi$.

If $\Pi$ is undecidable, then for every algorithm $A$, there exists at least one instance $I \in \mathcal{I}(\Pi)$ such that $A(I)$ does not find the correct answer (“yes” or “no”) in finite time.

Problem 7.19
Halting
Instance: A computer program $A$ and input $x$ for the program $A$.
Question: When program $A$ is executed with input $x$, will it halt in finite time?

The Halting problem is decidable IFF there exists an algorithm $\text{Halt}(I)$ that, given any instance $I = (A, x)$, will halt in finite time and correctly answer the question: “would a call to $A(x)$ halt in finite time?”
UNDECIDABILITY OF THE HALTING PROBLEM

Suppose that $Halt$ is a program that solves the Halting Problem.
The statement "$Halt$ solves the Halting problem" means that

$$Halt(A, x) = \begin{cases} 
\text{true} & \text{if } A(x) \text{ halts} \\
\text{false} & \text{if } A(x) \text{ doesn't halt.}
\end{cases}$$

Note that $A$ (the "algorithm") and $I$ (the "input" to $A$) are both strings over some finite alphabet.
If $\texttt{Halt}$ says $A$ will run forever, then we terminate in finite time.

If $\texttt{Halt}$ says $A$ will terminate in finite time, then we run forever.

Contradiction! Our only assumption, that $\texttt{Halt}$ exists, must be false!

Therefore, the Halting problem is undecidable.
Another Undecidable Problem

Here is another example of an undecidable problem. The problem *Halt-All* takes a program $A$ as input and asks if $A$ halts on all inputs $x$.

We describe a Turing reduction $\text{Halting} \leq^T \text{Halt-All}$, which proves that *Halt-All* is undecidable.

Assume we have a program *HaltAllSolver*.

For a fixed program $A$ and input $x$, let $B_x()$ be the program that executes $A(x)$ (so $B_x$ has no input).

Here is the reduction:

Given $A$ and $x$ (an instance of *Halting*), construct the program $B_x$.

Run $\text{HaltAllSolver}(B_x)$,

We have

$$\text{HaltAllSolver}(B_x) = \text{true} \iff A(x) \text{ halts},$$

so we can solve the halting problem.

If we have *HaltAllSolver* then we have *Halt*, but this is impossible, so *HaltAllSolver* cannot exist, so the Halt-All problem is undecidable!
BACK TO NP-COMPLETENESS

Exercises to wrap things up
Every problem in NP reduces to

Let's reduce subset sum to some other problem...
REDUCING SUBSET SUM TO 0-1 KNAPSACK

(Proving Subset-Sum $\leq_p$ 0-1 Knapsack)
Can I obtain profit $T$ (or better) by taking (whole) items with total weight $\leq M$?
This exercise: Show how to transform Subset-Sum input into 0-1 Knapsack input (in polynomial time).

Overarching goal: show Subset-Sum $\leq_P$ 0-1 Knapsack

Such that: if $I$ is a yes-instance of subset sum, then $f(I)$ is a yes-instance of 0-1 knapsack

And also: if $f(I)$ is a yes-instance of 0-1 knapsack, then $I$ is a yes-instance of subset sum
**Subset Sum \( \leq_P 0\text{-}1 \text{ Knapsack} \)**

Let \( I \) be an instance of \textbf{Subset Sum} consisting of sizes \([s_1, \ldots, s_n]\) and target sum \( T \).

Define

\[
\begin{align*}
p_i &= s_i, \quad 1 \leq i \leq n \\
w_i &= s_i, \quad 1 \leq i \leq n \\
M &= T
\end{align*}
\]

Then define \( f(I) \) to be the instance of \textbf{0-1 Knapsack} consisting of profits \([p_1, \ldots, p_n]\), weights \([w_1, \ldots, w_n]\), capacity \( M \) and target profit \( T \).

Exercise: Prove the correctness of this transformation.
Every problem in NP reduces to

Summary of Polynomial Transformations

CNF-SAT
↓ reduces to
3-CNF-SAT
↓
Clique
↓
Vertex Cover

Subset Sum
↓
0-1 Knapsack
Every problem in NP reduces to

CNF-SAT reduces to
3-CNF-SAT

Clique

Vertex Cover

Hamiltonian Cycle

Can reduce Hamiltonian Cycle easily to another problem we’ve seen...

This reduction is harder to prove... Doug posted an optional note showing how.
HOME EXERCISE:
REDUCE HAMILTONIAN CYCLE TO TSP-DECISION
**Problem 7.2**

**Hamiltonian Cycle**

**Instance:** An undirected graph $G = (V, E)$.

**Question:** Does $G$ contain a hamiltonian cycle?

A **hamiltonian cycle** is a cycle that passes through every vertex in $V$ exactly once.

**Problem 7.7**

**TSP-Decision**

**Instance:** A graph $G$, edge weights $w : E \rightarrow \mathbb{Z}^+$, and a target $T$.

**Question:** Does there exist a hamiltonian cycle $H$ in $G$ with $w(H) \leq T$?

**Overarching goal:** Show Hamiltonian Cycle $\leq_p$ TSP-Decision (Equivalently: exists a poly transformation from HC to TSP-DEC)

Such that: if $I$ is a yes-instance of Hamiltonian Cycle, then $f(I)$ is a yes-instance of TSP-Decision

**This exercise:** Show how to transform Hamiltonian Cycle input into TSP-Decision input (in poly time).

And also: if $f(I)$ is a yes-instance of TSP-Decision, then $I$ is a yes-instance of Hamiltonian Cycle.
Solution is hidden behind this box
Every problem in NP reduces to 3-CNF-SAT reduces to Clique reduces to Vertex Cover

Subset Sum \quad 0-1 Knapsack \quad Hamiltonian Cycle \quad TSP-Decision

All of these problems are NP-complete!
### SUMMARY OF COMPLEXITY CLASSES

- **P** (Poly-time)  
  E.g., (decision problem variants of:) BFS, Dijkstra’s, some DP algorithms

- **Decision** problems that can be solved by algorithms with runtime $\text{poly}(\text{input size})$

- **NP** (Non-deterministic poly-time)  
  All of P, and e.g., vertex cover, clique, SAT, subset sum

- **Decision** problems for which *certificates* can be *verified* in polynomial time

- Equivalently: decision problems that can be solved in poly-time if you have access to a non-deterministic oracle that returns a yes-certificate if one exists

- **NPC** (NP-complete)  
  E.g., vertex cover, clique, SAT, subset sum, TSP-decision

- **Decision** problems $\Pi \in \text{NP}$ s.t. every $\Pi' \in \text{NP}$ can be reduced to $\Pi$ in poly-time

- **NP-hard** (at least as hard as NPC)  
  All of NPC, and e.g., TSP-optimization, TSP-optimal value

- **Decision** problems $\Pi$ s.t. every $\Pi' \in \text{NP}$ can be reduced to $\Pi$ in poly-time

- **EXPTIME** (Exponential-time)  
  All of NP, and e.g., TSP-optimization, **Bounded** halting

- **Decision** problems that can be solved by algorithms with runtime $O(2^{\text{poly}(\text{input size})})$

- **Undecidable**  
  (Unbounded) halting

- **Decision** problems that *cannot* be solved (w/finite runtime for every input)