CS 561, Lecture 12

Jared Saia
University of New Mexico

Today's Outline

- P, NP and NP-Hardness
- Reductions
- Approximation Algorithms

Efficient Algorithms

- Q: What is a minimum requirement for an algorithm to be efficient?
- A: A long time ago, theoretical computer scientists decided that a minimum requirement of any efficient algorithm is that it runs in polynomial time: $O(n^c)$ for some constant c
- People soon recognized that not all problems can be solved in polynomial time but they had a hard time figuring out exactly which ones could and which ones couldn't

NP-Hard Problems

- Q: How to determine those problems which can be solved in polynomial time and those which can not
- Again a long time ago, Steve Cook and Dick Karp and others defined the class of *NP-hard* problems
- Most people believe that NP-Hard problems *cannot* be solved in polynomial time, even though so far nobody has *proven* a super-polynomial lower bound.
- What we do know is that if *any* NP-Hard problem can be solved in polynomial time, they *all* can be solved in polynomial time.

Circuit Satisfiability

- **Circuit satisfiability** is a good example of a problem that we don't know how to solve in polynomial time
- In this problem, the input is a *boolean circuit*: a collection of and, or, and not gates connected by wires
- We'll assume there are no loops in the circuit (so no delay lines or flip-flops)

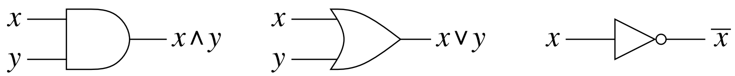
Circuit Satisfiability

- The input to the circuit is a set of m boolean (true/false) values $x_1, \ldots x_m$
- The output of the circuit is a single boolean value
- Given specific input values, we can calculate the output in polynomial time using depth-first search and evaluating the output of each gate in constant time

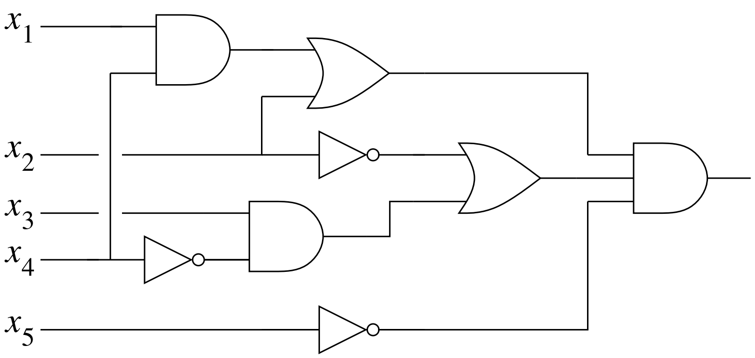
Circuit Satisfiability

- The circuit satisfiability problem asks, given a circuit, whether there is an input that makes the circuit output **True**
- In other words, does the circuit always output false for any collenction of inputs
- Nobody knows how to solve this problem faster than just trying all 2^m possible inputs to the circuit but this requires exponential time
- On the other hand nobody has every proven that this is the best we can do!

Example



An and gate, an or gate, and a not gate.



A boolean circuit. Inputs enter from the left, and the output leaves to the right.

Classes of Problems

We can characterize many problems into three classes:

- **P** is the set of yes/no problems that can be solved in polynomial time. Intuitively P is the set of problems that can be solved "quickly"
- **NP** is the set of yes/no problems with the following property: If the answer is yes, then there is a *proof* of this fact that can be checked in polynomial time
- **co-NP** is the set of yes/no problems with the following property: If the answer is no, then there is a *proof* of this fact that can be checked in polynomial time

NP

- **NP** is the set of yes/no problems with the following property: If the answer is yes, then there is a *proof* of this fact that can be checked in polynomial time
- Intuitively NP is the set of problems where we can verify a **Yes** answer quickly if we have a solution in front of us
- For example, circuit satisfiability is in NP since if the answer is yes, then any set of m input values that produces the **True** output is a proof of this fact (and we can check this proof in polynomial time)

P,NP, and co-NP

- If a problem is in P, then it is also in NP — to verify that the answer is yes in polynomial time, we can just throw away the proof and recompute the answer from scratch
- Similarly, any problem in P is also in co-NP
- In this sense, problems in P can only be easier than problems in NP and co-NP

Examples

- The problem: "For a certain circuit and a set of inputs, is the output **True**?" is in P (and in NP and co-NP)
- The problem: "Does a certain circuit have an input that makes the output **True**?" is in NP
- The problem: "Does a certain circuit always have output true for any input?" is in co-NP

P Examples

Most problems we've seen in this class so far are in P including:

- "Does there exist a path of distance $\leq d$ from u to v in the graph G?"
- "Does there exist a minimum spanning tree for a graph G that has cost $\leq c$?"
- "Does there exist an alignment of strings s_1 and s_2 which has cost $\leq c$?"

NP Examples

There are also several problems that are in NP (but probably not in P) including:

- **Circuit Satisfiability**
- **Coloring**: "Can we color the vertices of a graph G with c colors such that every edge has two different colors at its endpoints (G and c are inputs to the problem)
- **Clique**: "Is there a clique of size k in a graph G?" (G and k are inputs to the problem)
- **Hamiltonian Path**: "Does there exist a path for a graph G that visits every vertex exactly once?"

The \$1 Million Question

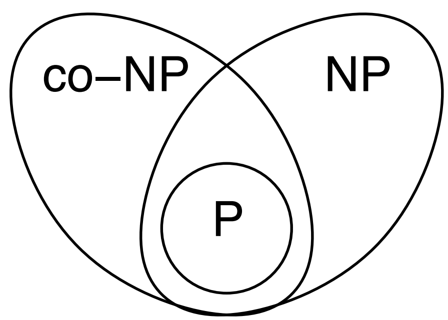
- The most important question in computer science (and one of the most important in mathematics) is: "Does P=NP?"
- Nobody knows.
- Intuitively, it *seems* obvious that P≠NP; in this class you've seen that some problems can be very difficult to solve, even though the solutions are obvious once you see them
- But nobody has proven that P≠NP

NP and co-NP

- Notice that the definition of NP (and co-NP) is not symmetric.
- Just because we can verify every yes answer quickly doesn't mean that we can check no answers quickly
- For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable
- In other words, we know that Circuit Satisfiability is in NP but we don't know if its in co-NP

Conjectures

- We conjecture that P≠NP and that NP≠co-NP
- Here's a picture of what we *think* the world looks like:



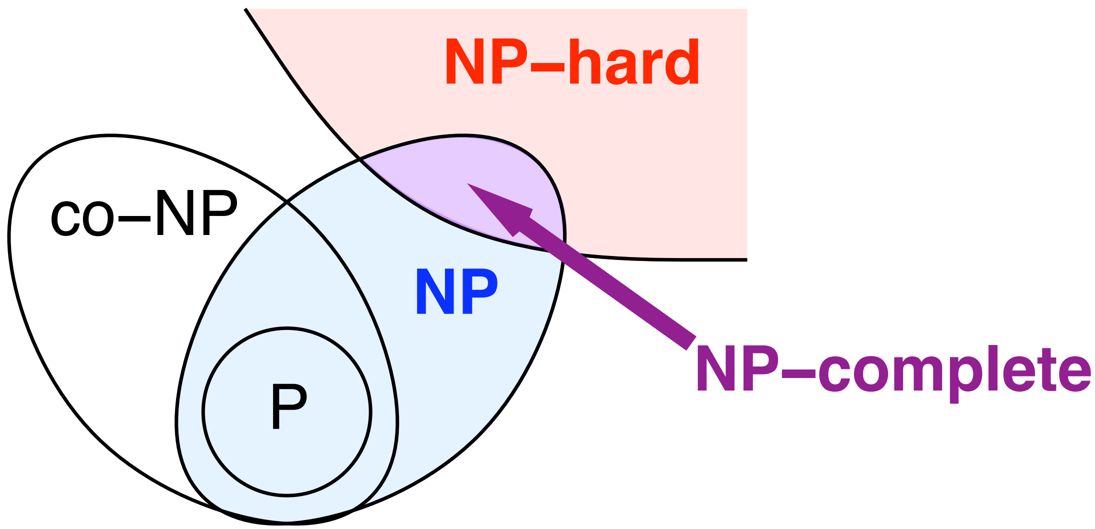
NP-Hard

- A problem Π is **NP-hard** if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for *every problem in NP*
- In other words: **Π is NP-hard iff If Π can be solved in polynomial time, then P=NP**
- In other words: if we can solve one particular NP-hard problem quickly, then we can quickly solve *any* problem whose solution is quick to check (using the solution to that one special problem as a subroutine)
- If you tell your boss that a problem is NP-hard, it's like saying: "Not only can't I find an efficient solution to this problem but neither can all these other very famous people." (you could then seek to find an approximation algorithm for your problem)

NP-Complete

- A problem is *NP-Easy* if it is in NP
- A problem is *NP-Complete* if it is NP-Hard and NP-Easy
- In other words, a problem is NP-Complete if it is in NP but is at least as hard as all other problems in NP.
- If anyone finds a polynomial-time algorithm for even one NP-complete problem, then that would imply a polynomial-time algorithm for *every* NP-Complete problem
- *Thousands* of problems have been shown to be NP-Complete, so a polynomial-time algorithm for one (i.e. all) of them is incredibly unlikely

Example



A more detailed picture of what we *think* the world looks like.

Proving NP-Hardness

- In 1971, Steve Cook proved the following theorem: **Circuit Satisfiability is NP-Hard**
- Thus, one way to show that a problem A is NP-Hard is to show that if you can solve it in polynomial time, then you can solve the Circuit Satisfiability problem in polynomial time.
- This is called a *reduction*. We say that we *reduce* Circuit Satisfiability to problem A
- This implies that problem A is "as difficult as" Circuit Satisfiability.

SAT

- Consider the *formula satisfiability* problem (aka *SAT*)
- The input to SAT is a boolean formula like

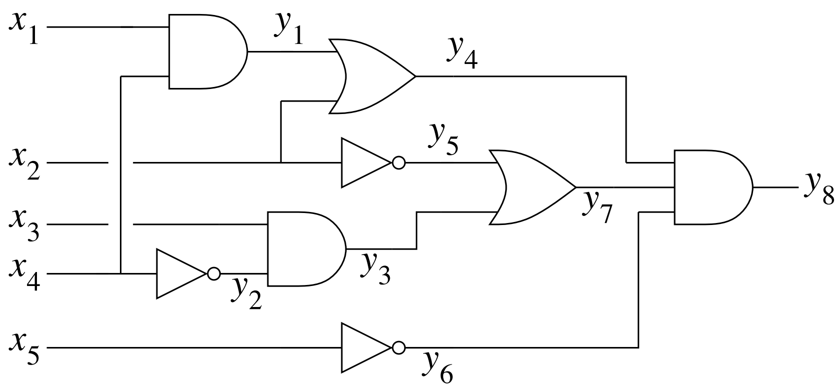
$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

- The question is whether it is possible to assign boolean values to the variables $a, b, c, \ldots$ so that the formula evaluates to TRUE
- To show that SAT is NP-Hard, we need to show that we can use a solution to SAT to solve Circuit Satisfiability

The Reduction

- *Given a boolean circuit, we can transform it into a boolean formula by creating new output variables for each gate and then just writing down the list of gates separated by AND*
- This simple algorithm is the reduction
- For example, we can transform the example ciruit into a formula as follows:

Example



$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge (y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (y_8 = y_4 \wedge y_7 \wedge y_6) \wedge y_8$$

A boolean circuit with gate variables added, and an equivalent boolean formula.

Reduction Picture

$$\begin{tikzcd} \boxed{\text{boolean circuit}} \arrow[r, "O(n)"] & \boxed{\text{boolean formula}} \arrow[d, Rightarrow, "\text{SAT}"] \\ \boxed{\text{True or False}} & \boxed{\text{True or False}} \arrow[l, "\text{trivial}"'] \end{tikzcd}$$

Reduction

- The original circuit is satisifiable iff the resulting formula is satisfiable
- We can transform any boolean circuit into a formula in linear time using DFS and the size of the resulting formula is only a constant factor larger than the size of the circuit
- Thus we've shown that if we had a polynomial-time algorithm for SAT, then we'd have a polynomial-time algorithm for Circuit Satisfiability (and this would imply that P=NP)
- This means that SAT is NP-Hard

Showing NP-Completeness

- We've shown that SAT is NP-Hard, to show that it is NP-Complete, we now must also show that it is in NP
- In other words, we must show that if the given formula is satisfiable, then there is a proof of this fact that can be checked in polynomial time
- To prove that a boolean formula is satisfiable, we only have to specify an assignment to the variables that makes the formula true (this is the "proof" that the formula is true)
- Given this assignment, we can check it in linear time just by reading the formula from left to right, evaluating as we go
- So we've shown that SAT is NP-Hard and that SAT is in NP, thus SAT is NP-Complete

Take Away

- In general to show a problem is NP-Complete, we first show that it is in NP and then show that it is NP-Hard
- To show that a problem is in NP, we just show that when the problem has a “yes” answer, there is a proof of this fact that can be checked in polynomial time (this is usually easy)
- To show that a problem is NP-Hard, we show that if we could solve it in polynomial time, then we could solve some other NP-Hard problem in polynomial time (this is called a reduction)

3-SAT

- A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (and) of several *clauses*, each of which is the disjunction (or) or several *literals*, each of which is either a variable or its negation. For example:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

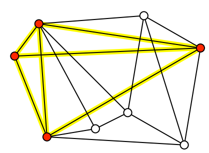
- A *3CNF* formula is a CNF formula with exactly three literals per clause
- The 3-SAT problem is just: "Is there any assignment of variables to a 3CNF formula that makes the formula evaluate to true?"

3-SAT

- 3-SAT is just a restricted version of SAT
- Surprisingly, 3-SAT also turns out to be NP-Complete (proof omitted for now)
- 3-SAT is very useful in proving NP-Hardness results for other problems, we'll see how it can be used to show that CLIQUE is NP-Hard

CLIQUE

- The last problem we'll consider in this lecture is CLIQUE
- The problem CLIQUE asks "Is there a clique of size k in a graph G?"
- Example graph with clique of size 4:



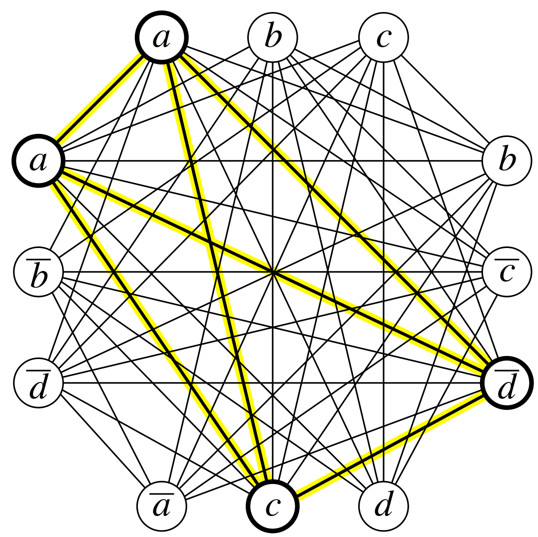
- We'll show that Clique is NP-Hard using a reduction from 3-SAT. (the proof that Clique is in NP is left as an exercise)

The Reduction

- Given a 3-CNF formula F, we construct a graph G as follows.
- The graph has one node for each instance of each literal in the formula
- Two nodes are connected by an edge if: (1) they correspond to literals in different clauses and (2) those literals do not contradict each other

Reduction Example

- Let F be the formula: $(a \vee b \vee c) \wedge (b \vee \overline{c} \vee \overline{d}) \wedge (\overline{a} \vee c \vee d) \wedge (a \vee \overline{b} \vee \overline{d})$
- This formula is transformed into the following graph:



(look for the edges that *aren't* in the graph)

Reduction

- Let F have k clauses. Then G has a clique of size k iff F has a satisfying assignment. The proof:
- **k-clique $\implies$ satisfying assignment:** If the graph has a clique of k vertices, then each vertex must come from a different clause. To get the satisfying assignment, we declare that each literal in the clique is true. Since we only connect non-contradictory literals with edges, this declaration assigns a consistent value to several of the variables. There may be variables that have no literal in the clique; we can set these to any value we like.
- **satisfying assignment $\implies$ k-clique:** If we have a satisfying assignment, then we can choose one literal in each clause that is true. Those literals form a k-clique in the graph.

Reduction Picture

$$\begin{tikzcd} \boxed{\text{3CNF formula with } k \text{ clauses}} \arrow[r, "O(n)"] & \boxed{\text{graph with } 3k \text{ nodes}} \arrow[d, Rightarrow, "\text{Clique of size } k?"] \\ \boxed{\text{True or False}} & \boxed{\text{True or False}} \arrow[l, "trivial"'] \end{tikzcd}$$

In-Class Exercise

Consider the formula: $(a \vee b) \wedge (b \vee \overline{c}) \wedge (c \vee \overline{b})$

- Q1: Transform this formula into a graph, G, using the reduction just given.
- Q2: What is the maximum clique size in G? Give the vertices in this maximum clique.

Independent Set

- Independent Set is the following problem: "Does there exist a set of k vertices in a graph G with no edges between them?"
- In the hw, you'll show that independent set is NP-Hard by a reduction from CLIQUE
- Thus we can now use Independent Set to show that other problems are NP-Hard

Vertex Cover

- A *vertex cover* of a graph is a set of vertices that touches every edge in the graph
- The problem *Vertex Cover* is: “Does there exist a vertex cover of size k in a graph G?”
- We can prove this problem is NP-Hard by an easy reduction from Independent Set

Key Observation

- Key Observation: If I is an independent set in a graph $G = (V, E)$, then $V - I$ is a vertex cover.
- Thus, there is an independent set of size k iff there is a vertex cover of size $|V| - k$.
- For the reduction, we want to show that a polynomial time algorithm for Vertex Cover can give a polynomial time algorithm for Independent Set

The Reduction

- We are given a graph $G = (V, E)$ and a value k and we must determine if there is an independent set of size k in G.
- To do this, we ask if there is a vertex cover of size $|V| - k$ in G.
- If so then we return that there *is* an independent set of size k in G
- If not, we return that there *is not* an independent set of size k in G

The Reduction

$$
\begin{tikzcd}
\boxed{\text{graph } G = (V, E),\ \ k} \arrow[r, "\text{trivial}"] & \boxed{\text{graph } G = (V, E),\ \ |V| - k} \arrow[d, Rightarrow, "\text{VertexCover}"] \\
\boxed{\text{True or False}} & \boxed{\text{True or False}} \arrow[l, "O(1)"']
\end{tikzcd}
$$

Graph Coloring

- A c-coloring of a graph G is a map $C : V \rightarrow \{1, 2, \ldots, c\}$ that assigns one of c "colors" to each vertex so that every edge has two different colors at its endpoints
- The graph coloring problem is: "Does there exist a c-coloring for the graph G?"
- Even when $c = 3$, this problem is hard. We call this problem *3Colorable* i.e. "Does there exist a 3-coloring for the graph G?"

3Colorable

- To show that 3Colorable is NP-hard, we will reduce from 3Sat
- This means that we want to show that a polynomial time algorithm for 3Colorable can give a polynomial time algorithm for 3Sat
- Recall that the 3-SAT problem is just: "Is there any assignment of variables to a 3CNF formula that makes the formula evaluate to true?"
- And a 3CNF formula is just a conjunct of a bunch of clauses, each of which contains exactly 3 variables e.g.

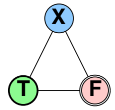
$$\overbrace{(a \vee b \vee c)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee d)$$

Reduction

- We are given a 3-CNF formula, F, and we must determine if it has a satisfying assignment
- To do this, we produce a graph as follows
- The graph contains one *truth* gadget, one *variable* gadget for each variable in the formula, and one *clause* gadget for each clause in the formula

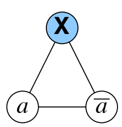
The Truth Gadget

- The truth gadget is just a triangle with three vertices T, F and X, which intuitively stand for **True**, **False**, and **other**
- Since these vertices are all connected, they must have different colors in any 3-coloring
- For the sake of convenience, we will name those colors **True**, **False**, and **Other**
- Thus when we say a node is colored "True", we just mean that it's colored the same color as the node T



The Variable Gadgets

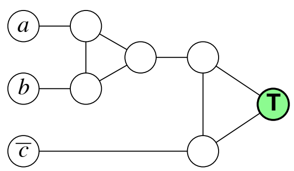
- The variable gadget for a variable a is also a triangle joining two new nodes labeled a and $\overline{a}$ to node X in the truth gadget
- Node a must be colored either "True" or "False", and so node $\overline{a}$ must be colored either "False" or "True", respectively.



- The variable gadget ensures that each of the literals is colored either "True" or "False"

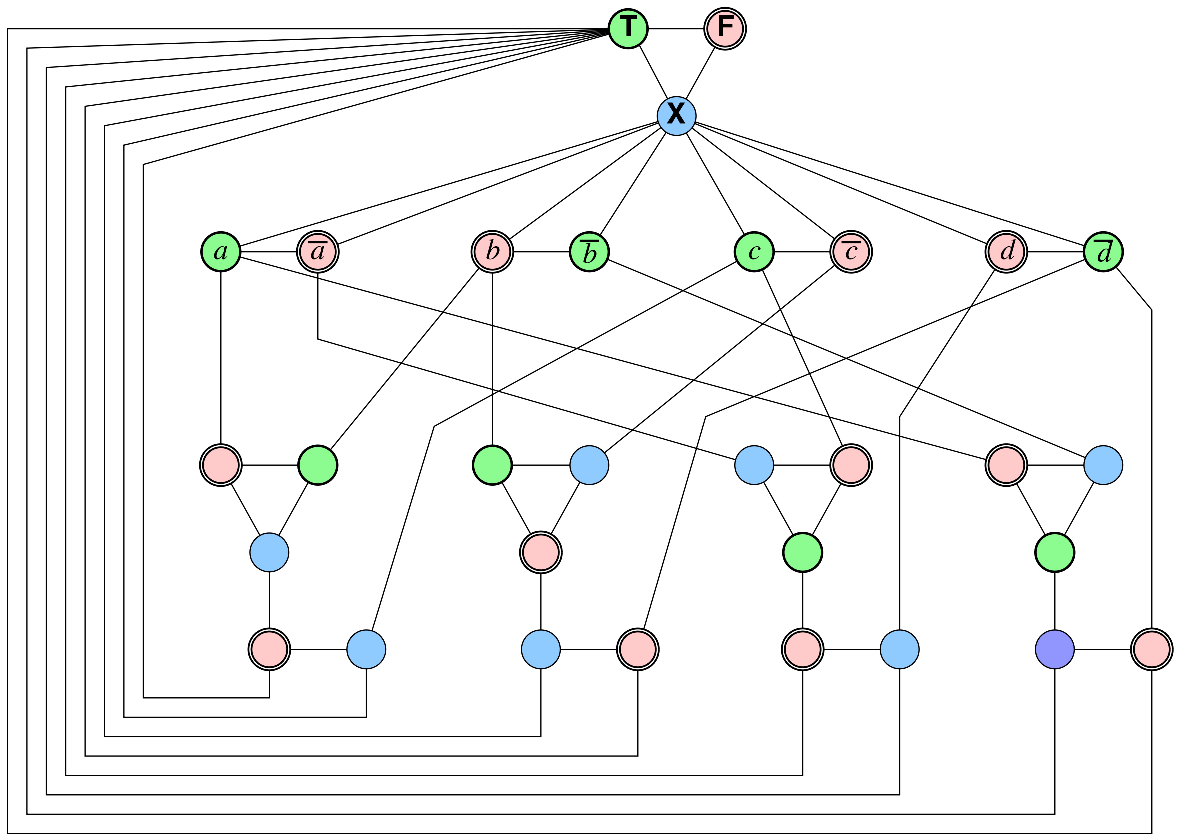
The Clause Gadgets

- Each clause gadget joins three literal nodes to node T in the truth gadget using five new unlabelled nodes and ten edges (as in the figure)
- This clause gadget ensures that at least one of the three literal nodes in each clause is colored "True"



Example

Consider the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$. Following is the graph created by the reduction:



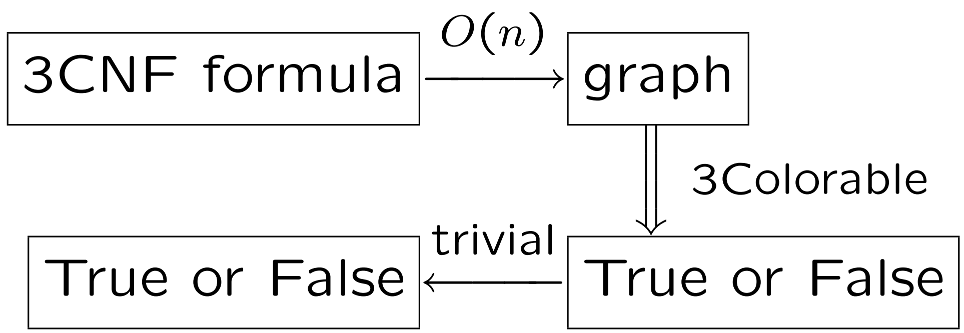
Example

- Note that the 3-coloring of this example graph corresponds to a satisfying assignment of the formula
- Namely, $a = c =$ True, $b = d =$ False.
- Note that the final graph contains only *one* node T, only *one* node F, only *one* node $\bar{a}$ for each variable a and so on

Correctness

- The proof of correctness for this reduction is direct
- If the graph is 3-colorable, then we can extract a satisfying assignment from any 3-coloring, since at least one of the three literal nodes in every clause gadget is colored "True"
- Conversely, if the formula is satisfiable, then we can color the graph according to any satisfying assignment

Reduction Picture



Wrap Up

- We've just shown that if 3Colorable can be solved in polynomial time then 3-SAT can be solved in polynomial time
- This shows that 3Colorable is NP-Hard
- To show that 3Colorable is in NP, we just need to note that we can easily verify that a graph has been correctly 3-colored in linear time: just compare the endpoints of every edge
- Thus, 3Coloring is NP-Complete.
- This implies that the more general graph coloring problem is also NP-Complete

In-Class Exercise

Consider the problem *4Colorable*: "Does there exist a 4-coloring for a graph G?"

- Q1: Show this problem is in NP by showing that there exists an efficiently verifiable proof of the fact that a graph is 4 colorable.
- Q2: Show the problem is NP-Hard by a reduction from the problem 3Colorable. In particular, show the following:
  - Given a graph G, you can create a graph G' such that G' is 4-colorable iff G is 3-colorable.
  - Creating G' from G takes polynomial time

Note: You've now shown that 4Colorable is NP-Complete!

Hamiltonian Cycle

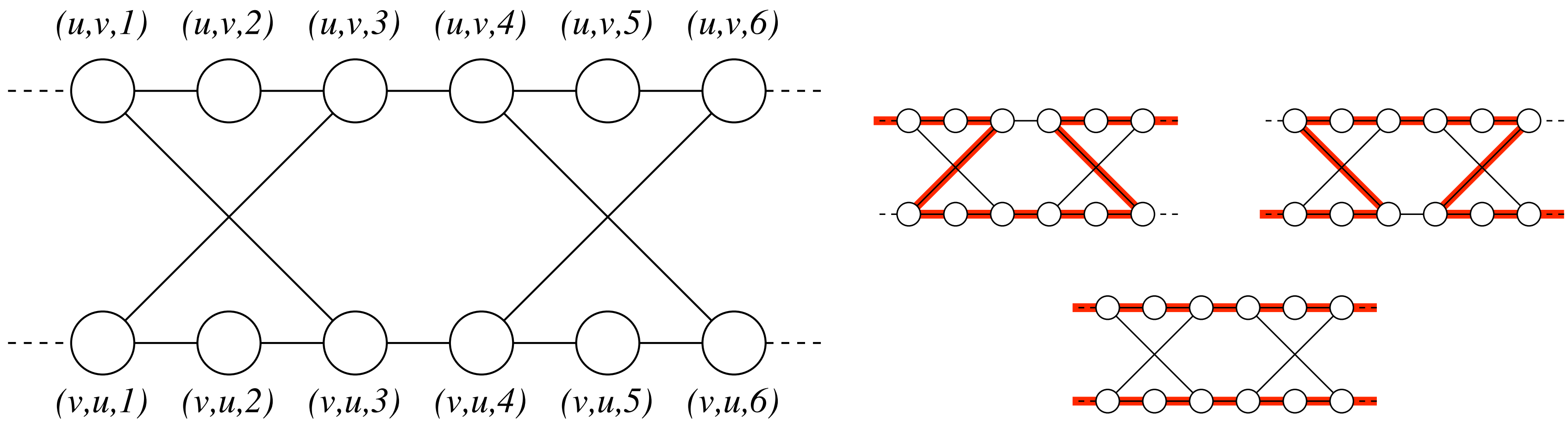
- A *Hamiltonian Cycle* in a graph is a cycle that visits every vertex exactly once (note that this is very different from an *Eulerian cycle* which visits every *edge* exactly once)
- The Hamiltonian Cycle problem is to determine if a given graph G has a Hamiltonian Cycle
- We will show that this problem is NP-Hard by a reduction from the vertex cover problem.

The Reduction

- To do the reduction, we need to show that we can solve Vertex Cover in polynomial time if we have a polynomial time solution to Hamiltonian Cycle.
- Given a graph G and an integer k, we will create another graph G' such that G' has a Hamiltonian cycle iff G has a vertex cover of size k
- As for the last reduction, our transformation will consist of putting together several "gadgets"

Edge Gadget and Cover Vertices

- For each edge (u, v) in G, we have an *edge gadget* in G' consisting of twelve vertices and fourteen edges, as shown below



An edge gadget for (u, v) and the only possible Hamiltonian paths through it.

Edge Gadget

- The four corner vertices $(u, v, 1)$, $(u, v, 6)$, $(v, u, 1)$, and $(v, u, 6)$ each have an edge leaving the gadget
- A Hamiltonian cycle can only pass through an edge gadget in one of the three ways shown in the figure
- These paths through the edge gadget will correspond to one or both of the vertices u and v being in the vertex cover.

Cover Vertices

- G' also contains k *cover vertices*, simply numbered 1 through k

Vertex Chains

- For each vertex u in G, we string together all the edge gadgets for edges (u, v) into a single *vertex chain* and then connect the ends of the chain to all the cover vertices
- Specifically, suppose u has d neighbors $v_1, v_2, \ldots, v_d$. Then G' has the following edges:
  - $d - 1$ edges between $(u, v_i, 6)$ and $(u, v_{i+1}, 1)$ (for all i between 1 and $d - 1$)
  - k edges between the cover vertices and $(u, v_1, 1)$
  - k edges between the cover vertices and $(u, v_d, 6)$

The Reduction

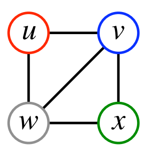
- It's not hard to prove that if $\{v_1, v_2, \ldots, v_k\}$ is a vertex cover of G, then G' has a Hamiltonian cycle
- To get this Hamiltonian cycle, we start at cover vertex 1, traverse through the vertex chain for v_1, then visit cover vertex 2, then traverse the vertex chain for v_2 and so forth, until we eventually return to cover vertex 1
- Conversely, one can prove that any Hamiltonian cycle in G' alternates between cover vertices and vertex chains, and that the vertex chains correspond to the k vertices in a vertex cover of G

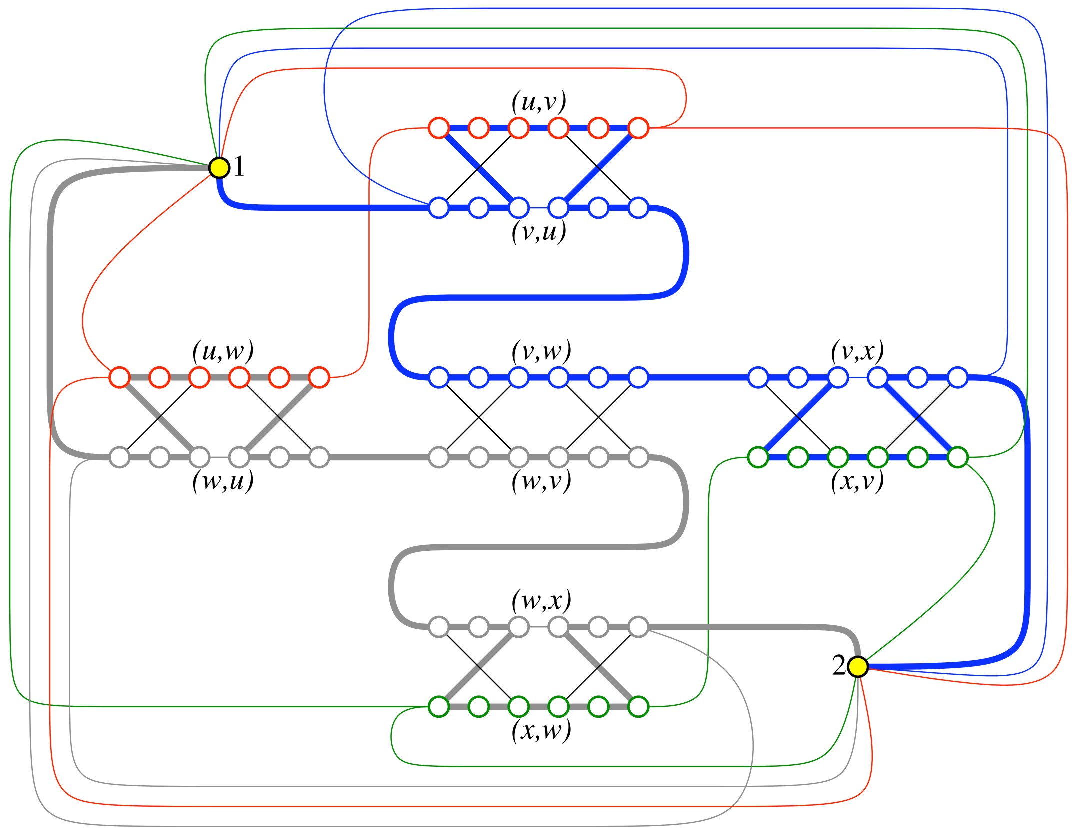
Thus, G has a vertex cover of size k iff G' has a Hamiltonian cycle

The Reduction

- The transformation from G to G' takes at most $O(|V|^2)$ time, so the Hamiltonian cycle problem is NP-Hard
- Moreover we can easily verify a Hamiltonian cycle in linear time, thus Hamiltonian cycle is also in NP
- Thus Hamiltonian Cycle is NP-Complete

Example





The original graph G with vertex cover $\{v, w\}$, and the transformed graph G' with a corresponding Hamiltonian cycle (bold edges).
Vertex chains are colored to match their corresponding vertices.

The Reduction

$$
\begin{tikzcd}
\boxed{\text{graph } G = (V, E),\ k} \arrow[r, "O(|V|^2)"] & \boxed{\text{graph } G'} \arrow[d, Rightarrow, "\text{Hamiltonian Cycle}"] \\
\boxed{\text{True or False}} & \boxed{\text{True or False}} \arrow[l, "O(1)"']
\end{tikzcd}
$$

Traveling Sales Person

- A problem closely related to Hamiltonian cycles is the famous *Traveling Salesperson Problem(TSP)*
- The TSP problem is: "Given a weighted graph G, find the shortest cycle that visits every vertex.
- Finding the shortest cycle is obviously harder than determining if a cycle exists at all, so since Hamiltonian Path is NP-hard, TSP is also NP-hard!

NP-Hard Games

- In 1999, Richard Kaye proved that the solitaire game Minesweeper is NP-Hard, using a reduction from Circuit Satifiability.
- Also in the last few years, Eric Demaine, et. al., proved that the game Tetris is NP-Hard

Challenge Problem

- Consider the *optimization* version of, say, the graph coloring problem: "Given a graph G, what is the smallest number of colors needed to color the graph?" (Note that unlike the *decision* version of this problem, this is not a yes/no question)
- Show that the optimization version of graph coloring is also NP-Hard by a reduction from the decision version of graph coloring.
- Is the optimization version of graph coloring also NP-Complete?

Challenge Problem

- Consider the problem 4Sat which is: "Is there any assignment of variables to a 4CNF formula that makes the formula evaluate to true?"
- Is this problem NP-Hard? If so, give a reduction from 3Sat that shows this. If not, give a polynomial time algorithm which solves it.

Challenge Problem

- Consider the following problem: "Does there exist a clique of size 5 in some input graph G?"
- Is this problem NP-Hard? If so, prove it by giving a reduction from some known NP-Hard problem. If not, give a polynomial time algorithm which solves it.

Vertex Cover

- A *vertex cover* of a graph is a set of vertices that touches every edge in the graph
- The decision version of *Vertex Cover* is: "Does there exist a vertex cover of size k in a graph G?".
- We've proven this problem is NP-Hard by an easy reduction from Independent Set
- The *optimization* version of *Vertex Cover* is: "What is the minimum size vertex cover of a graph G?"
- We can prove this problem is NP-Hard by a reduction from the decision version of Vertex Cover (left as an exercise).

Approximating Vertex Cover

- Even though the optimization version of Vertex Cover is NP-Hard, it's possible to *approximate* the answer efficiently
- In particular, in polynomial time, we can find a vertex cover which is no more than 2 times as large as the minimal vertex cover

Approximation Algorithm

- The approximation algorithm does the following until G has no more edges:
- It chooses an arbitrary edge (u, v) in G and includes both u and v in the cover
- It then removes from G all edges which are incident to either u or v

Approximation Algorithm

```
Approx-Vertex-Cover(G){
  C = {};
  E' = Edges of G;
  while(E' is not empty){
    let (u,v) be an arbitrary edge in E';
    add both u and v to C;
    remove from E' every edge incident to u or v;
  }
  return C;
}
```

Analysis

- If we implement the graph with adjacency lists, each edge need be touched at most once
- Hence the run time of the algorithm will be $O(|V| + |E|)$, which is polynomial time
- First, note that this algorithm does in fact return a vertex cover since it ensures that every edge in G is incident to some vertex in C
- Q: Is the vertex cover actually no more than twice the optimal size?

- Let A be the set of edges which are chosen in the first line of the while loop
- Note that no two edges of A share an endpoint
- Thus, *any* vertex cover must contain at least one endpoint of each edge in A
- Thus if $C*$ is an optimal cover then we can say that $|C*| \geq |A|$
- Further, we know that $|C| = 2|A|$
- This implies that $|C| \leq 2|C*|$

Which means that the vertex cover found by the algorithm is no more than twice the size of an optimal vertex cover.

TSP

- An optimization version of the TSP problem is: "Given a weighted graph G, what is the shortest Hamiltonian Cycle of G?"
- This problem is NP-Hard by a reduction from Hamiltonian Cycle
- However, there is a 2-approximation algorithm for this problem if the edge weights obey the *triangle inequality*

Triangle Inequality

- In many practical problems, it's reasonable to make the assumption that the weights, c, of the edges obey the *triangle inequality*
- The triangle inequality says that for all vertices $u, v, w \in V$:

$$c(u,w) \leq c(u,v) + c(v,w)$$

- In other words, the cheapest way to get from u to w is always to just take the edge (u,w)
- In the real world, this is usually a pretty natural assumption. For example it holds if the vertices are points in a plane and the cost of traveling between two vertices is just the euclidean distance between them.

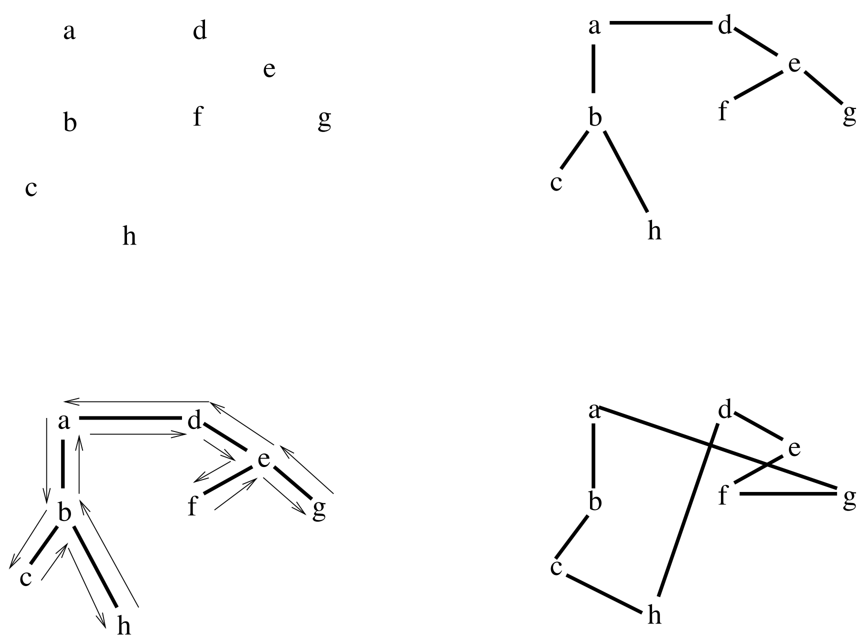
Approximation Algorithm

- Given a weighted graph G, the algorithm first computes a MST for G, T, and then arbitrarily selects a root node r of T.
- It then lets L be the list of the vertices visited in a depth first traversal of T starting at r.
- Finally, it returns the Hamiltonian Cycle, H, that visits the vertices in the order L.

Approximation Algorithm

```
Approx-TSP(G){
  T = MST(G);
  L = the list of vertices visited in a depth first traversal
      of T, starting at some arbitrary node in T;
  H = the Hamiltonian Cycle that visits the vertices in the
      order L;
  return H;
}
```

Example Run



The top left figure shows the graph G (edge weights are just the Euclidean distances between vertices); the top right figure shows the MST T. The bottom left figure shows the depth first walk on T, $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$; the bottom right figure shows the Hamiltonian cycle H obtained by deleting repeat visits from W, $H = (a, b, c, h, d, e, f, g)$.

Analysis

- The first step of the algorithm takes $O(|E| + |V| \log |V|)$ (if we use Prim's algorithm)
- The second step is $O(|V|)$
- The third step is $O(|V|)$.
- Hence the run time of the entire algorithm is polynomial

Analysis

An important fact about this algorithm is that: *the cost of the MST is less than the cost of the shortest Hamiltonian cycle.*

- To see this, let T be the MST and let $H*$ be the shortest Hamiltonian cycle.
- Note that if we remove one edge from $H*$, we have a spanning tree, T'
- Finally, note that $w(H*) \geq w(T') \geq w(T)$
- Hence $w(H*) \geq w(T)$

Analysis

- Now let W be a depth first walk of T which traverses each edge exactly twice (similar to what you did in the hw)
- In our example, $W = (a, b, c, b, h, b, a, d, e, f, e, g, e, d, a)$
- Note that $c(W) = 2c(T)$
- This implies that $c(W) \leq 2c(H*)$

Analysis

- Unfortunately, W is not a Hamiltonian cycle since it visits some vertices more than once
- However, we can delete a visit to any vertex and the cost will not increase *because of the triangle inequality*. (The path without an intermediate vertex can only be shorter)
- By repeatedly applying this operation, we can remove from W all but the first visit to each vertex, without increasing the cost of W.
- In our example, this will give us the ordering $H = (a, b, c, h, d, e, f, g)$

Analysis

- By the last slide, $c(H) \leq c(W)$.
- So $c(H) \leq c(W) = 2c(T) \leq 2c(H*)$
- Thus, $c(H) \leq 2c(H*)$
- In other words, the Hamiltonian cycle found by the algorithm has cost no more than twice the shortest Hamiltonian cycle.

Take Away

- Many real-world problems can be shown to not have an efficient solution unless $P = NP$ (these are the NP-Hard problems)
- However, if a problem is shown to be NP-Hard, all hope is not lost!
- In many cases, we can come up with an provably good approximation algorithm for the NP-Hard problem.