NP-Complete Problems: Reductions

CSCI 3100

Recall…

Last time, we said that a problem is NP-Complete if it is:

- In the NP class of problems and
- Is hard as any other problem in the NP class

Today:

- Why do we care to know if a problem is NP-Complete
- How can we show that a problem is as hard as any other problem?
- Examples

Why does it matter?

Suppose your boss tells you to code up a solution to some problem. This problem happens to be NP-Complete. You know that you don't have an efficient way to solve it.

- You won't waste time trying to come up with an exact solution
- Can use some alternative techniques that find a "good enough" solution

Suppose you come up with an efficient algorithm to a problem you know to be NP-Complete.

- There's a $1,000,000 prize for you!
- Do share it with your Algorithms instructor

Proving NP-Completeness using reductions

Given a problem S, transform/reduce some other known NP-Complete problem X to S such that

- If the answer to S is true then the answer to X is true
- If the answer to S is false, then the answer to X is false

Perform the transformation in polynomial time.

Then if there is a polynomial time algorithm for S, there is a polynomial time algorithm to X

The first NP-Complete Problem

Circuit Satisfiability Problem (SAT)

Given a boolean formula: $F(x_1, x_2, \ldots, x_n)$, can F evaluate to true?

Formula is usually given in Conjunctive Normal Form:

- Conjunction (AND clauses) of disjunctions (OR clauses)
- Example: (a + b + c)^(a' + b' + c)^(a + b' + c')^(a' + b + c').
- a' means "not a"
- ^ means AND
- + means OR

An arbitrary SAT instance can be transformed to CNF in linear time

- Tseytin transformation

Which assignment of variables satisfies this SAT formula: (a+b+c')^(a'+b+c)^(a'+b'+c')

A. a = T, b = F, c = F

B. a = F, b = T, c = T

C. a = T, b = T, c = T

D. a = T, b = F, c = F

E. None of the above

k-SAT

SAT instance in CNF with *k* literals in each disjunction

A literal is a variable or its negation

2-SAT is in P

- Algorithm can be found here (https://www.geeksforgeeks.org/2-satisfiability-2-sat-problem/)

3-SAT is NP-Complete

k-SAT is NP-complete, for k > 2

The Clique Problem

A clique of an undirected graph G = (V, E) is a subset $V' \subseteq V$ where

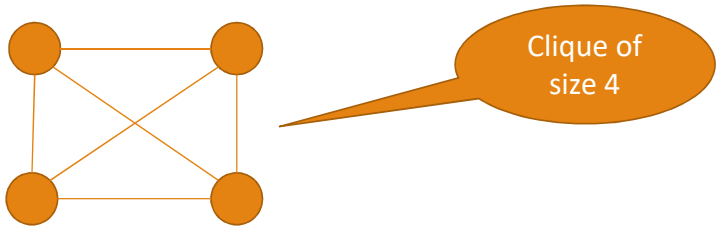
- For each pair of vertices x, y in V' there is an edge (x, y) in E

The clique problem as an optimization problem:

- Given an undirected graph, find a clique of maximum size

The clique problem as a decision problem:

- Given an undirected graph, is there a clique of size at least K in this graph



The clique problem is NP-Complete

Show that k-clique can be verified in polynomial time (trivial)

Reduce some other NP-Complete problem to Clique in polynomial time

- We'll reduce 3-SAT to Clique

Reduction from 3-SAT to Clique

Let F = C_1^C_2^…^C_k be the 3-SAT formula with k clauses

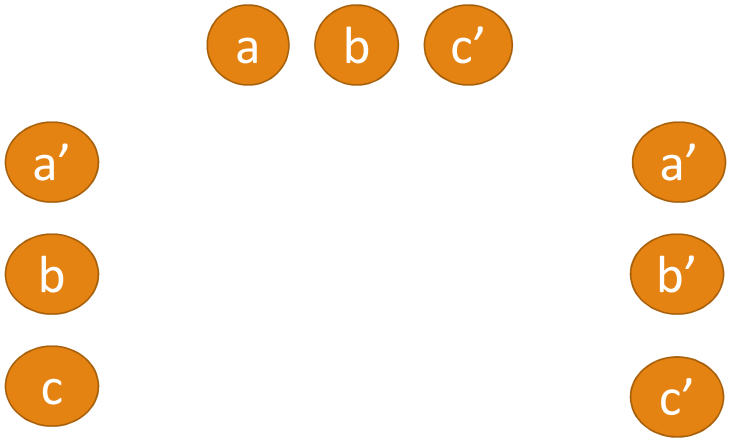
- Here, each C_i is a disjunction of 3 literals

We will construct a graph G, such that F is satisfiable if and only if G has a clique of size k

Each clause C_i has exactly 3 literals: L_1^i, L_2^i, L_3^i

Create a vertex for each literal of each clause

Create a vertex for each literal of each clause of (a+b+c')^(a'+b+c)^(a'+b'+c')

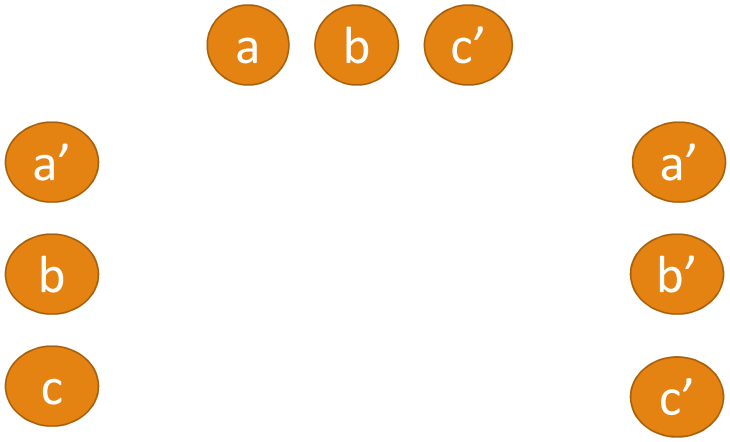


Add edges between pairs of vertices L_i^r, L_j^s that satisfy both of the following

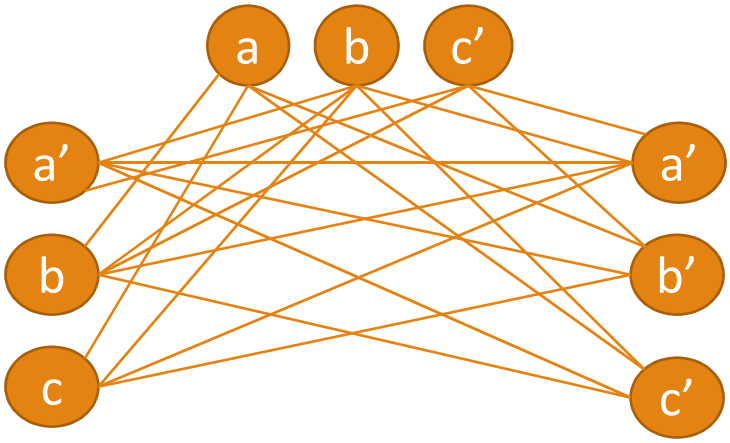
r ≠ s (no edges between literals that are in the same clause)

L_i^r is not a negation of L_j^s (no edges between x and x')

Add edges for the vertices created from this 3-SAT instance (a+b+c')^(a'+b+c)^(a'+b'+c')



Add edges for the vertices created from this 3-SAT instance (a+b+c')^(a'+b+c)^(a'+b'+c')



3-SAT instance F = C_1^C_2^…^C_k is satisfiable IFF graph G we constructed has a k-clique

IFF means if and only if

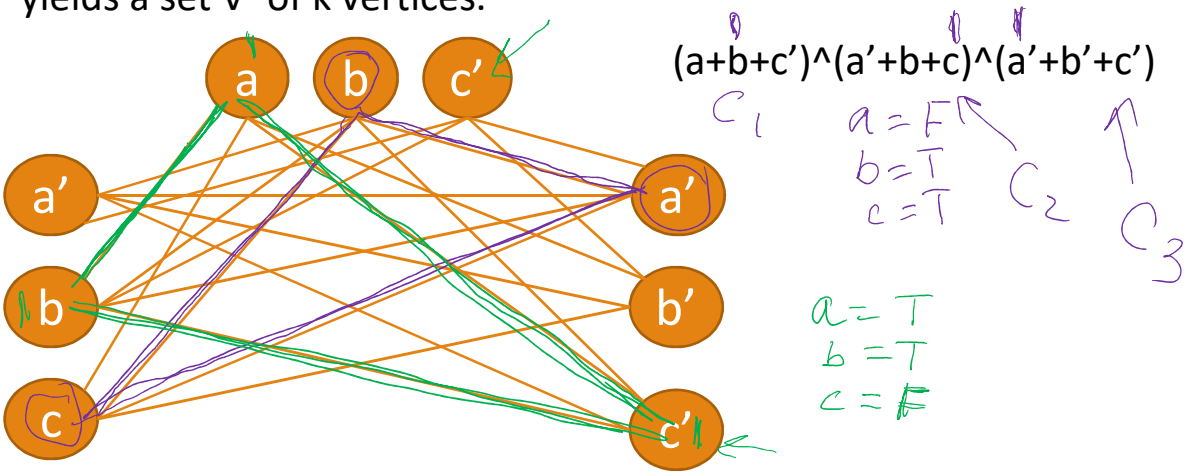
We must show that

- If 3-SAT instance F is satisfiable, then G has a k-clique

AND

- If G has a k-clique, then the 3-SAT instance is satisfiable

Suppose F = C_1^C_2^…^C_k is satisfiable. Then each clause has at least one literal that is assigned 1. Each such literal corresponds to a vertex L_i^r. Picking one such "true" literal from each clause yields a set V' of k vertices.



Suppose F = C_1^C_2^…^C_k is satisfiable. Then each clause has at least one literal that is assigned 1. Each such literal corresponds to a vertex L_i^r. Picking one such "true" literal from each clause yields a set V' of k vertices. V' is a clique. Why?

A. We picked vertices from different clauses, and their corresponding literals evaluate to 1. Therefore, these vertices/literals cannot be negations of each other. So there must be an edge connecting them.

B. We picked vertices from different clauses, and we know that there are edges connecting all pairs of vertices (x, y) if x and y come from different clauses.

C. Suppose V' is a clique. Then all vertices of V' are connected to each other, which means that these vertices are not negations of each other. This is consistent with their corresponding literals being assigned to 1.

Suppose G has a clique V' of size k. No edges of G connect vertices from the same clause, so V' has exactly one literal from each clause. We can assign those vertices/literals to 1, since there are no edges between a literal and its complement (negation). This assignment satisfies the 3-SAT problem instance.

A. This is a valid proof

B. This proof is flawed because some literals of the 3-SAT problem may remain unassigned (if the clique contains vertices that are the same variable)

C. This proof is flawed because the last statement "This assignment satisfies the 3-SAT problem instance" is false.

If we had a polynomial time algorithm for the Clique problem, we could solve 3-SAT in polynomial time

Given: an instance of a 3-SAT problem

Construct graph G in polynomial time (as described earlier)

Solve the k-Clique problem on G

If G has a k-Clique, assign the literals of the clique to 1, derive variable assignment for 3-SAT

Any NP-Complete problem can be reduced to any other NP-Complete problem in polynomial time.