CS 6212 DESIGN AND ANALYSIS OF ALGORITHMS

LECTURE: NP-COMPLETE THEORY – PART II

Instructor: Abdou Youssef

WHAT YOU LEARNED IN THE PREVIOUS LECTURE

- Taxonomy of computational problems
- Yes-no problems
- The NP class/family and its significance
- Definition of NP problems and NP algorithms
- Development of NP algorithms for a number of NP problems

OBJECTIVES OF THIS LECTURE

By the end of this lecture, you will be able to:

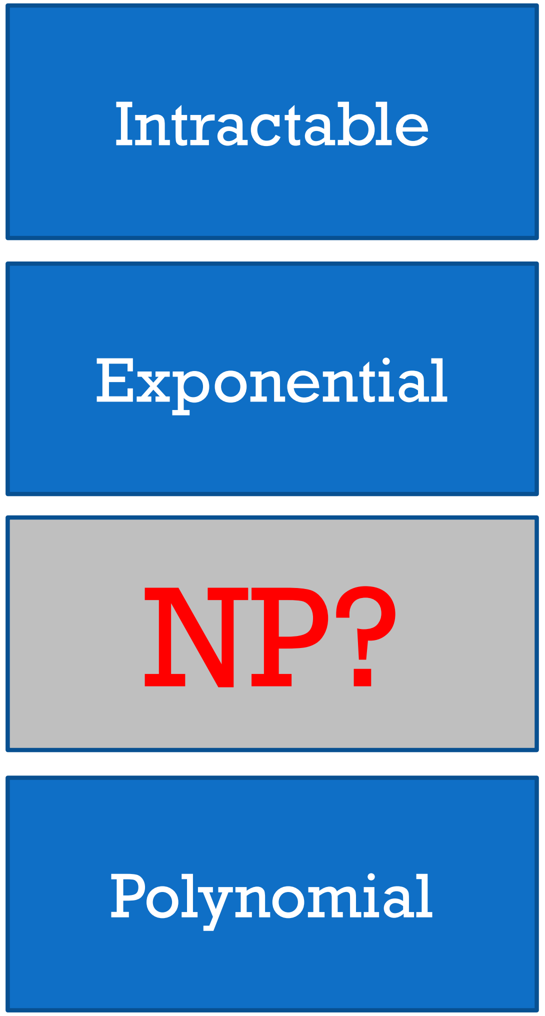
- Describe polynomial-time answer-preserving transforms b/w yes-no problems
- Generalize transforms to broader contexts, and apply them to lower-bound theory
- Explain reductions and their significance to NP-Completeness, both technically and with useful analogies
- Explain NP-completeness, both conceptually and in practical reduction-based terms
- Explicate and apply a reduction-based strategy for proving certain problems NP-complete
- Construct a transform from the satisfiability problem (SAT) to the k-clique problem, and prove its validity, to conclude that the CLIQUE problem is NP-complete
- Convey the meaning and wide-ranging implications of the most famous open question in CS: $P \overset{?}{=} NP$
- Express the significance of what it means for a problem to be NP-complete

OUTLINE

- A review of what NP is (NP algorithms and NP problems)
- Transforms for yes-no problems
- Generalizations of transforms and some applications
- Reductions (using transforms)
- Definition of NP-completeness, conceptually first, and using reductions second
- Practical strategy for proving new problems to be NP-complete
- Proving CLIQUE to be NP-complete, using a transform/reduction
- Closing thoughts

TAXONOMY OF COMPUTATIONAL PROBLEMS
-- FOUR BROAD CLASSES OF PROBLEMS --

- The world of computation can be subdivided into 3 classes:
  - Polynomial problems (P)
  - Exponential problems (E)
  - Intractable/undecidable (non-computable) problems (I)
- There is a very large and important class of problems that
  - we know how to solve exponentially,
  - we don't know how to solve polynomially, and
  - we don't know if they can be solved polynomially at all
- This class is a gray area between the P-class and the E-class. It is the class of NP problems (which could be = P, or a subclass of E)



FOCUS ON YES-NO PROBLEMS

- **Definition**: A *yes-no problem* consists of an instance (or input I) and a yes-no question Q.
- Yes-no problems are also called *decision problems*
- Much of NP-complete theory focuses on yes-no problems

TEMPLATE OF NP ALGORITHMS

```
begin
    // the guessing stage: guesses solution X[1:N] in O(N) time
    for i=1 to N do
        X[i] := choose(i);
    endfor

    // the verification stage
    Write code that does not use "choose" and that verifies in
    polynomial time if X[1:N] is a correct solution to the
    problem. If correct, return (yes); else, return no.
end
```

The guessing stage (uses "choose")

The verification stage (NO "choose")

DEFINITION OF NP

-- ALTERNATIVE DEFINITION: THE IMAGINARY “CHOOSE” --

- **Definition 3 of NP**: A yes-no problem is said to be NP if
    - there exists an NP algorithm for (i.e., a polynomial guess-verify algorithm), and
    - the solutions come from a finite set of possibilities.
- **Remark**: For the NP algorithm template to be polynomial, the solution size N must be polynomial in the input size n, and the verification stage must be polynomial in n.

LESSONS LEARNED SO FAR

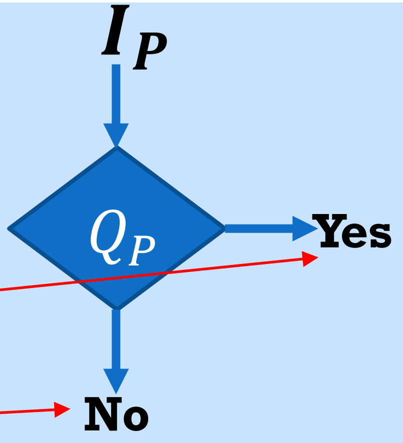
- There are some problems, called NP problems, that can be solved in exponential time, but we don't know if they are inherently exponential or can one day be solved in polynomial time
- Proving a yes-no problem to be NP is easy: design a guess-verify algorithm (where the guess stage uses "choose" but the verify stage does not), such that the verify stage takes poly time
- The verify-stage is similar to the Bound function in Backtracking

TRANSFORMS

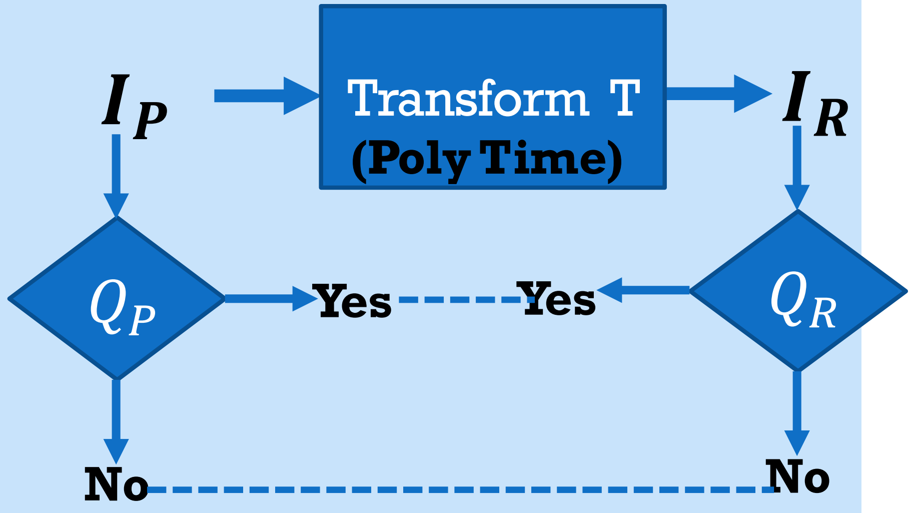
- **Notation:** If P stands for a yes-no problem, then
  - I_P: denotes an instance (i.e., an input) of P
  - Q_P: denotes the question of P
  - Answer(Q_P,I_P): the answer to the question Q_P given input I_P

    Answer(Q_P, I_P) is Yes

    or No



- Let P and R be two yes-no problems
- **Definition:** A ***transform*** (that transforms a problem P to a problem R) is an algorithm T such that:
  - T takes polynomial time
  - The input of T is I_P, and the output of T is I_R
  - Answer(Q_P,I_P)=Answer(Q_R,I_R), i.e., both Yes or both No

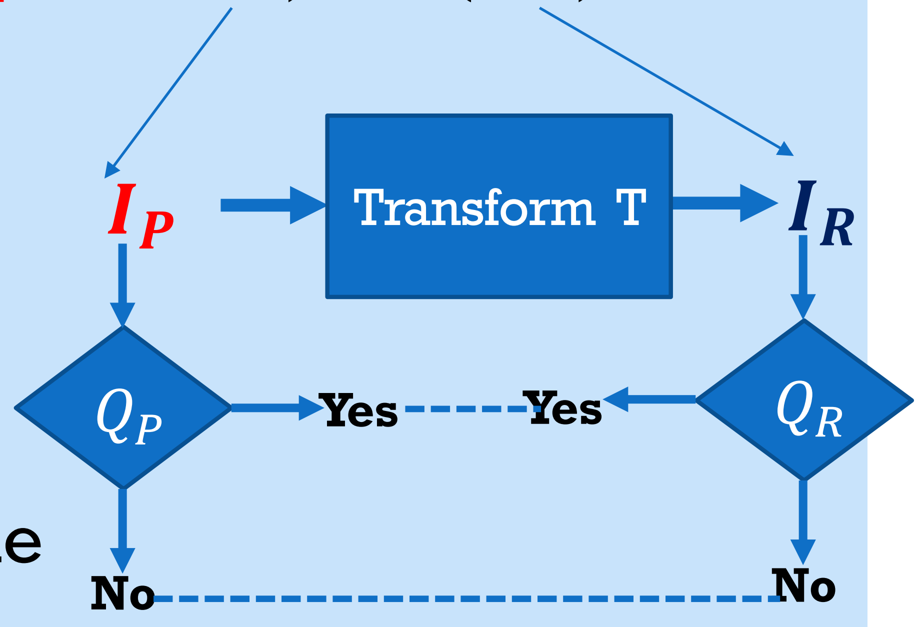


TRANSFORM ELABORATION

- For example, take
  - P to be the SAT problem, where I_P is F, and Q_P is "Is F satisfiable"?
  - R to be the CLIQUE problem, where I_P is (G, k), and Q_P is "Does G have a k-clique?"
- A transform T from P to R transforms a (Boolean expression F) to a (G, k)

  such that:
  - Answer(Q_P, I_P)=Answer(Q_R, I_R)
  - That is, both answers are Yes or both are No
  - i.e., if F is satisfiable, then G has a k-clique,

    if F is not satisfiable, then G has no k-clique



A BROADER VIEW OF TRANSFORMS

- We defined transforms in the context of NP-complete theory
- But we can broaden the definition of transforms, EX:
  - Transforms between problems that are not yes-no problems
  - Transforms that take O(n) time only (or any threshold time you wish)
- Such general transforms can be used to:
  - Import/adapt a solution of an old problem to a new problem
  - Establish some lower bounds on new problems, using lower bounds of old problems
  - Understand the structure/dynamics of a new problem using the structure/dynamics of a well-understood old problem

USING TRANSFORMS IN OTHER APPLICATIONS

- Prove that sorting a heap is $\Theta(n \log n)$
  - Hint: Use heap-building as a general transform, and take advantage of the fact that general sorting is $\Theta(n \log n)$
- Prove that building a BST from an arbitrary input array is $\Theta(n \log n)$
  - Hint: Use BST-building as a general transform, and take stock of the time it takes to sort a BST. Also, use the fact that general sorting is $\Theta(n \log n)$

LESSONS LEARNED SO FAR

- NP problems can be solved exponentially but we don’t know if they are inherently exponential or are polynomial
- To prove a yes-no problem to be NP, design a guess-verify algorithm where “verify” is deterministically polynomial
- The verify-stage is similar to Bound in Backtracking
- A transform polynomially maps (the input) of a yes-no problem to (the input of) another yes-no problem, preserving the answer (by “preserving the structure” of the first problem)
- Transforms can be generalized, and applied in other contexts

SWITCH BACK TO OUR RESTRICTED DEFINITION OF TRANSFORMS

- Yes, it is useful to know that transforms apply in many other contexts and many other applications
- But for the rest of the lecture, we will use transforms only in the restricted sense we defined initially: **poly-time** **answer-preserving** transforms <u>between</u> **yes-no problems**

REDUCTIONS

- **Definition**: We say that a problem P ***reduces*** to a problem R if there exists a (poly-time answer-preserving) transform from P to R.
- We denote reduction of P to R as P $\leq_P$ R
  - The subscript P in $\leq_P$ stands for "polynomial"
  - We'll see the rationale for that notation in a little bit
- **Observation** (Transitivity of reduction): If P, R and S are three yes-no problems, and if P $\leq_P$ R and R $\leq_P$ S, then P $\leq_P$ S
  - The reason is that by performing the first transform (from P to R) and then the second transform (from R to S), we get a transform from P to S.

WHAT REDUCTION IMPLIES

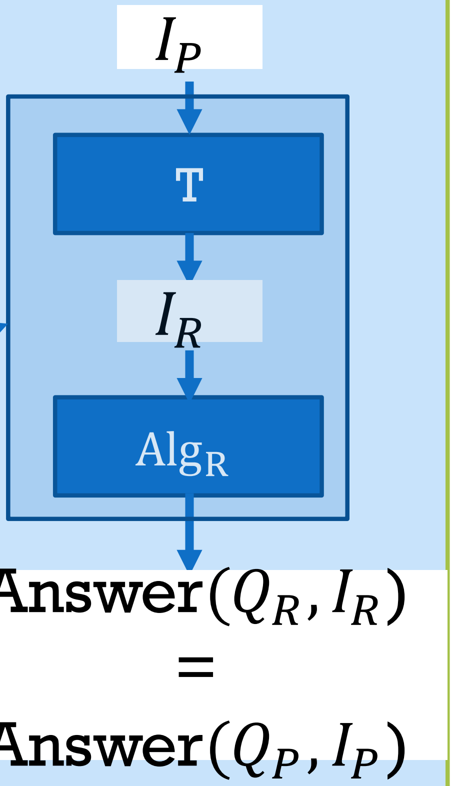
- **Theorem**: Let P and R be two problems. If P $\leq_P$ R (i.e., P reduces to R) and R is polynomial, then P is polynomial.
- **Proof**:
  - Let T be the poly-time transform that transforms P to R

    $$\text{Answer}(Q_P, I_P) = \text{Answer}(Q_R, I_R)$$

  - Let Alg_R be the polynomial-time algorithm for R

    $$\text{Alg}_\text{R}(I_R) = \text{Answer}(Q_R, I_R)$$

  - Since $\text{Answer}(Q_P, I_P) = \text{Answer}(Q_R, I_R)$, this algorithm solves problem P too, and takes polynomial time because polynomials compose (i.e., poly of poly is poly, and the sum of polys is a poly). Q.E.D.



MORE INTUITION INTO WHAT REDUCTION REALLY MEANS

a. **Theorem**: Let P and R be two problems. If P $\leq_P$ R (i.e., P reduces to R) and R is polynomial, then P is polynomial. (The theorem we just proved)

b. That means that when P $\leq_P$ R, if we can solve R, then we can solve P

c. By solving R we mean "finding a poly-time algorithm for R"

d. Metaphorically, solving R is like "*conquering*" R

$$\text{solve} \equiv \text{conquer}$$

e. Thus part (b) above can be translated as: when P $\leq_P$ R, if we can conquer R, then we can conquer P

f. For "if we can conquer R, then we can conquer P" to be true, R must be at least as hard (as tough) as P

g. Thus P $\leq_P$ R implies that R is at least as hard as P, or, casually R is harder than P, or, P is easier than R (in polynomial-time sense).

h. $\leq_P$ conveys "easier"

DEFINITION OF NP-COMPLETENESS

- **Definition**: A problem R is **NP complete** if
  - R is NP
  - Every NP problem P reduces to R (i.e., P $\leq_P$ R)
- An equivalent but informal definition:
  - An NP problem R is NP-complete if it is the "most difficult" of all NP problems
  - More precisely, if R is at least as hard as all NP problems

HOW TO PROVE A PROBLEM NP-COMPLETE?

- To prove a yes-no problem R to be NP complete, we need to show that it meets both conditions of the definition:
    1. Prove R to be NP, which is easy: develop a guess-verify NP algorithm for it
    2. Prove that R is at least as hard as every conceivable NP problem
- **Dilemma**: The second part is extremely problematic (Why?)
    - Not all the NP problems are known to us to compare to R
    - The set of NP problems is potentially infinite, so we'll never finish comparing R to each of them

A WAY OUT

-- MAKING THE PROOF OF NP-COMPLETENESS FEASIBLE --

- Analogy first: Similarity to boxing championship
  - How can we prove that a new boxer is the strongest?
    - We can have him box and defeat (or tie with) each boxer in the world –- too time consuming
    - Or, let him box and beat (or tie with) the current word champion
- **Theorem**: A problem R is NP-complete if
  - R is NP, and
  - There exists an NP-complete problem R_0 that reduces to R (i.e., $R_0 \leq_P R$)

That is, R_0 is a world champion, and by showing $R_0 \leq_P R$, we establish that R is at least as hard as R_0 and so R is a world champion

A WAY OUT

-- PROOF THAT IT WORKS --

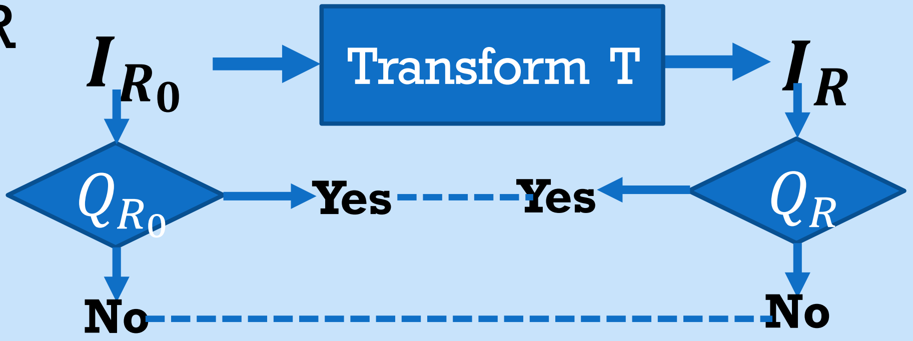
- **Theorem**: A problem R is NP-complete if
  1. R is NP, and
  2. There is an NP-complete problem R_0 that reduces to R (i.e., $R_0 \leq_P R$)
- **Proof**:
  - Since R is NP, it remains to show that every arbitrary NP problem P reduces to R (i.e., $P \leq_P R$)
  - Let P be an arbitrary NP problem
  - Since R_0 is NP-complete, it follows that $P \leq_P R_0$
  - Since $R_0 \leq_P R$ (by assumption 2 of the theorem), we have

$$P \leq_P R_0 \leq_P R$$

  - Therefore $P \leq_P R$ (by transitivity of reduction). Q.E.D.

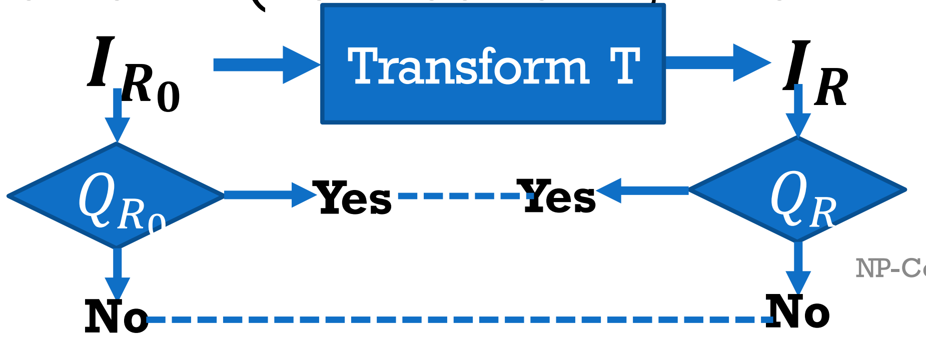
STRATEGY FOR PROVING NP-COMPLETENESS

- The previous theorem amounts to a strategy for proving new problems to be NP complete.
- **Strategy**: to prove a new problem R to be NP-complete, the following steps are sufficient:
  1. Prove R to be NP (by easily developing a poly-time guess-verify algorithm)
  2. Find an already known NP-complete problem R_0, and come up with a transform that reduces R_0 to R



LESSONS LEARNED SO FAR

- NP problems can be solved exponentially but we don't know if they are inherently exponential or are polynomial
- To prove a yes-no problem ∈ NP, design a guess-verify algorithm where "verify" is deterministically polynomial, and similar to Bound in Backtracking
- Transforms (aka, reductions) are an essential tool in NP-complete theory, and can be generalized and applied in other contexts and for other purposes
- A strategy for proving a new NP problem R to be NP-complete: reduce a previously known NP-complete problem R_0 to the new problem R by constructing a polynomial-time answer-preserving transform from the old problem R_0 to the new problem R (not vice versa, which is a common mistake)



NEED FOR A FIRST NP-COMPLETE PROBLEM

- For this strategy to become effective, we need at least one NP-complete problem R_0. Cook gave us the first one: SAT
- **Cook's Theorem**: **SAT is NP-complete**.
- **Proof**:
  - Beyond the scope of this course. Several pages long.
  - Nevertheless, the proof has to deal with the dilemma of showing that SAT is at least as hard as every NP problem
  - Cook dealt with it using **modeling**
    - He came up with a model that models every NP problem
  - The model is **Turing Machine (TM)**

USEFULNESS OF THAT STRATEGY

- With that strategy, thousands of important problems have been proved to be NP-complete
- Many of those problems are extremely practical and widely used, such as various versions of scheduling, optimization, etc.
- We will show the strategy at work in a little bit
- But first, some general thoughts about a couple of "big questions"

IMPORTANCE OF $P \overset{?}{=} NP$?

- Recall the most famous open question in CS: $P \overset{?}{=} NP$
- If it turns out that $P = NP$, then all NP problems, including NP-complete problems, can be solved in polynomial time, with tremendous economic implications (savings) because
  - many important problems can be computed fast,
  - saving humanity a lot of time and increasing productivity
- But, though there is no proof yet, mounting evidence is pointing towards $\boldsymbol{P \neq NP}$
- Many "proofs" for the equality or inequality have been put forth, but all turned out to be flawed – the question is still open!!!

IMPLICATIONS FOR A PROBLEM TO BE NP-COMPLETE

- Since most likely $P \neq NP$, chances are that all NP-complete problems are non-polynomial, i.e., exponential
  - Sad but most likely true
- So, when we try hard to find a fast algorithm for a new problem and fail, we suspect that it is NP-complete
- If we succeed in proving it to be NP-complete, then, at least for the foreseeable future, we should give up on solving it polynomially (and tell your boss that news)
- Instead, we settle for fast **approximation** algorithms, using the greedy method, B&B, randomized algorithms, or other heuristics
  - That is a huge separate area of algorithm design

LESSONS LEARNED SO FAR

- NP problems can be solved exponentially but we don't know if they are inherently exponential or are polynomial
- To prove a yes-no problem ∈ NP, design a guess-verify algorithm where "verify" is deterministically polynomial, and similar to Bound in Backtracking
- Transforms (aka, reductions) are an essential tool in NP-complete theory, and can be generalized and applied in other contexts and for other purposes
- A strategy for proving a new NP problem NP-complete: reduce a known NP-complete problem to the new problem by a poly answer-preserving transform
- Mounting evidence points to $\boldsymbol{P \neq NP}$
- Thus, in the foreseeable future, avoid trying to solve NP-complete problems polynomially, and instead look for heuristic approximation algorithms (that give near-optimal solutions)

THE K-CLIQUE PROBLEM IS NP-COMPLETE

- We will use the strategy to prove that CLIQUE is NP-complete
- First, we need to show the k-clique problem is NP, but we showed that already some time back
- Now we need to do the bigger part:
  1. Select a convenient, already known NP-complete problem R_0
     - We will take SAT
  2. Construct a <u>valid transform</u> T from R_0 to k-clique
     - By valid we mean: mapping I_{SAT} to I_{CLIQUE}, (i.e., a Bool expr F to (G,k)), polynomial, and answer-preserving

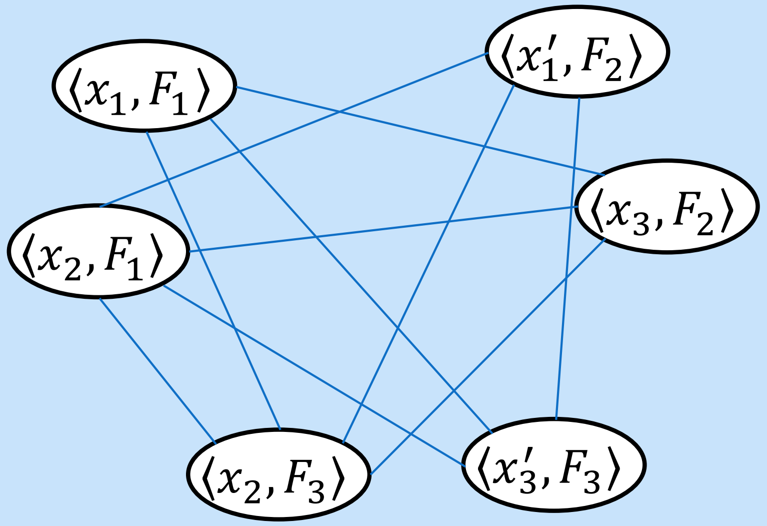
THE K-CLIQUE PROBLEM IS NP-COMPLETE
-- THE TRANSFORM FROM SAT TO CLIQUE (1/3) --

- On this (left) side, we'll construct the transform T and prove its validity
- On the right side, we show an example illustrating both the transform T and its validity proof
- Let F be a Boolean expression, in the form of a product of sums: $F = F_1F_2 \dots F_r$, for some $r \geq 1$
  - Every factor F_i is a sum of literals (a *literal* is a Boolean variable x or its complement x')
- Take $k = r$ and $G(V, E)$ defined as follows:
  - $V = \{ \langle x_i, F_j \rangle \mid x_i \text{ is a literal in } F_j \text{ for some } j\}$
  - $E = \{ (\langle x_i, F_j \rangle, \langle y_s, F_t \rangle) \mid j \neq t \text{ and } x_i \neq y_s' \}$
- Think of the node labels $\langle x_i, F_j \rangle$ as made up of "first name" (x_i) and "family name" (F_j)
- No two nodes from the same family are adjacent, and no two nodes from different families are adjacent if their first names are "opposite"

Time: $O(|F|^2)$ => Poly

Example: $F = (x_1 + x_2)(x_1' + x_3)(x_2 + x_3')$

- So, $F_1 = x_1 + x_2$, $F_2 = x_1' + x_3$, $F_3 = x_2 + x_3'$
- $r = 3$, so we take $k = 3$
- The graph $G(V, E)$ is:



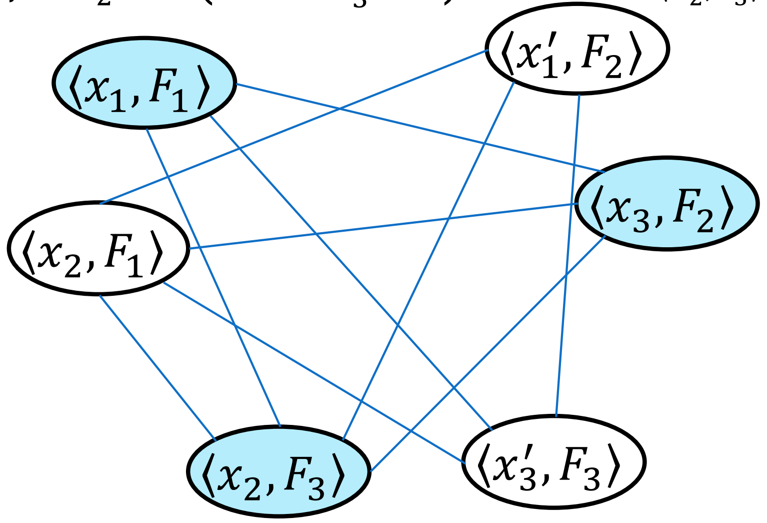
$\langle x_1, F_1 \rangle$
$\langle x_1', F_2 \rangle$
$\langle x_3, F_2 \rangle$
$\langle x_2, F_1 \rangle$
$\langle x_2, F_3 \rangle$
$\langle x_3', F_3 \rangle$

THE K-CLIQUE PROBLEM IS NP-COMPLETE
-- THE TRANSFORM FROM SAT TO CLIQUE (2/3) --

- Answer-preserving: We will prove that F is satisfiable if and only if (iff) G has a k-clique
- Assume first that F is satisfiable, and prove that G has a k-clique
- Since F is satisfiable, there is an assignment that makes F equal to 1
- This implies that $F_1 = 1, F_2 = 1, \dots, F_r = 1$
- Therefore, in every factor F_i there is (at least) one variable assigned 1. Call that variable z_i
- Observe that $\langle z_1, F_1\rangle, \langle z_2, F_2\rangle, \dots, \langle z_k, F_k\rangle$ is a k-clique in G because
    - they are k distinct nodes, and
    - each pair $(\langle z_i, F_i\rangle, \langle z_j, F_j\rangle)$ forms an edge since the endpoints come from different factors and $z_i \neq z_j'$ due to the fact that they are both assigned 1 (in complements, if one is 1, the other must be 0).

Example: $F = (x_1 + x_2)(x_1' + x_3)(x_2 + x_3')$

- So, $F_1 = x_1 + x_2, F_2 = x_1' + x_3, F_3 = x_2 + x_3'$; $k = r = 3$
- F_1=1, so x_1 or x_2 must be 1. Say $x_1 = 1$. $\langle x_1, F_1\rangle$
- F_2=1, so $x_3 = 1$ (since $x_1' = 0$). $\langle x_3, F_2\rangle$
- F_3=1, so $x_2 = 1$ (since $x_3' = 0$) $\langle x_2, F_3\rangle$


$\langle x_1, F_1\rangle$
$\langle x_1', F_2\rangle$
$\langle x_3, F_2\rangle$
$\langle x_2, F_1\rangle$
$\langle x_2, F_3\rangle$
$\langle x_3', F_3\rangle$

- Note how the corresponding nodes form a 3-clique

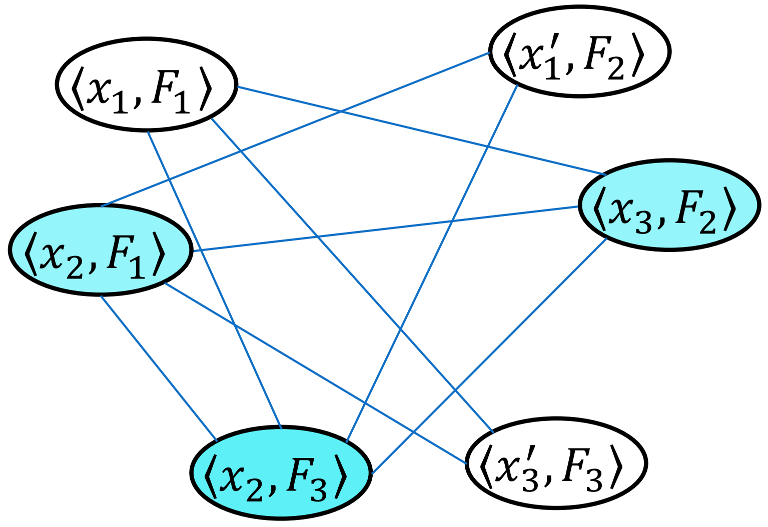
THE K-CLIQUE PROBLEM IS NP-COMPLETE

-- THE TRANSFORM FROM SAT TO CLIQUE (3/3) --

- Next, prove that if G has a k-clique, then F is satistiable
- Assume G has a k-clique $\langle u_1, F_1\rangle, \langle u_2, F_2\rangle, \dots, \langle u_k, F_k\rangle$ which are pairwise adjacent
- These k nodes come from the k different factors, one node per factor, because no two nodes from the same factor are adjacent
- Furthermore, no two u_i and u_j are complements because the two nodes $\langle u_i, F_i\rangle, \langle u_j, F_j\rangle$ are adjacent, and adjacent nodes have non-complement first-names
- Thus, we can consistently assign each u_i the value 1.
- This assignment makes each F_i equal to 1 because u_i is one of the additive literals in F_i
- Consequently, $F = F_1 F_2 \dots F_r$=1.1. … .1=1, and so F is satisfiable.
- Hence, SAT reduces to CLIQUE, and thus, CLIQUE is NP-complete. Q.E.D.

Example: $F = (x_1 + x_2)(x_1' + x_3)(x_2 + x_3')$

- $F_1 = x_1 + x_2, F_2 = x_1' + x_3, F_3 = x_2 + x_3'$
- Take the 3-clique $\langle x_2, F_1\rangle, \langle x_3, F_2\rangle, \langle x_2, F_3\rangle$,


$\langle x_1, F_1\rangle$
$\langle x_1', F_2\rangle$
$\langle x_3, F_2\rangle$
$\langle x_2, F_1\rangle$
$\langle x_2, F_3\rangle$
$\langle x_3', F_3\rangle$

- So, assign x_2=1, x_3=1 (and whatever to x_1). You can see that $F_1 = x_1 + x_2 = x_1 + 1 = 1, F_2 = x_1' + x_3 = x_1' + 1 = 1$, $F_3 = x_2 + x_3' = 1 + 1' = 1 + 0 = 1$, and thus :$F$=1

SOME CLOSING THOUGHTS

-- TURNING OPTIMIZATION PROBLEMS INTO YES-NO PROBLEMS --

- Given an optimization problem, where for a given input I, the question is: "Find a minimum solution for I"
- We can turn that problem into a yes-no problem (we did same for TSP):
  - Input: Same input I, but also a number d
  - Question: Does I have a solution of cost $\leq d$?
- If we prove the yes-no version to be NP-complete, that means it is most likely non-polynomial
- Since the original (optimization) problem is at least as hard, and since the simpler yes-no version is hard enough to be most likely non-polynomial, we conclude that the original problem is most likely non-polynomial

SOME CLOSING THOUGHTS
-- PROVING SIMPLER VERSIONS NP-COMPLETE --

- Even if a problem is originally a yes-no problem, sometimes it is easier if we can take a simpler version of that problem and prove it to be NP-complete
- Example: 3-SAT is a special case of SAT where every factor has at most 3 literals. If 3-SAT is NP-complete (which it is), then it follows that the general SAT is at least NP-complete

SOME CLOSING THOUGHTS
-- NP-HARD --

- A problem P is NP-hard if it satisfies the second condition of NP-completeness
  - That is, if every NP problem reduces to P
- In other terms, an NP-hard problem need not be NP itself
- Clearly, every NP-complete problem is NP-hard, but not necessarily vice versa
- Typically, if a yes-no version of a (e.g., optimization) problem is NP-complete, we say that the original (non yes-no) problem is NP-hard

SOME CLOSING THOUGHTS
-- THE LIST OF KNOWN NP-COMPLETE PROBLEMS --

- Since the time that the NP-complete theory started (and Cook's theorem proved) in the early 1970's, computer scientists have been proving many problems to be NP-hard or NP-complete
- **Tens of thousands** of problems have been proved NP-hard, and more problems are so proved every day

SOME CLOSING THOUGHTS

-- IT HAS BECOME EASIER TO PROVE NP-COMPLETENESS --

- It has become easier over the years to prove a new problem to be NP-hard or NP-complete for two reasons

1. We now have an ever wider list of known NP-complete problems to choose from and reduce to the new problems: choose the problem more akin to new problem
2. Scientists have developed many techniques for creating transforms, which we can use for creating new transforms or adapting existing transforms

SOME CLOSING THOUGHTS

-- A BIT OF HISTORY ABOUT NP-COMPLETENESS PROOFS --

- In the early days of NP-complete theory, proving a new NP-complete would constitute a PhD dissertation
- As we gained more experience with the theory, a new NP-completeness proof could only be accepted as a journal/conference paper
- Later on, they became only homework problems
- Now, they're barely pop quizzes ☺

MORE PROBLEMS TO PROVE NP-COMPLETE

- **Exercise 1**: Prove that CLIQUE $\leq_P$ HAM. Once proved, we will conclude that HAM is NP-complete. (Recall that HAM is the Hamiltonian cycle problem).
- **Exercise 2**: Prove that HAM $\leq_P$ TSP. (Recall that TSP is the yes-no version of the traveling salesman problem.) Again, once proved, we conclude that TSP is NP-complete

THIS IS THE END

- This is the end of the course
- It has been great having you
- I hope you have learned a lot from this course this semester
- Good luck on the final exam!