Recall the definition of polynomial time reductions:

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$. 
Recall the definition of polynomial time reductions:

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

- In general, algorithm for $Y$ can call the algorithm for $X$ many times.
Recall the definition of polynomial time reductions:

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

- In general, algorithm for $Y$ can call the algorithm for $X$ many times.
- However, for most reductions, we call algorithm for $X$ only once.
Recall the definition of polynomial time reductions:

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

- In general, algorithm for $Y$ can call the algorithm for $X$ many times.
- However, for most reductions, we call algorithm for $X$ only once.
- That is, for a given instance $s_Y$ for $Y$, we only construct one instance $s_X$ for $X$. 
Given an instance $s_Y$ of problem $Y$, show how to construct in polynomial time an instance $s_X$ of problem such that:

- $s_Y$ is a yes-instance of $Y \Rightarrow s_X$ is a yes-instance of $X$
- $s_X$ is a yes-instance of $X \Rightarrow s_Y$ is a yes-instance of $Y$
Q: How far away are we from proving or disproving $P = NP$?
Q: How far away are we from proving or disproving $P = NP$?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.
Q: How far away are we from proving or disproving P = NP?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
Q: How far away are we from proving or disproving $P = NP$?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n =$ number variables
Q: How far away are we from proving or disproving P = NP?

- Try to prove an “unconditional” lower bound on running time of an algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n =$ number variables
  - Best algorithm runs in time $O(c^n)$ for some constant $c > 1$
Q: How far away are we from proving or disproving $P = NP$?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n = \text{number variables}$
  - Best algorithm runs in time $O(c^n)$ for some constant $c > 1$
  - Best lower bound is $\Omega(n)$
Q: How far away are we from proving or disproving $P = NP$?

- Try to prove an “unconditional” lower bound on running time of algorithm solving a NP-complete problem.
- For 3-Sat problem:
  - Assume the number of clauses is $\Theta(n)$, $n =$ number variables
  - Best algorithm runs in time $O(c^n)$ for some constant $c > 1$
  - Best lower bound is $\Omega(n)$
- Essentially we have no techniques for proving lower bound for running time
Dealing with NP-Hard Problems

- Faster exponential time algorithms
- Solving the problem for special cases
- Fixed parameter tractability
- Approximation algorithms
Faster Exponential Time Algorithms

3-SAT:

Brute-force: \(O(2^n \cdot \text{poly}(n))\)

In practice: TSP Solver can solve Euclidean TSP instances with more than 100,000 vertices
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
Faster Exponential Time Algorithms

3-SAT:

- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
3-SAT:

- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:
- Brute-force: $O(n! \cdot \text{poly}(n))$
Faster Exponential Time Algorithms

3-SAT:
- Brute-force: \( O(2^n \cdot \text{poly}(n)) \)
- \( 2^n \rightarrow 1.844^n \rightarrow 1.3334^n \)
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:
- Brute-force: \( O(n! \cdot \text{poly}(n)) \)
- Better algorithm: \( O(2^n \cdot \text{poly}(n)) \)
3-SAT:
- Brute-force: $O(2^n \cdot \text{poly}(n))$
- $2^n \rightarrow 1.844^n \rightarrow 1.3334^n$
- Practical SAT Solver: solves real-world sat instances with more than 10,000 variables

Travelling Salesman Problem:
- Brute-force: $O(n! \cdot \text{poly}(n))$
- Better algorithm: $O(2^n \cdot \text{poly}(n))$
- In practice: TSP Solver can solve Euclidean TSP instances with more than 100,000 vertices
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

- trees (Quiz 10)
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

- trees (Quiz 10)
- bounded tree-width graphs
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

- trees (Quiz 10)
- bounded tree-width graphs
- interval graphs
Solving the problem for special cases

Maximum independent set problem is NP-hard on general graphs, but easy on

- trees (Quiz 10)
- bounded tree-width graphs
- interval graphs
- ...
Collaborative delivery problem (reduction from 3DM) is NP-hard on general graphs, but easy on
Collaborative delivery problem (reduction from 3DM) is NP-hard on general graphs, but easy on
- path (HW2 Problem 2)
Collaborative delivery problem (reduction from 3DM) is NP-hard on general graphs, but easy on
- path (HW2 Problem 2)
- trees
Solving the problem for special cases

Collaborative delivery problem (reduction from 3DM) is NP-hard on general graphs, but easy on

- path (HW2 Problem 2)
- trees
- . . .
Problem: whether there is a vertex cover of size $k$, for a small $k$ (number of nodes is $n$, number of edges is $\Theta(n)$.)
Fixed Parameter Tractability

- **Problem**: whether there is a vertex cover of size $k$, for a small $k$ (number of nodes is $n$, number of edges is $\Theta(n)$).
- **Brute-force algorithm**: $O(kn^{k+1})$
Problem: whether there is a vertex cover of size $k$, for a small $k$ (number of nodes is $n$, number of edges is $\Theta(n)$.)

Brute-force algorithm: $O(\kappa n^{k+1})$

Better running time: $O(2^k \cdot kn)$
Problem: whether there is a vertex cover of size $k$, for a small $k$ (number of nodes is $n$, number of edges is $\Theta(n)$.)

Brute-force algorithm: $O(kn^{k+1})$

Better running time: $O(2^k \cdot kn)$

Running time is $f(k)n^c$ for some $c$ independent of $k$
Fixed Parameter Tractability

- Problem: whether there is a vertex cover of size $k$, for a small $k$ (number of nodes is $n$, number of edges is $\Theta(n)$.)
- Brute-force algorithm: $O(kn^{k+1})$
- Better running time: $O(2^k \cdot kn)$
- Running time is $f(k)n^c$ for some $c$ independent of $k$
- Vertex-Cover is fixed-parameter tractable.
Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time.
Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in polynomial time.
- Approximation ratio is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution.
Approximation Algorithms

- For optimization problems, approximation algorithms will find sub-optimal solutions in \textit{polynomial time}.
- \textbf{Approximation ratio} is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution.
- We want to make the approximation ratio as small as possible, while maintaining the property that the algorithm runs in polynomial time.
For optimization problems, approximation algorithms will find sub-optimal solutions in \textit{polynomial time}.

**Approximation ratio** is the ratio between the quality of the solution output by the algorithm and the quality of the optimal solution.

We want to make the approximation ratio as small as possible, while maintaining the property that the algorithm runs in polynomial time.

There is an 2-approximation for the vertex cover problem: we can efficiently find a vertex cover whose size is at most 2 times that of the optimal vertex cover.
2-Approximation Algorithm for Vertex Cover

**Algorithm:** VertexCover\((G)\)

1. \(C \leftarrow \emptyset\)
2. **while** \(E \neq \emptyset\) **do**
3. select an edge \((u, v) \in E\), \(C \leftarrow C \cup \{u, v\}\)
4. Remove from \(E\) every edge incident on either \(u\) or \(v\)
5. **return** \(C\)

- Let the set \(C\) and \(C^*\) be the sets output by above algorithm and an optimal alg, respectively. Let \(S\) be the set of edges selected.
- Since no two edge in \(S\) are covered by the same vertex (Once an edge is picked in line 3, all other edges that are incident on its endpoints are removed from \(E\) in line 4), we have \(|C^*| \geq |S|\);
- As we have added both vertices of edge \((u, v)\), we get \(|C| = 2|S|\) but \(C^*\) have to add one of the two, thus, \(|C|/|C^*| \leq 2\).
Outline

1. Some Hard Problems
2. P, NP and Co-NP
3. Polynomial Time Reductions and NP-Completeness
4. NP-Complete Problems
5. Dealing with NP-Hard Problems
6. Summary
Summary

- We consider decision problems
- Inputs are encoded as \(\{0, 1\}\)-strings

**Def.** The complexity class \(P\) is the set of decision problems \(X\) that can be solved in polynomial time.

- Alice has a supercomputer, fast enough to run an exponential time algorithm
- Bob has a slow computer, which can only run a polynomial-time algorithm

**Def.** (Informal) The complexity class \(NP\) is the set of problems for which Alice can convince Bob a yes instance is a yes instance.
Def. $B$ is an \textbf{efficient certifier} for a problem $X$ if

- $B$ is a polynomial-time algorithm that takes two input strings $s$ and $t$
- there is a polynomial function $p$ such that, $X(s) = 1$ if and only if there is string $t$ such that $|t| \leq p(|s|)$ and $B(s, t) = 1$.

The string $t$ such that $B(s, t) = 1$ is called a \textbf{certificate}.

Def. The complexity class \textbf{NP} is the set of all problems for which there exists an efficient certifier.
**Summary**

**Def.** Given a black box algorithm $A$ that solves a problem $X$, if any instance of a problem $Y$ can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to $A$, then we say $Y$ is polynomial-time reducible to $X$, denoted as $Y \leq_P X$.

**Def.** A problem $X$ is called NP-complete if

1. $X \in \text{NP}$, and
2. $Y \leq_P X$ for every $Y \in \text{NP}$.

- If any NP-complete problem can be solved in polynomial time, then $P = NP$
- Unless $P = NP$, a NP-complete problem can not be solved in polynomial time
Summary

Proof of NP-Completeness for Circuit-Sat

- Fact 1: a polynomial-time algorithm can be converted to a polynomial-size circuit
- Fact 2: for a problem in NP, there is an efficient certifier.

Given a problem $X \in \text{NP}$, let $B(s, t)$ be the certifier

Convert $B(s, t)$ to a circuit and hard-wire $s$ to the input gates

$s$ is a yes-instance if and only if the resulting circuit is satisfiable

Proof of NP-Completeness for other problems by reductions