CS 341: ALGORITHMS

Lecture 21: intractability III – complexity class NP, poly transformations

Readings: see website

Trevor Brown

https://student.cs.uwaterloo.ca/~cs341

trevor.brown@uwaterloo.ca

THIS TIME

- Complexity class **NP**
  - Oracles, certificates, polytime verification algorithms
  - Two problems in NP
    - Subset sum
    - Hamiltonian Cycle
- Relationship between P and NP
- Polynomial **transformations**

COMPLEXITY CLASS **NP**

NP: Non-deterministic polynomial time

EXAMPLE: SUBSET-SUM PROBLEM

- Suppose we are given some integers, -7, -3, -2, 5, 8
- Does **some** subset of these **sum to zero?**
  - In this case, yes: (-3) + (-2) + 5 = 0

Finding such a subset can be extremely difficult

Suppose I give you a **certificate** consisting of an array of numbers, and **claim** it represents such a subset

Of course, I might lie and give you a subset that does **not sum to zero**…

If I'm telling the truth, then we call this a **yes-certificate.** It is is essentially a **proof** that "yes" is the correct output.

I could even give you numbers that are **not in the input**…

Can you use a yes-certificate to solve the problem efficiently?

Can you determine whether I am lying in polynomial time?

SUBSET-SUM VIA NON-DETERMINISTIC ORACLE

- Suppose there is a **non-deterministic oracle**, which returns **a subset that sums to 0 <u>if one exists</u>** and otherwise can return **anything** (even garbage)
- We call the oracle's output a **certificate**
- Given a **certificate**, can you **verify in polytime** whether it describes a solution to the problem?

Otherwise, either C is not a subset of the input (return false), or C sums to a non-zero value (return false)

If there **exists** a subset that sums to 0, then **C** is one such subset, and we return **true**

```
SubsetSumWithOracle(I)
    C = Oracle(I)
    return verify(I, C)

verify(I, C)
    if C not subset of I then return false
    return (sum(C) == 0)
```

Given such an oracle, this algorithm would **solve** subset-sum

"**Non-deterministic**" is the **N** in **NP**, and it is so named because of oracles

Here "**non-deterministic**" just means the oracle is magically guaranteed to return a yes-certificate if one exists

SUBSET-SUM VIA NON-DETERMINISTIC ORACLE

- Suppose there is a **non-deterministic oracle**, which returns **a subset that sums to 0 if one exists** and otherwise can return **anything** (even garbage)
- We call the oracle's output a **certificate**
- Given a **certificate**, can you **verify in polytime** whether it describes a solution to the problem?

```
SubsetSumWithOracle(I)
    C = Oracle(I)
    return verify(I, C)

verify(I, C)
    if C not subset of I then return false
    return (sum(C) == 0)
```

Given a certificate from the oracle, would **verify** solve the problem in **poly-time?**

Test whether C is a subset of I

For loop with $|C||I|$ time…

Test whether C sums to 0

For loop with $|C|$ time…

Input to **verify** is (I, C).
Runtime is $O(|C||I|)$, which is in $O(Size(I)^2) = O((|C| + |I|)^2)$

DUMB SUBSET-SUM ALGORITHM:
PRETEND YOU'RE AN ORACLE AND MAKE CERTS.

```
SubsetSum(X[1..n])
    for every possible subset S of X
        if sumsToZero(S) then return true
    return false
```

Generate **every subset certificate S**

Verify certificate S (valid + sums to zero)

If any certificate S sums to zero, it is a **yes-certificate** (a proof that the answer to the decision problem is "true"), and we return true

A certificates that does **not** sum to zero doesn't really prove anything (would need to know that **all** certificates sum to non-zero)

Generating these certificates is expensive; exponential time!

But **verifying one** certificate is fast; runtime is $poly(|S|)$

If there was such a thing as a **no-certificate**, what would it look like?
How long would it take to verify it?

Certificates

Certificate: Informally, a certificate for a yes-instance I is some "extra information" C which makes it easy to **verify** that I is a yes-instance.

Certificate Verification Algorithm: Suppose that *Ver* is an algorithm that verifies certificates for yes-instances. Then *Ver*(I, C) outputs "yes" if I is a yes-instance and C is a valid certificate for I. If *Ver*(I, C) outputs "no", then either I is a no-instance, or I is a yes-instance and C is an invalid certificate.

Polynomial-time Certificate Verification Algorithm: A certificate verification algorithm *Ver* is a polynomial-time certificate verification algorithm if the complexity of *Ver* is $O(n^k)$, where k is a positive integer and $n =$ *Size*(I).

Always keep the following in mind: finding a certificate can be much more difficult than verifying a given certificate.

As a rough analogy, finding a proof for a theorem can be much harder than verifying the correctness of someone else's proof.

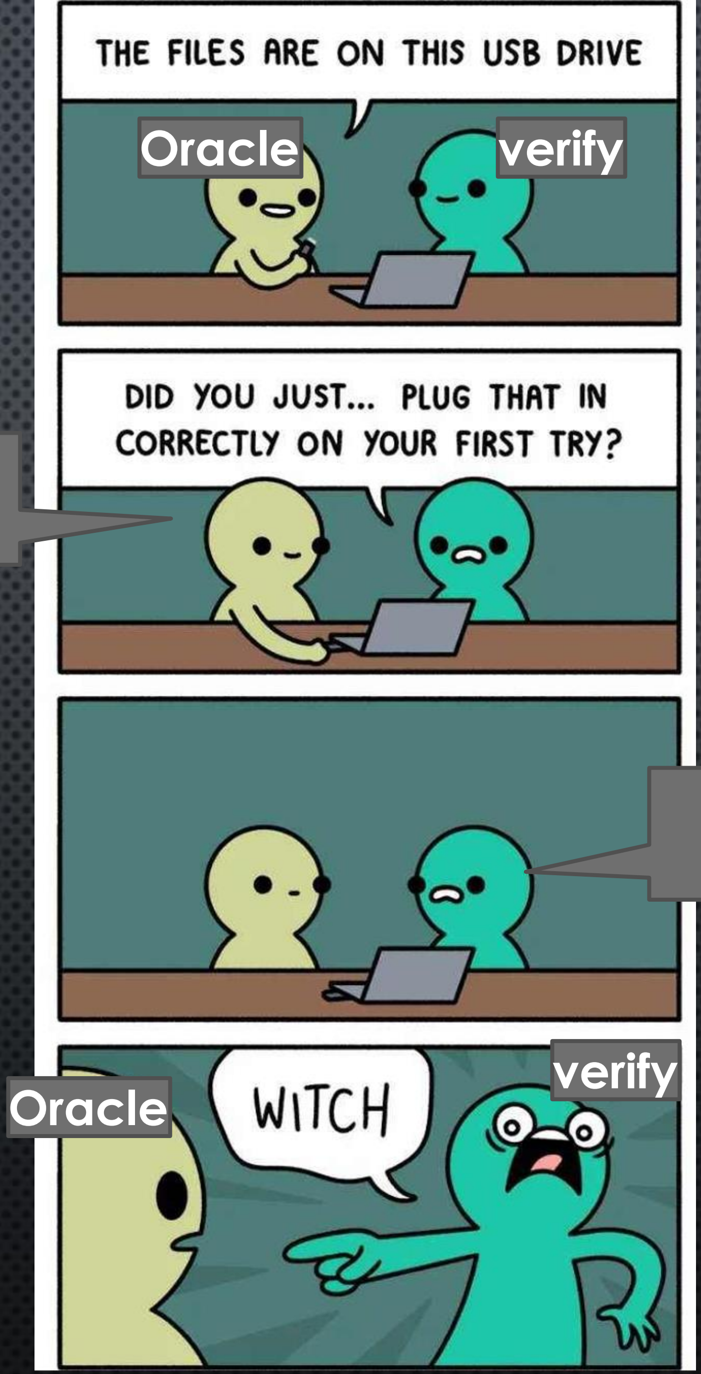
GENERALIZING BEYOND SUBSET-SUM

- You can solve **any decision problem** in non-deterministic poly-time, given:
  1. a poly-time non-deterministic **oracle**, and
  2. a poly-time $\boldsymbol{verify}$ algorithm
- Such that:
  - If I is a **yes-**instance, then the oracle returns a **yes-**certificate C (i.e., a "proof" the answer is "yes") and $verify(I, C)$ returns $true$
  - If I is a **no-**instance, then $verify(I, C)$ returns $false$ **for all** $\boldsymbol{C}$ (i.e., it must be impossible to fool $verify$ into returning true)
- The algorithm:

```
SolveAnyProblemWithOracle(I)
    C = Oracle(I)
    return verify(I, C)
```

Our definition of NP will <u>not</u> explicitly involve non-deterministic oracles. But it *is* based on **certificate verification**, which makes more sense if you think of such oracles…

Could you "fool" the subset-sum verify function?



Guesses solution in O(1) time

Verifies solution in poly-time

As we are about to see: **existence** of a poly-time verifier for a problem means **problem is in NP**

DEFINING NP

Intuition: For a yes-instance, there must exist **some certificate** that verify would accept (and, if one exists, the oracle would find it, solving the problem). For a no-instance, verify must always reject.

- A decision problem Π is **solved** by a poly-time $verify$ alg. iff:
  - for every **yes**-instance I, **there exists** a certificate C such that $verify(I, C)$ returns true, and
  - for every **no**-instance I, $verify(I, C)$ returns $false$ **for every** C

Crucial definition!

- **The complexity class NP** denotes the set of all decision problems that **can be solved** by poly-time $verify$ algorithms
  - **No oracle needed!** Note it is **not** necessary for an oracle to actually exist for a problem to be in NP.
    We can simply **assume** certificates come from an oracle, and show a poly-time $verify$ algorithm exists.

MECHANICS OF SHOWING A PROBLEM IS **IN NP**

- How to show $\Pi \in NP$

1. Define a yes-certificate
2. Design a poly-time $verify(I, C)$ algorithm
3. Correctness proof
   - **Case 1:** Let I be any yes-instance; Find C such that $verify(I, C) = true$
   - **Case 2:** Let I be any no-instance, and C be any certificate; Prove $verify(I, C) = false$

Subset-sum as an example:
A yes-certificate is a list of indices in the input array where the elements should sum to 0

How to verify a certificate C **is a subset of input I with sum zero**?

$\forall c \in C$, add $I[c]$ to sum, and return true iff sum=0

$O(|C|)$ **time**

This is certainly polytime...

Case 1: Let I be a yes-instance.
There *is* a subset in I that sums to 0.
For any such subset C, verify(I,C) will return true.

Case 2: Let I be a no-instance & C be any certificate.
No subset of I sums to 0.
So,$\Sigma_{c \in C} I[c] \neq 0$ and verify returns false.

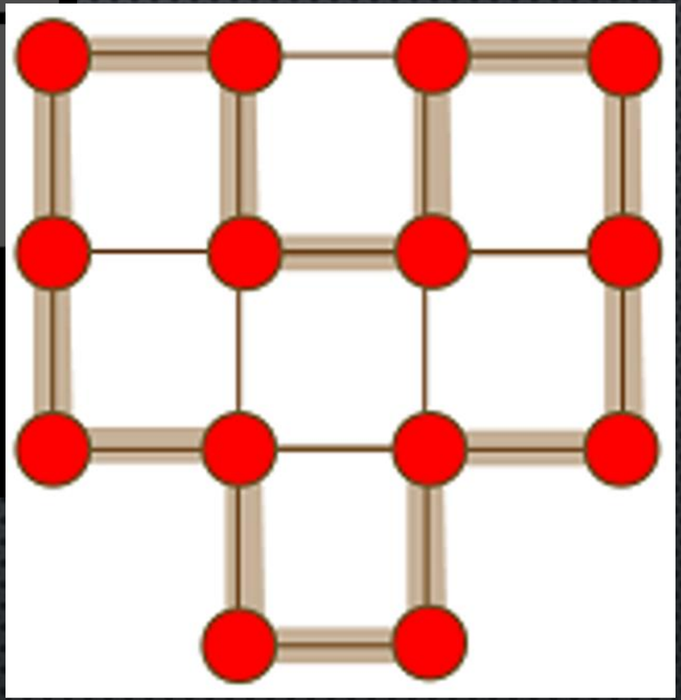
So, subset-sum $\in NP$

ANOTHER EXAMPLE: HAMILTONIAN CYCLE PROBLEM

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.



Let's show that this problem is in NP!

Have to find a poly-time $verify$ algorithm…

Defining a yes-certificate: **array** of nodes representing a Hamiltonian cycle

How to verify that a **given** array of nodes represents a **cycle**?

How about a **Hamiltonian** cycle?

EXAMPLE: SHOWING "HAMILTONIAN CYCLE" IS **IN NP**

```
HamiltonianCycleVerify(G=(V,n,E,m), X)
    if size(X) is not n then return false
    used[1..n] = array containing all false
    for i = 1..n
        if used[X[i]] then return false
        used[X[i]] = true
    for i = 1..(n-1)
        if no edge X[i] to X[i+1] then return false
    if no edge X[n] to X[1] then return false
    return true
```

This is a $\boldsymbol{verify}$ algorithm that we imagine being called on the certificate X produced by $oracle(G)$

A **<u>certificate X</u>** consists of an **array** of node names (1…n), which **might** represent a Hamiltonian cycle

If G is a **yes-instance** of the problem, then must show there **exists** **<u>some</u> possible certificate X** for which this procedure returns will true

What would such a certificate look like?

Yes-instance implies there **is** a Hamiltonian cycle.
Suppose X is a sequence of n consecutive nodes on that cycle.
Then we return true!

EXAMPLE: SHOWING "HAMILTONIAN CYCLE" IS **IN NP**

```
HamiltonianCycleVerify(G=(V,n,E,m), X)
    if size(X) is not n then return false
    used[1..n] = array containing all false
    for i = 1..n
        if used[X[i]] then return false
        used[X[i]] = true
    for i = 1..(n-1)
        if no edge X[i] to X[i+1] then return false
    if no edge X[n] to X[1] then return false
    return true
```

This is a $\boldsymbol{verify}$ algorithm that we imagine being called on the certificate X produced by $oracle(G)$

A **certificate X** consists of an **array** of node names (1…n), which **might** represent a Hamiltonian cycle

If G is a **no-instance** of the problem, then "**every** possible certificate should cause verify to return false"

Easier to prove the contrapositive:
"if verify returns true, then G is a yes-instance."

If we return true, then the graph contains a cycle with n distinct nodes… So G is a yes-instance

So, Hamiltonian Cycle **is in NP**

HOW ARE **P** AND **NP** RELATED?

- $P \subseteq NP$
  - Consider a problem $\Pi \in P$
  - We show there exists a poly-time $verify(I, C)$ **such that:**
    - For every **yes**-instance I of Π, $verify(I, C) = true$ for **some** C
    - For every **no**-instance I of Π, $verify(I, C) = false$ for **all** C
  - By definition, there is a poly-time algorithm A to solve Π
    - Implement $verify(I, C)$ by simply running $A(I)$ **[ignoring C]**
    - Regardless of what C is, $verify(I, C)$ satisfies the above
- How about $NP \subseteq P$?

Million dollar question. We think not.

POLYNOMIAL **TRANSFORMATIONS**

A subclass of poly-time reductions

commonly used for **NP-completeness** and **impossibility** results

POLYNOMIAL TRANSFORMATIONS

For a decision problem Π, let $\mathcal{I}(\Pi)$ denote the set of all instances of Π. Let $\mathcal{I}_{\mathbf{yes}}(\Pi)$ and $\mathcal{I}_{\mathbf{no}}(\Pi)$ denote the set of all yes-instances and no-instances (respectively) of Π.

Suppose that Π_1 and Π_2 are decision problems. We say that there is a **polynomial transformation** from Π_1 to Π_2 (denoted $\Pi_1 \leq_P \Pi_2$) if there exists a function $f : \mathcal{I}(\Pi_1) \to \mathcal{I}(\Pi_2)$ such that the following properties are satisfied:

$f(I)$ is computable in polynomial time (as a function of *size*(I), where $I \in \mathcal{I}(\Pi_1)$)

if $I \in \mathcal{I}_{\mathbf{yes}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\mathbf{yes}}(\Pi_2)$

if $I \in \mathcal{I}_{\mathbf{no}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\mathbf{no}}(\Pi_2)$

[Mechanics] to give a polynomial transformation, you must:
1. **specify** $f(I)$,
2. **show** it runs in poly-time, and
3. **show** I is a yes-instance of Π_1 **IFF** $f(I)$ is a yes-instance of Π_2.

POLYNOMIAL TRANSFORMATIONS (CONT.)

A polynomial transformation can be thought of as a (simple) special case of a polynomial-time Turing reduction, i.e., if $\Pi_1 \leq_P \Pi_2$, then $\Pi_1 \leq_P^T \Pi_2$.

Given a polynomial transformation f from Π_1 to Π_2, the corresponding Turing reduction is as follows:

Given $I \in \mathcal{I}(\Pi_1)$, construct $f(I) \in \mathcal{I}(\Pi_2)$.
Given an oracle for Π_2, say A, run $A(f(I))$.

We transform the instance, and then make a single call to the oracle.

Very important point: We do not know whether I is a yes-instance or a no-instance of Π_1 when we transform it to an instance $f(I)$ of Π_2.

To prove the implication "if $I \in \mathcal{I}_{\mathbf{no}}(\Pi_1)$, then $f(I) \in \mathcal{I}_{\mathbf{no}}(\Pi_2)$", we usually prove the contrapositive statement "if $f(I) \in \mathcal{I}_{\mathbf{yes}}(\Pi_2)$, then $I \in \mathcal{I}_{\mathbf{yes}}(\Pi_1)$.

Also known as **Karp reductions** and **many-one reductions**

We saw one instance where a contrapositive was easier to prove when we discussed Hamiltonian cycles

The contrapositive can help when it is hard to precisely characterize certificates for no-instances (or when such certificates don't prove much)

SUMMARIZING
THE MORE CONVENIENT DEFINITION

- Let Π_1 and Π_2 be decision problems
- **$\Pi_1 \le_P \Pi_2$ iff** there exists $f : \mathcal{I}(\Pi_1) \to \mathcal{I}(\Pi_2)$ such that:
  - $f(I)$ is computable in poly-time, for all $I \in \mathcal{I}(\Pi_1)$
  - If $I \in \mathcal{I}_{yes}(\Pi_1)$ then $f(I) \in \mathcal{I}_{yes}(\Pi_2)$
  - **If $f(I) \in \mathcal{I}_{yes}(\Pi_2)$ then $I \in \mathcal{I}_{yes}(\Pi_1)$**

Note: this is the same as saying
$(\boldsymbol{I} \in \mathcal{I}_{yes}(\boldsymbol{\Pi_1})) \Leftrightarrow (\boldsymbol{f}(\boldsymbol{I}) \in \mathcal{I}_{yes}(\boldsymbol{\Pi_2}))$

This is the contrapositive. Was previously (2 slides ago):
If $I \in \mathcal{I}_{no}(\Pi_1)$ then $f(I) \in \mathcal{I}_{no}(\Pi_2)$

This property justifies correctness for the following generic **poly-time Karp reduction:**

```
P1toP2KarpReduction(I)
  fI = f(I)
  return OracleForP2(fI)
```

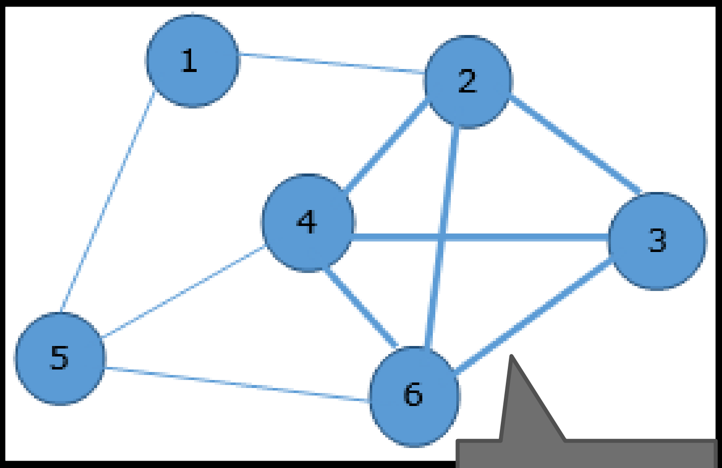
EXAMPLE POLYNOMIAL TRANSFORMATION

Problem 7.8

Clique

Instance: *An undirected graph* $G = (V, E)$ *and an integer* k, *where* $1 \leq k \leq |V|$.

Question: *Does* G *contain a clique of size* $\geq k$? *(A* **clique** *is a subset of vertices* $W \subseteq V$ *such that* $uv \in E$ *for all* $u, v \in W$, $u \neq v$.*)*

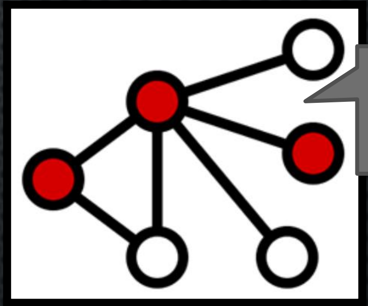


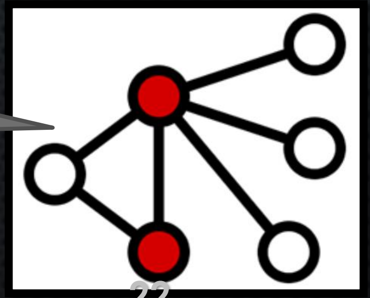
Problem 7.9

Vertex Cover

Instance: *An undirected graph* $G = (V, E)$ *and an integer* k, *where* $1 \leq k \leq |V|$.

Question: *Does* G *contain a vertex cover of size* $\leq k$? *(A* **vertex cover** *is a subset of vertices* $W \subseteq V$ *such that* $\{u, v\} \cap W \neq \emptyset$ *for all edges* $uv \in E$.*)*

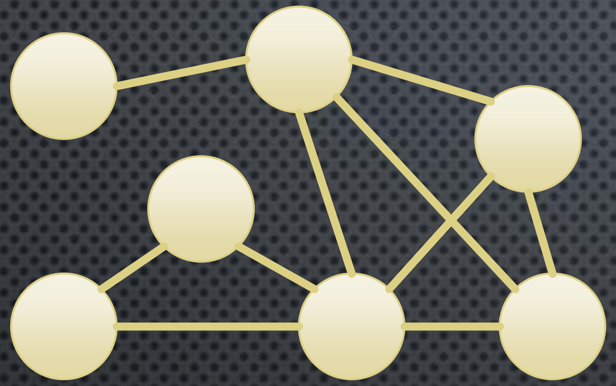




CLIQUE $\leq_P$ VERTEX-COVER

- Suppose $I = (G, k)$ is an instance of Clique where $G = (V, E), V = \{v_1, \dots, v_n\}$ and $1 \leq k \leq n$

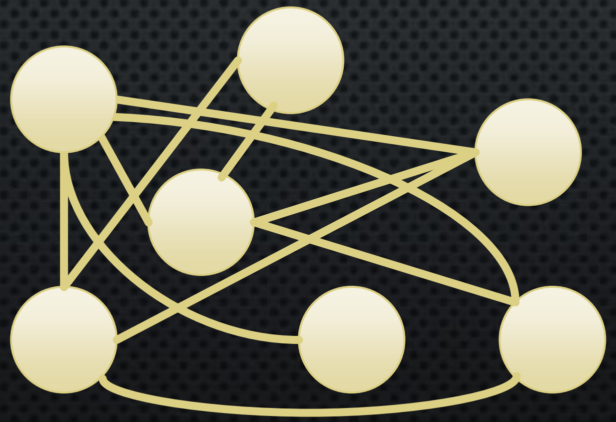
Want to solve $Clique(G, k)$



Claim: there is a k-clique in G ***iff*** there is an $(n-k)$ Vertex-Cover in $\overline{\boldsymbol{G}}$

- **Construct** instance $f(I) = (\overline{\boldsymbol{G}}, n-k)$ of Vertex-Cover, where $H = (V, \bar{E})$ and $v_i v_j \in \bar{E} \Leftrightarrow v_i v_j \notin E$

Idea: reduce to $VertexCover(\overline{\boldsymbol{G}}, n-k)$



Consider the **complement graph** $\overline{\boldsymbol{G}}$ of G

Every edge of G is a non-edge of $\overline{\boldsymbol{G}}$. Every non-edge of G is an edge of $\overline{\boldsymbol{G}}$.

Given an adjacency matrix for G, get $\overline{\boldsymbol{G}}$ by **flipping 0's and 1's.**

PROVING THIS IS A **POLYNOMIAL TRANSFORMATION**

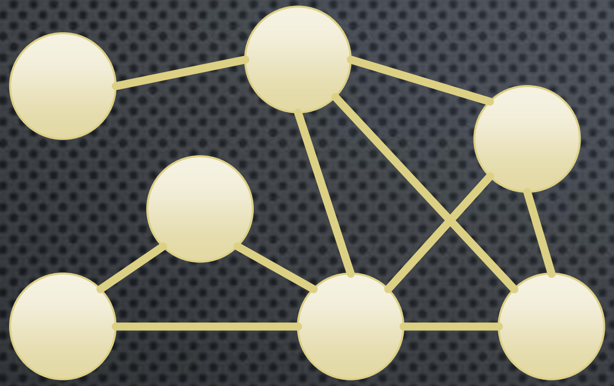
- We denote Clique by CL and Vertex-Cover by VC
- $CL \leq_P VC$ **iff** there exists $f : \mathcal{I}(CL) \rightarrow \mathcal{I}(VC)$ such that:
  - **$f(I)$ is computable in poly-time, for all $I \in \mathcal{I}(CL)$**
  - If $I \in \mathcal{I}_{yes}(CL)$ then $f(I) \in \mathcal{I}_{yes}(VC)$
  - If $f(I) \in \mathcal{I}_{yes}(VC)$ then $I \in \mathcal{I}_{yes}(CL)$

First let's show this

COMPLEXITY OF THE TRANSFORMATION

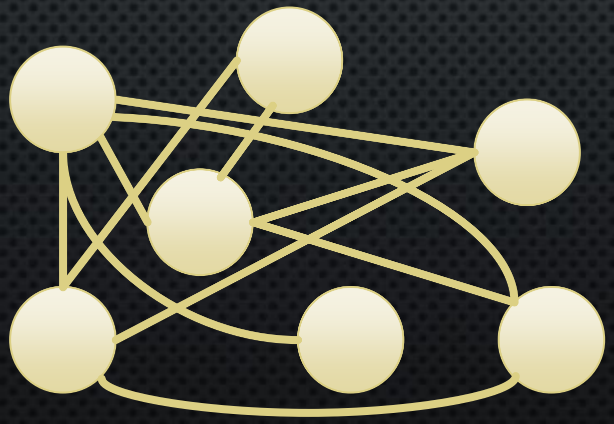
- Suppose $\boldsymbol{I = (G, k)}$ is an instance of Clique where $G = (V, E), V = \{v_1, \dots, v_n\}$ and $1 \le k \le n$

Want to solve $Clique(G, k)$



- **Construct** instance $\boldsymbol{f(I) = (\overline{G}, n - k)}$ of Vertex-Cover, where $\overline{\boldsymbol{G}} = (V, \overline{E})$ and $v_i v_j \in \overline{E} \Leftrightarrow v_i v_j \notin E$

Idea: reduce to $VertexCover(\overline{\boldsymbol{G}}, n - k)$



Assuming adjacency matrix, $Size(I) = \Theta(n^2 + \log_2 k)$

Time to compute $f(I)$?

Constructing $\overline{\boldsymbol{G}}$ takes $O(n^2)$ time, and computing $n - k$ takes $O(\log n)$ time.

So computing $f(I)$ takes $O(n^2)$ time, which is polynomial in $Size(I)$.

PROVING THIS IS A **POLYNOMIAL TRANSFORMATION**

- We denote Clique by CL and Vertex-Cover by VC
- $CL \leq_P VC$ **iff** there exists $f : \mathcal{I}(CL) \rightarrow \mathcal{I}(VC)$ such that:
  - $f(I)$ is computable in poly-time, for all $I \in \mathcal{I}(CL)$
  - **If $\boldsymbol{I \in \mathcal{I}_{yes}(CL)}$ then $\boldsymbol{f(I) \in \mathcal{I}_{yes}(VC)}$**
  - If $f(I) \in \mathcal{I}_{yes}(VC)$ then $I \in \mathcal{I}_{yes}(CL)$

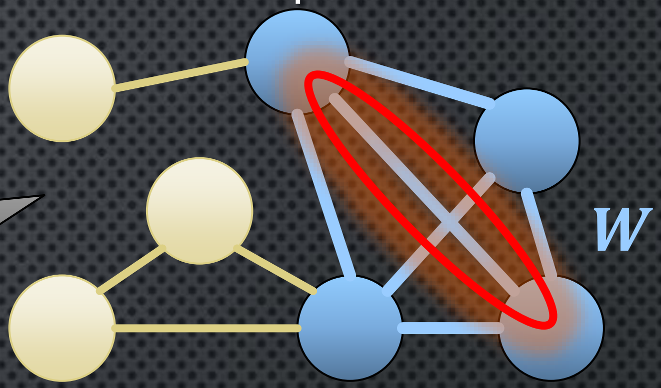
Now let's show this, i.e.,
if $\boldsymbol{G}$ contains a $\boldsymbol{k}$-clique then
$\boldsymbol{\overline{G}}$ contains an $\boldsymbol{(n-k)}$ vertex cover.

PROVING: $I \in \mathcal{I}_{yes}(CL) \Rightarrow f(I) \in \mathcal{I}_{yes}(VC)$

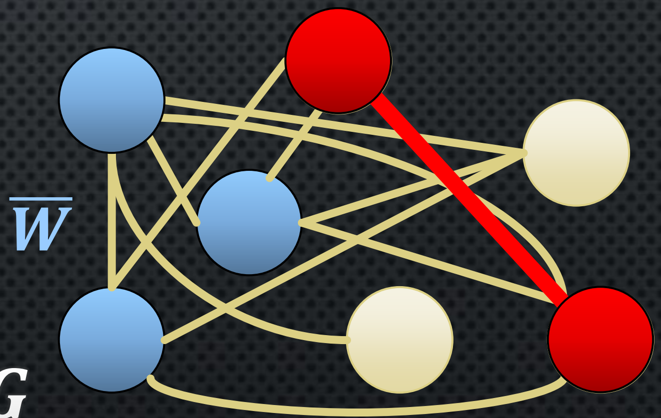
- Suppose $I = (G, k)$ is a **yes**-instance of Clique
- Then there is a set W of k vertices in a clique (with **all-to-all** edges)
- Define $\boldsymbol{\overline{W}} = V \setminus W$. Clearly $|\boldsymbol{\overline{W}}| = n - k$.
- We **claim** $\boldsymbol{\overline{W}}$ is a vertex cover of $\boldsymbol{\overline{G}}$
- Consider any edge $(u, v) \in \boldsymbol{\overline{G}}$
- If either u or v is in $\boldsymbol{\overline{W}}$, then we are done, so assume $u, v \notin \boldsymbol{\overline{W}}$ to obtain a contradiction
- Then $u, v \in W$, and W is a clique in G, so $(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{G}$
- But $(u, v) \in \boldsymbol{\overline{G}}$ implies $(u, v) \notin G$. Contradiction!

Example: $Clique(G, 4)$

Graph $\boldsymbol{G}$



Graph $\boldsymbol{\overline{G}}$



PROVING THIS IS A **POLYNOMIAL TRANSFORMATION**

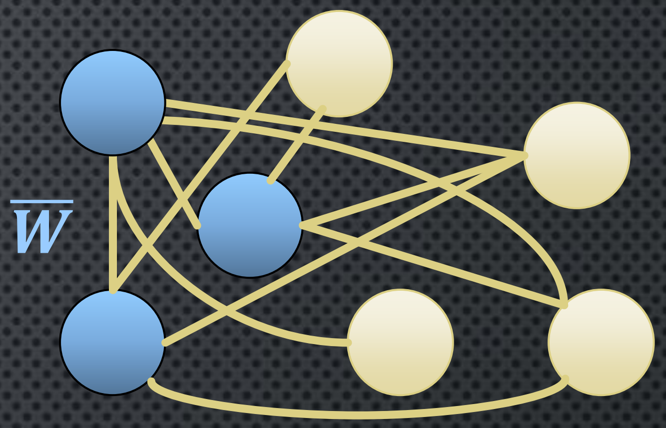
- We denote Clique by CL and Vertex-Cover by VC
- $CL \leq_P VC$ **iff** there exists $f : \mathcal{I}(CL) \rightarrow \mathcal{I}(VC)$ such that:
  - $f(I)$ is computable in poly-time, for all $I \in \mathcal{I}(CL)$
  - If $I \in \mathcal{I}_{yes}(CL)$ then $f(I) \in \mathcal{I}_{yes}(VC)$
  - **If $\boldsymbol{f(I) \in \mathcal{I}_{yes}(VC)}$ then $\boldsymbol{I \in \mathcal{I}_{yes}(CL)}$**

Now let's show this, i.e.,
if $\overline{G}$ contains an $(n-k)$ vertex cover, then G contains a k-clique

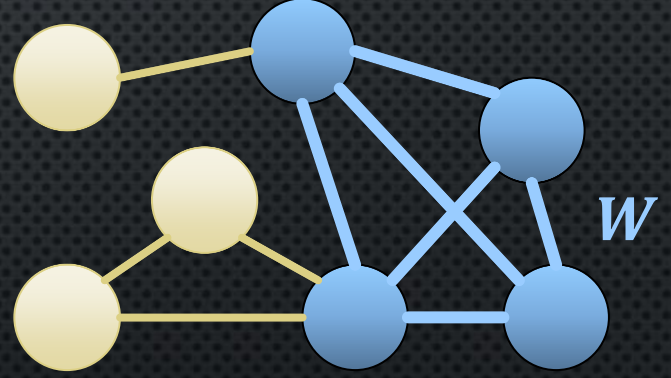
PROVING: $f(I) \in \mathcal{I}_{yes}(VC) \Rightarrow I \in \mathcal{I}_{yes}(CL)$

- Suppose $f(I) = (\overline{\boldsymbol{G}}, n - k)$ is a **yes**-instance of VC
- Then there is a set of $n - k$ vertices $\overline{\boldsymbol{W}}$ that is a vertex cover of $\overline{\boldsymbol{G}}$
- Define $W = V \setminus \overline{\boldsymbol{W}}$. Clearly $|W| = k$.
- We **claim** W is a clique in G
- Since $\overline{\boldsymbol{W}}$ is a vertex cover of $\overline{\boldsymbol{G}}$, **every edge** in $\overline{\boldsymbol{G}}$ has at least one endpoint in $\overline{\boldsymbol{W}}$
- Therefore, **no edge** in $\overline{\boldsymbol{G}}$ has two endpoints in W
- So, in $\boldsymbol{G}$, there are edges between all pairs of nodes in W. So, W is a clique in G.

Graph $\overline{\boldsymbol{G}}$



Graph $\boldsymbol{G}$



So, we have demonstrated a polynomial transformation from CLIQUE to VERTEX-COVER