P, NP and Intractability

CS 4104: Data and Algorithm Analysis

Yoseph Berhanu Alebachew

May 11, 2025

Virginia Tech

Table of contents

Introduction

Recap: Efficient Algorithms

- Throughout this course, we adopted a definition of efficient algorithm to mean any algorithm with a polynomial running time.

Recap: Efficient Algorithms

- Throughout this course, we adopted a definition of efficient algorithm to mean any algorithm with a polynomial running time.
- We call all these class of algorithms P.

Recap: Efficient Algorithms

- Throughout this course, we adopted a definition of efficient algorithm to mean any algorithm with a polynomial running time.
- We call all these class of algorithms P.
- P refers to the fact that the algorithms have polynomial running time.

Recap: Efficient Algorithms

- Throughout this course, we adopted a definition of efficient algorithm to mean any algorithm with a polynomial running time.
- We call all these class of algorithms P.
- P refers to the fact that the algorithms have polynomial running time.
- This served as to identify problems with no known efficient solution

Recap: Efficient Algorithms

- Throughout this course, we adopted a definition of efficient algorithm to mean any algorithm with a polynomial running time.
- We call all these class of algorithms P.
- P refers to the fact that the algorithms have polynomial running time.
- This served as to identify problems with no known efficient solution
- However, we've seen that some modified versions of problems we discussed don't have efficient solutions.

Recap: Efficient Algorithms

- Throughout this course, we adopted a definition of efficient algorithm to mean any algorithm with a polynomial running time.
- We call all these class of algorithms P.
- P refers to the fact that the algorithms have polynomial running time.
- This served as to identify problems with no known efficient solution
- However, we've seen that some modified versions of problems we discussed don't have efficient solutions.
- Instead, the best known algorithms for these class of problems are exponential at best (if they exist).

Example: Shortest Path in a Graph

- **Original Problem**:
  - Find the shortest path from a source node to a destination node in a weighted graph.

Example: Shortest Path in a Graph

- **Original Problem**:
  - Find the shortest path from a source node to a destination node in a weighted graph.
  - **DP/Greedy Solution**: Algorithms like Dijkstra's algorithm or Bellman-Ford algorithm solve this problem in polynomial time.

Example: Shortest Path in a Graph

- **Original Problem**:
  - Find the shortest path from a source node to a destination node in a weighted graph.
  - **DP/Greedy Solution**: Algorithms like Dijkstra's algorithm or Bellman-Ford algorithm solve this problem in polynomial time.
- **Modified Problem**:
  - Introduce negative weight cycles in the graph, or constraints that paths must pass through specific intermediate nodes or avoid certain nodes altogether.

Example: Shortest Path in a Graph

- **Original Problem**:
  - Find the shortest path from a source node to a destination node in a weighted graph.
  - **DP/Greedy Solution**: Algorithms like Dijkstra's algorithm or Bellman-Ford algorithm solve this problem in polynomial time.
- **Modified Problem**:
  - Introduce negative weight cycles in the graph, or constraints that paths must pass through specific intermediate nodes or avoid certain nodes altogether.
  - Finding shortest paths in such modified graphs can make the problem NP-hard or undecidable.

Example: Subset Sum Problem

- **Original Problem**:
  - Determine if there is a subset of a given set of integers that sums up to a given target.

Example: Subset Sum Problem

- **Original Problem**:
  - Determine if there is a subset of a given set of integers that sums up to a given target.
  - **DP Solution**: The problem can be solved using dynamic programming in pseudo-polynomial time, specifically $O(n * target)$.

Example: Subset Sum Problem

- **Original Problem**:
  - Determine if there is a subset of a given set of integers that sums up to a given target.
  - **DP Solution**: The problem can be solved using dynamic programming in pseudo-polynomial time, specifically $O(n * target)$.
- **Modified Problem**:
  - Add additional constraints such as requiring certain subsets to be excluded from consideration or requiring that the subset elements satisfy additional arbitrary constraints.

Example: Subset Sum Problem

- **Original Problem**:
  - Determine if there is a subset of a given set of integers that sums up to a given target.
  - **DP Solution**: The problem can be solved using dynamic programming in pseudo-polynomial time, specifically $O(n * target)$.
- **Modified Problem**:
  - Add additional constraints such as requiring certain subsets to be excluded from consideration or requiring that the subset elements satisfy additional arbitrary constraints.
  - These modifications can make the problem NP-hard.

Example: Minimum Spanning Tree (MST)

- **Original Problem**:
  - Find a subset of edges in a weighted graph that connects all vertices with the minimum total edge weight.

Example: Minimum Spanning Tree (MST)

- **Original Problem**:
  - Find a subset of edges in a weighted graph that connects all vertices with the minimum total edge weight.
  - **Greedy Solution**: Algorithms like Kruskal's or Prim's algorithm solve this problem in polynomial time.

Example: Minimum Spanning Tree (MST)

- **Original Problem**:
  - Find a subset of edges in a weighted graph that connects all vertices with the minimum total edge weight.
  - **Greedy Solution**: Algorithms like Kruskal's or Prim's algorithm solve this problem in polynomial time.
- **Modified Problem**:
  - Add constraints that some edges must or must not be included, or introduce dependencies between edges, where selecting one edge requires or forbids selecting another.

Example: Minimum Spanning Tree (MST)

- **Original Problem**:
  - Find a subset of edges in a weighted graph that connects all vertices with the minimum total edge weight.
  - **Greedy Solution**: Algorithms like Kruskal's or Prim's algorithm solve this problem in polynomial time.
- **Modified Problem**:
  - Add constraints that some edges must or must not be included, or introduce dependencies between edges, where selecting one edge requires or forbids selecting another.
  - These modifications can turn the problem into an NP-hard problem.

Example: Traveling Salesman Problem (TSP)

- **Original Problem**:
  - Find the shortest possible route that visits each city exactly once and returns to the origin city.

Example: Traveling Salesman Problem (TSP)

- **Original Problem**:
  - Find the shortest possible route that visits each city exactly once and returns to the origin city.
  - **DP Solution**: The problem can be solved using dynamic programming with Held-Karp algorithm in $O(n^2 * 2^n)$ time, which is not polynomial but is the best known exact approach.

Example: Traveling Salesman Problem (TSP)

- **Original Problem**:
  - Find the shortest possible route that visits each city exactly once and returns to the origin city.
  - **DP Solution**: The problem can be solved using dynamic programming with Held-Karp algorithm in $O(n^2 * 2^n)$ time, which is not polynomial but is the best known exact approach.
- **Modified Problem**:
  - Introduce constraints such as visiting certain cities in a specific order or having variable costs that depend on the sequence of cities visited.

Example: Traveling Salesman Problem (TSP)

- **Original Problem**:
  - Find the shortest possible route that visits each city exactly once and returns to the origin city.
  - **DP Solution**: The problem can be solved using dynamic programming with Held-Karp algorithm in $O(n^2 * 2^n)$ time, which is not polynomial but is the best known exact approach.
- **Modified Problem**:
  - Introduce constraints such as visiting certain cities in a specific order or having variable costs that depend on the sequence of cities visited.
  - These constraints can turn the problem into an even more complex version that remains NP-hard and lacks a known polynomial-time solution.

Example: Graph Coloring

- **Original Problem**:
  - Assign colors to the vertices of a graph so that no two adjacent vertices share the same color using the fewest number of colors.

Example: Graph Coloring

- **Original Problem**:
  - Assign colors to the vertices of a graph so that no two adjacent vertices share the same color using the fewest number of colors.
  - **Greedy/DP Solution**: The problem can be approximated using greedy algorithms, such as the Welsh-Powell algorithm, which can provide a solution in polynomial time for certain types of graphs.

Example: Graph Coloring

- **Original Problem**:
  - Assign colors to the vertices of a graph so that no two adjacent vertices share the same color using the fewest number of colors.
  - **Greedy/DP Solution**: The problem can be approximated using greedy algorithms, such as the Welsh-Powell algorithm, which can provide a solution in polynomial time for certain types of graphs.
- **Modified Problem**:
  - Introduce constraints such as specific vertices needing to be a certain color or certain pairs of vertices needing to have different colors.

Example: Graph Coloring

- **Original Problem**:
  - Assign colors to the vertices of a graph so that no two adjacent vertices share the same color using the fewest number of colors.
  - **Greedy/DP Solution**: The problem can be approximated using greedy algorithms, such as the Welsh-Powell algorithm, which can provide a solution in polynomial time for certain types of graphs.
- **Modified Problem**:
  - Introduce constraints such as specific vertices needing to be a certain color or certain pairs of vertices needing to have different colors.
  - Add restrictions where the coloring must adhere to additional conditions, like distance constraints (vertices within a certain distance must also have different colors).

Example: Graph Coloring

- **Original Problem**:
  - Assign colors to the vertices of a graph so that no two adjacent vertices share the same color using the fewest number of colors.
  - **Greedy/DP Solution**: The problem can be approximated using greedy algorithms, such as the Welsh-Powell algorithm, which can provide a solution in polynomial time for certain types of graphs.
- **Modified Problem**:
  - Introduce constraints such as specific vertices needing to be a certain color or certain pairs of vertices needing to have different colors.
  - Add restrictions where the coloring must adhere to additional conditions, like distance constraints (vertices within a certain distance must also have different colors).
  - These modifications can turn the problem into an NP-hard problem, making it infeasible to solve in polynomial time.

Example: Observation

- **Fine Line Between Easy and Hard Problems**

Example: Observation

- **Fine Line Between Easy and Hard Problems**
  - Problems that have efficient solutions can be easily modified to have no known efficient solutions.

Example: Observation

- **Fine Line Between Easy and Hard Problems**
  - Problems that have efficient solutions can be easily modified to have no known efficient solutions.
  - The reverse is also true: hard problems can be solved efficiently if we impose certain constraints on the problem definition.

Example: Observation

- **Fine Line Between Easy and Hard Problems**
  - Problems that have efficient solutions can be easily modified to have no known efficient solutions.
  - The reverse is also true: hard problems can be solved efficiently if we impose certain constraints on the problem definition.
- **Practical Importance**

Example: Observation

- **Fine Line Between Easy and Hard Problems**
  - Problems that have efficient solutions can be easily modified to have no known efficient solutions.
  - The reverse is also true: hard problems can be solved efficiently if we impose certain constraints on the problem definition.
- **Practical Importance**
  - These problems are very practical in the real world, and having efficient solutions for them can make a significant impact.

Example: Observation

- **Fine Line Between Easy and Hard Problems**
  - Problems that have efficient solutions can be easily modified to have no known efficient solutions.
  - The reverse is also true: hard problems can be solved efficiently if we impose certain constraints on the problem definition.
- **Practical Importance**
  - These problems are very practical in the real world, and having efficient solutions for them can make a significant impact.
  - Many of these problems are interconnected.

Example: Observation

- **Fine Line Between Easy and Hard Problems**
  - Problems that have efficient solutions can be easily modified to have no known efficient solutions.
  - The reverse is also true: hard problems can be solved efficiently if we impose certain constraints on the problem definition.
- **Practical Importance**
  - These problems are very practical in the real world, and having efficient solutions for them can make a significant impact.
  - Many of these problems are interconnected.
    - Solving one hard problem could potentially lead to solutions for other related problems.

Example: Observation

- **Fine Line Between Easy and Hard Problems**
  - Problems that have efficient solutions can be easily modified to have no known efficient solutions.
  - The reverse is also true: hard problems can be solved efficiently if we impose certain constraints on the problem definition.
- **Practical Importance**
  - These problems are very practical in the real world, and having efficient solutions for them can make a significant impact.
  - Many of these problems are interconnected.
    - Solving one hard problem could potentially lead to solutions for other related problems.
- **Role of Heuristics and Approximation Algorithms**

Example: Observation

- **Fine Line Between Easy and Hard Problems**
  - Problems that have efficient solutions can be easily modified to have no known efficient solutions.
  - The reverse is also true: hard problems can be solved efficiently if we impose certain constraints on the problem definition.
- **Practical Importance**
  - These problems are very practical in the real world, and having efficient solutions for them can make a significant impact.
  - Many of these problems are interconnected.
    - Solving one hard problem could potentially lead to solutions for other related problems.
- **Role of Heuristics and Approximation Algorithms**
  - For problems that are difficult or impossible to solve exactly in polynomial time, heuristics and approximation algorithms can provide near-optimal solutions in a reasonable amount of time.

Example: Observation

- **Fine Line Between Easy and Hard Problems**
  - Problems that have efficient solutions can be easily modified to have no known efficient solutions.
  - The reverse is also true: hard problems can be solved efficiently if we impose certain constraints on the problem definition.
- **Practical Importance**
  - These problems are very practical in the real world, and having efficient solutions for them can make a significant impact.
  - Many of these problems are interconnected.
    - Solving one hard problem could potentially lead to solutions for other related problems.
- **Role of Heuristics and Approximation Algorithms**
  - For problems that are difficult or impossible to solve exactly in polynomial time, heuristics and approximation algorithms can provide near-optimal solutions in a reasonable amount of time.
  - Understanding when and how to use these techniques is crucial for tackling real-world problems that are NP-hard.

The Clique Problem

The Clique Problem

- Given an undirected graph, determine if there exists a clique of size k.
- A clique is a subset of vertices such that every two vertices in the subset are connected by an edge.
- Given an undirected graph G, the problem can take different forms
    - Is the graph G a clique ?

The Clique Problem

- Given an undirected graph, determine if there exists a clique of size k.
- A clique is a subset of vertices such that every two vertices in the subset are connected by an edge.
- Given an undirected graph G, the problem can take different forms
    - Is the graph G a clique ?
        - A polynomial solution exists - $O(|V|^2)$

The Clique Problem

- Given an undirected graph, determine if there exists a clique of size k.
- A clique is a subset of vertices such that every two vertices in the subset are connected by an edge.
- Given an undirected graph G, the problem can take different forms
  - Is the graph G a clique ?
    - A polynomial solution exists - $O(|V|^2)$
  - Is there a clique within G ?

The Clique Problem

- Given an undirected graph, determine if there exists a clique of size k.
- A clique is a subset of vertices such that every two vertices in the subset are connected by an edge.
- Given an undirected graph G, the problem can take different forms
  - Is the graph G a clique ?
    - A polynomial solution exists - $O(|V|^2)$
  - Is there a clique within G ?
  - Is there a clique within G of size k?

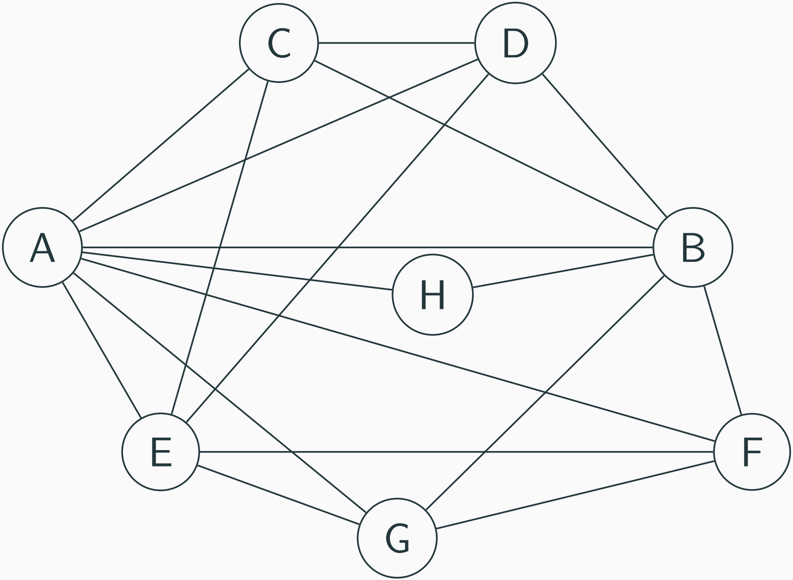
The Clique Problem

- Given an undirected graph, determine if there exists a clique of size k.
- A clique is a subset of vertices such that every two vertices in the subset are connected by an edge.
- Given an undirected graph G, the problem can take different forms
  - Is the graph G a clique ?
    - A polynomial solution exists - $O(|V|^2)$
  - Is there a clique within G ?
  - Is there a clique within G of size k?
    - Which nodes and edges make such a clique ?

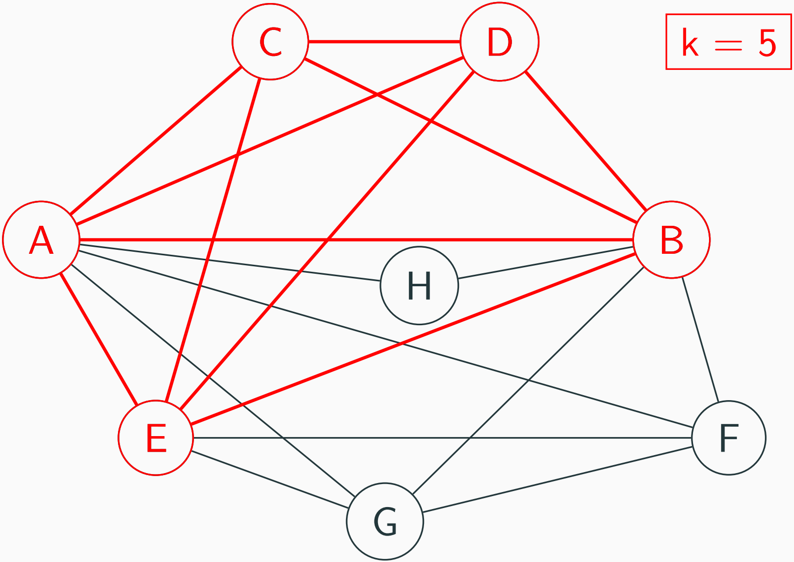
The Clique Problem

- Given an undirected graph, determine if there exists a clique of size k.
- A clique is a subset of vertices such that every two vertices in the subset are connected by an edge.
- Given an undirected graph G, the problem can take different forms
    - Is the graph G a clique ?
        - A polynomial solution exists - $O(|V|^2)$
    - Is there a clique within G ?
    - Is there a clique within G of size k?
        - Which nodes and edges make such a clique ?
- No known polynomial-time solution exists for the last three
- **Brute Force Solution**: Check all possible subsets of vertices of size k to see if they form a clique, which takes exponential time.

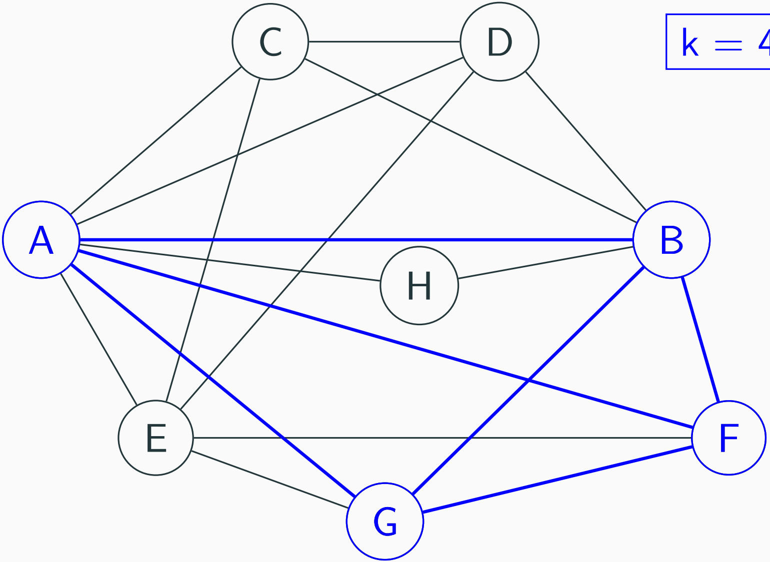
The Clique Problem: Example



The Clique Problem: Example

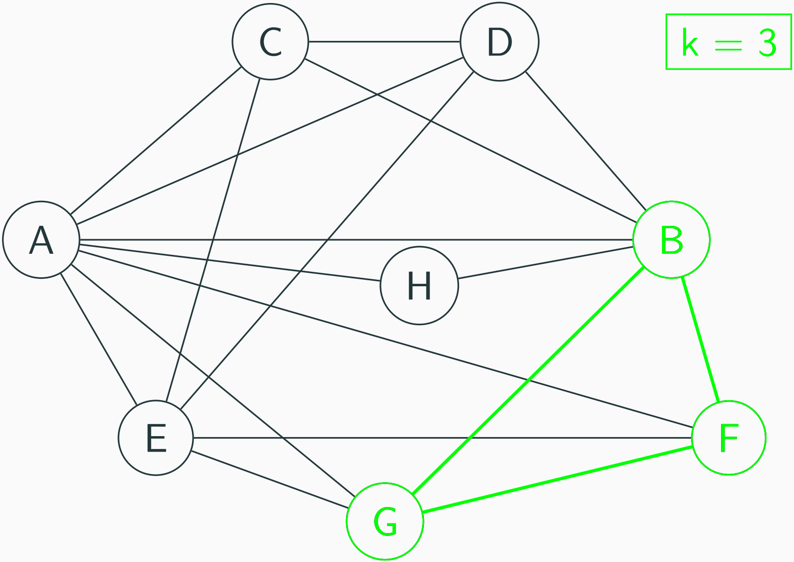


The Clique Problem: Example



$\boxed{k = 4}$

The Clique Problem: Example



The Clique Problem: NP

- No known polynomial-time solution exists for the last three
- **Brute Force Solution**: Check all possible subsets of vertices of size k to see if they form a clique, which takes exponential time.

The Clique Problem: NP

- No known polynomial-time solution exists for the last three
- **Brute Force Solution**: Check all possible subsets of vertices of size k to see if they form a clique, which takes exponential time.
- Since verifying a proposed clique takes polynomial time the clique problem is NP

Definitions

Definitions: Certificate

- A certificate is a solution to a problem that can be verified in polynomial time.

Definitions: Certificate

- A certificate is a solution to a problem that can be verified in polynomial time.
- For NP problems, given a "yes" answer, a certificate exists that can be checked quickly to confirm the answer is correct.

Definitions: Certificate

- A certificate is a solution to a problem that can be verified in polynomial time.
- For NP problems, given a "yes" answer, a certificate exists that can be checked quickly to confirm the answer is correct.
  - Example: For the Hamiltonian Path problem, a certificate is a specific sequence of vertices that forms a Hamiltonian Path.
- For Co-NP problems, given a "no" answer, a certificate exists that can be checked quickly to confirm the answer is correct.
  - Example: For the Composite Number problem, a Co-NP problem, a certificate for a "no" answer (i.e., the number is prime) is a demonstration that there are no factors other than 1 and the number itself.

P and NP

- A decision yes/no problem of input size n can be solved in $O(n^c)$ time for any constant c, the problem is in the complexity class P (Polynomial)

P and NP

- A decision yes/no problem of input size n can be solved in $O(n^c)$ time for any constant c, the problem is in the complexity class P (Polynomial)
  - E.g., Is the graph G a clique ?

P and NP

- A decision yes/no problem of input size n can be solved in $O(n^c)$ time for any constant c, the problem is in the complexity class P (Polynomial)
  - E.g., Is the graph G a clique ?
- If, for all yes instances of a decision problem of input size , there exists a **certificate**, that can be **verified** in polynomial time, we say it is in class NP (Nondeterministic Polynomial)

P and NP

- A decision yes/no problem of input size n can be solved in $O(n^c)$ time for any constant c, the problem is in the complexity class P (Polynomial)
    - E.g., Is the graph G a clique ?
- If, for all yes instances of a decision problem of input size , there exists a **certificate**, that can be **verified** in polynomial time, we say it is in class NP (Nondeterministic Polynomial)
    - E.g., Is there a clique within G ?

P and NP

- A decision yes/no problem of input size n can be solved in $O(n^c)$ time for any constant c, the problem is in the complexity class P (Polynomial)
  - E.g., Is the graph G a clique ?
- If, for all yes instances of a decision problem of input size , there exists a **certificate**, that can be **verified** in polynomial time, we say it is in class NP (Nondeterministic Polynomial)
  - E.g., Is there a clique within G ?
    - If we find some magical procedure to propose a sub graph we can verify if it is a clique in polynomial time

P and NP

- A decision yes/no problem of input size n can be solved in $O(n^c)$ time for any constant c, the problem is in the complexity class P (Polynomial)
    - E.g., Is the graph G a clique ?
- If, for all yes instances of a decision problem of input size , there exists a **certificate**, that can be **verified** in polynomial time, we say it is in class NP (Nondeterministic Polynomial)
    - E.g., Is there a clique within G ?
        - If we find some magical procedure to propose a sub graph we can verify if it is a clique in polynomial time
    - These class of problems are *yes heavy* - we can check yes cases in polynomial time - not the no cases

NP: General Outline

```
proc solve(input):
    // This step is not well defined, hence, non-deterministic
    // If this step is done in polynomial time
    //    then we have a polynomial solution
    certificates = generateAllPossibleSolutions(input)

    // Verify the given solutions for the problem
    // This should run in polynomial time
    for certificate in certificates:
        if verifySolution(input, certificate):
            return certificate
    return "No solution found"
```

NP vs Co-NP

- If you a have decision problem where negative instances are verified in polynomial time but you don't need to verify yes instances we say they are **Co-NP** (Complement is in NP)

NP vs Co-NP

- If you a have decision problem where negative instances are verified in polynomial time but you don't need to verify yes instances we say they are **Co-NP** (Complement is in NP)
- $P \subseteq NP$

NP vs Co-NP

- If you a have decision problem where negative instances are verified in polynomial time but you don't need to verify yes instances we say they are **Co-NP** (Complement is in NP)
- $P \subseteq NP$
- $P \subseteq Co - NP$

NP vs Co-NP

- If you a have decision problem where negative instances are verified in polynomial time but you don't need to verify yes instances we say they are **Co-NP** (Complement is in NP)
- $P \subseteq NP$
- $P \subseteq Co - NP$
    - For any instance of a problem in P, an empty certificate is sufficient

NP vs Co-NP

- If you a have decision problem where negative instances are verified in polynomial time but you don't need to verify yes instances we say they are **Co-NP** (Complement is in NP)
- $P \subseteq NP$
- $P \subseteq Co - NP$
  - For any instance of a problem in P, an empty certificate is sufficient
  - We can verify that it is yes or no in polynomial time

NP vs Co-NP

- If you a have decision problem where negative instances are verified in polynomial time but you don't need to verify yes instances we say they are **Co-NP** (Complement is in NP)
- $P \subseteq NP$
- $P \subseteq Co - NP$
  - For any instance of a problem in P, an empty certificate is sufficient
  - We can verify that it is yes or no in polynomial time
- $x \in NP$ is a statement of ease, not a statement of hardness

NP vs Co-NP

- If you a have decision problem where negative instances are verified in polynomial time but you don't need to verify yes instances we say they are **Co-NP** (Complement is in NP)
- $P \subseteq NP$
- $P \subseteq Co - NP$
  - For any instance of a problem in P, an empty certificate is sufficient
  - We can verify that it is yes or no in polynomial time
- $x \in NP$ is a statement of ease, not a statement of hardness
- NP is an upper limit not a lower limit

Definitions: P (Polynomial Time)

- **P (Polynomial Time)**
  - P is a class of problems that can be solved by an algorithm in polynomial time.
  - Polynomial time means that the time complexity of the algorithm is $O(n^k)$ for some constant k, where n is the size of the input.
  - Examples: Sorting algorithms like Merge Sort and Quick Sort, and searching algorithms like Binary Search.

Definitions: P (Polynomial Time)

- **P (Polynomial Time)**
  - P is a class of problems that can be solved by an algorithm in polynomial time.
  - Polynomial time means that the time complexity of the algorithm is $O(n^k)$ for some constant k, where n is the size of the input.
  - Examples: Sorting algorithms like Merge Sort and Quick Sort, and searching algorithms like Binary Search.
- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.

Definitions: P (Polynomial Time)

- **P (Polynomial Time)**
  - P is a class of problems that can be solved by an algorithm in polynomial time.
  - Polynomial time means that the time complexity of the algorithm is $O(n^k)$ for some constant k, where n is the size of the input.
  - Examples: Sorting algorithms like Merge Sort and Quick Sort, and searching algorithms like Binary Search.
- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.
  - A problem is in NP if, for every instance where the answer is "yes," there is a certificate (or witness) that can be checked quickly (in polynomial time) to confirm the answer.

Definitions: P (Polynomial Time)

- **P (Polynomial Time)**
  - P is a class of problems that can be solved by an algorithm in polynomial time.
  - Polynomial time means that the time complexity of the algorithm is $O(n^k)$ for some constant k, where n is the size of the input.
  - Examples: Sorting algorithms like Merge Sort and Quick Sort, and searching algorithms like Binary Search.
- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.
  - A problem is in NP if, for every instance where the answer is "yes," there is a certificate (or witness) that can be checked quickly (in polynomial time) to confirm the answer.
  - Examples: Clique Problem, Satisfiability (SAT), Hamiltonian Path, Subset Sum.

Definitions: Reduction

- **Reduction** is a way of converting one problem to another problem.

Definitions: Reduction

- **Reduction** is a way of converting one problem to another problem.
- A problem A can be reduced to problem B if an algorithm for solving B efficiently (in polynomial time) can be used to solve A efficiently.

Definitions: Reduction

- **Reduction** is a way of converting one problem to another problem.
- A problem A can be reduced to problem B if an algorithm for solving B efficiently (in polynomial time) can be used to solve A efficiently.
- Reductions are used to prove the hardness of problems.
    - Examples: Reducing 3-SAT to the Hamiltonian Cycle problem to prove Hamiltonian Cycle is NP-hard.
        - 3-SAT $<_p$ Hamiltonian Cycle

Definitions: Reduction

- **Reduction** is a way of converting one problem to another problem.
- A problem A can be reduced to problem B if an algorithm for solving B efficiently (in polynomial time) can be used to solve A efficiently.
- Reductions are used to prove the hardness of problems.
  - Examples: Reducing 3-SAT to the Hamiltonian Cycle problem to prove Hamiltonian Cycle is NP-hard.
    - 3-SAT $<_p$ Hamiltonian Cycle
- A polynomial-time reduction is a process of transforming one problem into another in polynomial time.

Definitions: Reduction

- **Reduction** is a way of converting one problem to another problem.
- A problem A can be reduced to problem B if an algorithm for solving B efficiently (in polynomial time) can be used to solve A efficiently.
- Reductions are used to prove the hardness of problems.
  - Examples: Reducing 3-SAT to the Hamiltonian Cycle problem to prove Hamiltonian Cycle is NP-hard.
    - 3-SAT $<_p$ Hamiltonian Cycle
- A polynomial-time reduction is a process of transforming one problem into another in polynomial time.
- Polynomial-time reductions are used to show that a problem is NP-hard.

Definitions: Reduction

- **Reduction** is a way of converting one problem to another problem.
- A problem A can be reduced to problem B if an algorithm for solving B efficiently (in polynomial time) can be used to solve A efficiently.
- Reductions are used to prove the hardness of problems.
  - Examples: Reducing 3-SAT to the Hamiltonian Cycle problem to prove Hamiltonian Cycle is NP-hard.
    - 3-SAT $<_p$ Hamiltonian Cycle
- A polynomial-time reduction is a process of transforming one problem into another in polynomial time.
- Polynomial-time reductions are used to show that a problem is NP-hard.
- Reductions are transitive but not symmetric.

Definitions: Reduction

- **Reduction** is a way of converting one problem to another problem.
- A problem A can be reduced to problem B if an algorithm for solving B efficiently (in polynomial time) can be used to solve A efficiently.
- Reductions are used to prove the hardness of problems.
    - Examples: Reducing 3-SAT to the Hamiltonian Cycle problem to prove Hamiltonian Cycle is NP-hard.
        - 3-SAT $<_p$ Hamiltonian Cycle
- A polynomial-time reduction is a process of transforming one problem into another in polynomial time.
- Polynomial-time reductions are used to show that a problem is NP-hard.
- Reductions are transitive but not symmetric.
    - If $X <_p Y$ and $Y <_p Z$, then $X <_p Z$.

Definitions: Reduction

- **Reduction** is a way of converting one problem to another problem.
- A problem A can be reduced to problem B if an algorithm for solving B efficiently (in polynomial time) can be used to solve A efficiently.
- Reductions are used to prove the hardness of problems.
    - Examples: Reducing 3-SAT to the Hamiltonian Cycle problem to prove Hamiltonian Cycle is NP-hard.
        - 3-SAT $<_p$ Hamiltonian Cycle
- A polynomial-time reduction is a process of transforming one problem into another in polynomial time.
- Polynomial-time reductions are used to show that a problem is NP-hard.
- Reductions are transitive but not symmetric.
    - If $X <_p Y$ and $Y <_p Z$, then $X <_p Z$.
    - $X <_p Y \not\Rightarrow Y <_p X$

Definitions: NP-hard

- A problem is NP-hard if **every problem** in NP can be reduced to it in polynomial time.

Definitions: NP-hard

- A problem is NP-hard if **every problem** in NP can be reduced to it in polynomial time.
  - In practice we only need to show reduction to **one** well known NP hard problem as reduction is **transitive**

Definitions: NP-hard

- A problem is NP-hard if **every problem** in NP can be reduced to it in polynomial time.
  - In practice we only need to show reduction to **one** well known NP hard problem as reduction is **transitive**
- NP-hard problems are at least as hard as the hardest problems in NP.

Definitions: NP-hard

- A problem is NP-hard if **every problem** in NP can be reduced to it in polynomial time.
  - In practice we only need to show reduction to **one** well known NP hard problem as reduction is **transitive**
- NP-hard problems are at least as hard as the hardest problems in NP.
- Solving an NP-hard problem efficiently (in polynomial time) would imply that $P = NP$.

Definitions: NP-hard

- A problem is NP-hard if **every problem** in NP can be reduced to it in polynomial time.
  - In practice we only need to show reduction to **one** well known NP hard problem as reduction is **transitive**
- NP-hard problems are at least as hard as the hardest problems in NP.
- Solving an NP-hard problem efficiently (in polynomial time) would imply that $P = NP$.
- NP-hard problems do not have to be in NP; they might not even have solutions that can be verified in polynomial time.

Definitions: NP-hard

- A problem is NP-hard if **every problem** in NP can be reduced to it in polynomial time.
  - In practice we only need to show reduction to **one** well known NP hard problem as reduction is **transitive**
- NP-hard problems are at least as hard as the hardest problems in NP.
- Solving an NP-hard problem efficiently (in polynomial time) would imply that $P = NP$.
- NP-hard problems do not have to be in NP; they might not even have solutions that can be verified in polynomial time.
- Examples: Halting Problem, Traveling Salesman Problem (general case), 3-SAT.

Definitions: NP-hard

- A problem is NP-hard if **every problem** in NP can be reduced to it in polynomial time.
  - In practice we only need to show reduction to **one** well known NP hard problem as reduction is **transitive**
- NP-hard problems are at least as hard as the hardest problems in NP.
- Solving an NP-hard problem efficiently (in polynomial time) would imply that $P = NP$.
- NP-hard problems do not have to be in NP; they might not even have solutions that can be verified in polynomial time.
- Examples: Halting Problem, Traveling Salesman Problem (general case), 3-SAT.
- $x \in NP - Hard$ is a statement of hardness (a lower limit)

NP vs NP-hard

- **NP (Nondeterministic Polynomial Time)**

NP vs NP-hard

- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.

NP vs NP-hard

- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.
  - If a problem is in NP, it means that, given a "yes" answer, there is a certificate (or witness) that can be checked quickly (in polynomial time) to confirm the answer is correct.

NP vs NP-hard

- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.
  - If a problem is in NP, it means that, given a "yes" answer, there is a certificate (or witness) that can be checked quickly (in polynomial time) to confirm the answer is correct.
  - Examples: SAT (Satisfiability), Hamiltonian Path, Subset Sum.

NP vs NP-hard

- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.
  - If a problem is in NP, it means that, given a "yes" answer, there is a certificate (or witness) that can be checked quickly (in polynomial time) to confirm the answer is correct.
  - Examples: SAT (Satisfiability), Hamiltonian Path, Subset Sum.
- **NP-hard**

NP vs NP-hard

- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.
  - If a problem is in NP, it means that, given a "yes" answer, there is a certificate (or witness) that can be checked quickly (in polynomial time) to confirm the answer is correct.
  - Examples: SAT (Satisfiability), Hamiltonian Path, Subset Sum.
- **NP-hard**
  - A problem is NP-hard if every problem in NP can be reduced to it in polynomial time.

NP vs NP-hard

- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.
  - If a problem is in NP, it means that, given a "yes" answer, there is a certificate (or witness) that can be checked quickly (in polynomial time) to confirm the answer is correct.
  - Examples: SAT (Satisfiability), Hamiltonian Path, Subset Sum.
- **NP-hard**
  - A problem is NP-hard if every problem in NP can be reduced to it in polynomial time.
  - NP-hard problems are at least as hard as the hardest problems in NP.

NP vs NP-hard

- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.
  - If a problem is in NP, it means that, given a "yes" answer, there is a certificate (or witness) that can be checked quickly (in polynomial time) to confirm the answer is correct.
  - Examples: SAT (Satisfiability), Hamiltonian Path, Subset Sum.
- **NP-hard**
  - A problem is NP-hard if every problem in NP can be reduced to it in polynomial time.
  - NP-hard problems are at least as hard as the hardest problems in NP.
  - Solving an NP-hard problem efficiently (in polynomial time) would imply that P = NP.

NP vs NP-hard

- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.
  - If a problem is in NP, it means that, given a "yes" answer, there is a certificate (or witness) that can be checked quickly (in polynomial time) to confirm the answer is correct.
  - Examples: SAT (Satisfiability), Hamiltonian Path, Subset Sum.
- **NP-hard**
  - A problem is NP-hard if every problem in NP can be reduced to it in polynomial time.
  - NP-hard problems are at least as hard as the hardest problems in NP.
  - Solving an NP-hard problem efficiently (in polynomial time) would imply that P = NP.
  - NP-hard problems do not have to be in NP; they might not even have solutions that can be verified in polynomial time.

NP vs NP-hard

- **NP (Nondeterministic Polynomial Time)**
  - NP is a class of problems for which a given solution can be verified in polynomial time.
  - If a problem is in NP, it means that, given a "yes" answer, there is a certificate (or witness) that can be checked quickly (in polynomial time) to confirm the answer is correct.
  - Examples: SAT (Satisfiability), Hamiltonian Path, Subset Sum.
- **NP-hard**
  - A problem is NP-hard if every problem in NP can be reduced to it in polynomial time.
  - NP-hard problems are at least as hard as the hardest problems in NP.
  - Solving an NP-hard problem efficiently (in polynomial time) would imply that P = NP.
  - NP-hard problems do not have to be in NP; they might not even have solutions that can be verified in polynomial time.
  - Examples: Halting Problem, Traveling Salesman Problem (general case), 3-SAT.

Definitions: NP-Complete

- A problem is said to be NP-Complete if it is in both NP and NP-hard
- It is NP in that we can verify a certificate in polynomial time
- It is NP-hard, meaning it we can reduce all NP problems to it
  - Remember, in practice we only need to reduce one known NP-hard problem
- Problems in NP-complete have the property that they are difficult but if one is solved efficiently, all will be solved automatically

Definitions: NP-complete

- **NP-complete**

Definitions: NP-complete

- **NP-complete**
  - NP-complete problems are a subset of NP problems that are both in NP and NP-hard.

Definitions: NP-complete

- **NP-complete**
  - NP-complete problems are a subset of NP problems that are both in NP and NP-hard.
  - A problem is NP-complete if it is in NP and as hard as any problem in NP, meaning every problem in NP can be reduced to it in polynomial time.

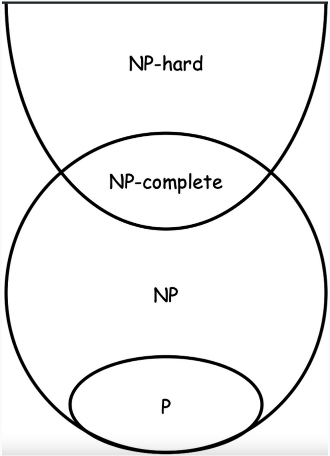
Definitions: NP-complete

- **NP-complete**
  - NP-complete problems are a subset of NP problems that are both in NP and NP-hard.
  - A problem is NP-complete if it is in NP and as hard as any problem in NP, meaning every problem in NP can be reduced to it in polynomial time.
  - If any NP-complete problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time ($\mathrm{P} = \mathrm{NP}$).

Definitions: NP-complete

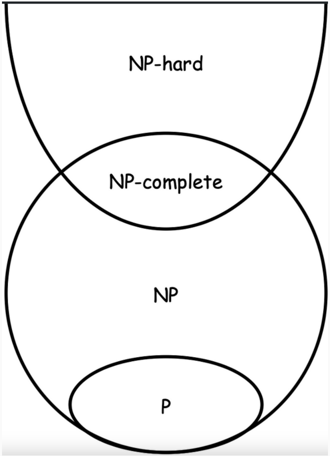
- **NP-complete**
  - NP-complete problems are a subset of NP problems that are both in NP and NP-hard.
  - A problem is NP-complete if it is in NP and as hard as any problem in NP, meaning every problem in NP can be reduced to it in polynomial time.
  - If any NP-complete problem can be solved in polynomial time, then all problems in NP can be solved in polynomial time ($P = NP$).
  - Examples: Satisfiability (SAT), Traveling Salesman Problem (decision version), 3-SAT.

NP vs NP-hard vs NP Complete



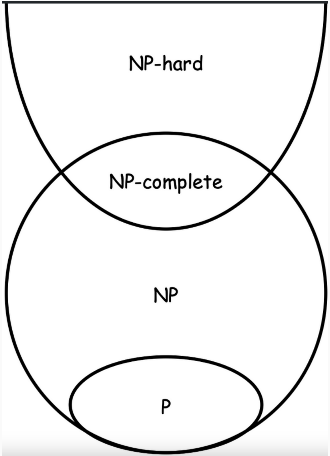
- NP-hard problems encompass both problems in NP and problems that are even harder.

NP vs NP-hard vs NP Complete



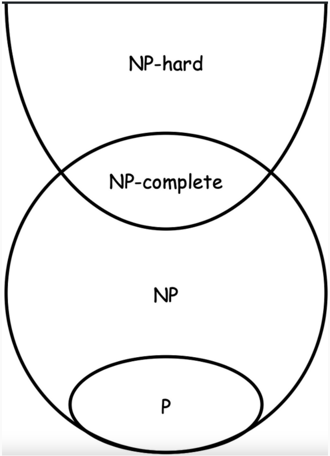
- NP-hard problems encompass both problems in NP and problems that are even harder.
- All NP-complete problems are both in NP and NP-hard.

NP vs NP-hard vs NP Complete



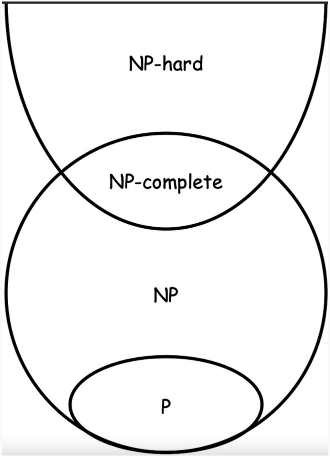
- NP-hard problems encompass both problems in NP and problems that are even harder.
- All NP-complete problems are both in NP and NP-hard.
- If any NP-complete problem can be solved in polynomial time, then all NP problems can be solved in polynomial time (P = NP).

NP vs NP-hard vs NP Complete



- NP-hard problems encompass both problems in NP and problems that are even harder.
- All NP-complete problems are both in NP and NP-hard.
- If any NP-complete problem can be solved in polynomial time, then all NP problems can be solved in polynomial time (P = NP).
- However, solving an NP-hard problem does not necessarily provide a polynomial-time solution for all NP problems

NP vs NP-hard vs NP Complete



- NP-hard problems encompass both problems in NP and problems that are even harder.
- All NP-complete problems are both in NP and NP-hard.
- If any NP-complete problem can be solved in polynomial time, then all NP problems can be solved in polynomial time (P = NP).
- However, solving an NP-hard problem does not necessarily provide a polynomial-time solution for all NP problems
    - Unless the problem is also NP-complete.

Image Source: https://medium.com/intuitionmath/p-np-would-mean-were-a-bunch-of-dumb-apes-20c6e50f0ba3

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time
  - i.e., Is there a clique of size k in graph G

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time
  - i.e., Is there a clique of size k in graph G
- Can we use it to find the size of the largest clique efficiently?

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time
  - i.e., Is there a clique of size k in graph G
- Can we use it to find the size of the largest clique efficiently?
  - Assume this solution is a produce called $clque(G, k)$ and has a running time of n^c

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time
  - i.e., Is there a clique of size k in graph G
- Can we use it to find the size of the largest clique efficiently?
  - Assume this solution is a produce called $clque(G, k)$ and has a running time of n^c
  - We can use search (binary or linear) for $k = |G| \to 0$

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time
  - i.e., Is there a clique of size k in graph G
- Can we use it to find the size of the largest clique efficiently?
  - Assume this solution is a produce called $clque(G, k)$ and has a running time of n^c
  - We can use search (binary or linear) for $k = |G| \rightarrow 0$
    - This takes $|G| * n^c$, which is still polynomial

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time
  - i.e., Is there a clique of size k in graph G
- Can we use it to find the size of the largest clique efficiently?
  - Assume this solution is a produce called $clque(G, k)$ and has a running time of n^c
  - We can use search (binary or linear) for $k = |G| \rightarrow 0$
    - This takes $|G| * n^c$, which is still polynomial
- What about identifying what the clique is?

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time
    - i.e., Is there a clique of size k in graph G
- Can we use it to find the size of the largest clique efficiently?
    - Assume this solution is a produce called $clque(G, k)$ and has a running time of n^c
    - We can use search (binary or linear) for $k = |G| \to 0$
        - This takes $|G| * n^c$, which is still polynomial
- What about identifying what the clique is?
    - First find the maximum value k

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time
  - i.e., Is there a clique of size k in graph G
- Can we use it to find the size of the largest clique efficiently?
  - Assume this solution is a produce called $clque(G, k)$ and has a running time of n^c
  - We can use search (binary or linear) for $k = |G| \to 0$
    - This takes $|G| * n^c$, which is still polynomial
- What about identifying what the clique is?
  - First find the maximum value k
  - Then, try to remove a vertex one at a time and see if the remaining graph is clique

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time
  - i.e., Is there a clique of size k in graph G
- Can we use it to find the size of the largest clique efficiently?
  - Assume this solution is a produce called $clque(G, k)$ and has a running time of n^c
  - We can use search (binary or linear) for $k = |G| \to 0$
    - This takes $|G| * n^c$, which is still polynomial
- What about identifying what the clique is?
  - First find the maximum value k
  - Then, try to remove a vertex one at a time and see if the remaining graph is clique
    - if it is not, this vertex is required in the largest clique so put it back and continue to the other nodes

Decision Problems vs Optimization Problems

- Let's assume we can solve the clique decision problem in polynomial time
    - i.e., Is there a clique of size k in graph G
- Can we use it to find the size of the largest clique efficiently?
    - Assume this solution is a produce called $clque(G, k)$ and has a running time of n^c
    - We can use search (binary or linear) for $k = |G| \rightarrow 0$
        - This takes $|G| * n^c$, which is still polynomial
- What about identifying what the clique is?
    - First find the maximum value k
    - Then, try to remove a vertex one at a time and see if the remaining graph is clique
        - if it is not, this vertex is required in the largest clique so put it back and continue to the other nodes
        - Stop when you are left with k vertices

The Independent Set Problem

The Independent Set Problem

- Given an undirected graph, determine if there exists an independent set of size k.

The Independent Set Problem

- Given an undirected graph, determine if there exists an independent set of size k.
- An independent set is a subset of vertices such that no two vertices in the subset are connected by an edge.

The Independent Set Problem

- Given an undirected graph, determine if there exists an independent set of size k.
- An independent set is a subset of vertices such that no two vertices in the subset are connected by an edge.
- The problem can take different forms:
    - Is the graph G an independent set?

The Independent Set Problem

- Given an undirected graph, determine if there exists an independent set of size k.
- An independent set is a subset of vertices such that no two vertices in the subset are connected by an edge.
- The problem can take different forms:
    - Is the graph G an independent set?
    - Is there an independent set within G?

The Independent Set Problem

- Given an undirected graph, determine if there exists an independent set of size k.
- An independent set is a subset of vertices such that no two vertices in the subset are connected by an edge.
- The problem can take different forms:
  - Is the graph G an independent set?
  - Is there an independent set within G?
  - Is there an independent set within G of size k?

The Independent Set Problem

- Given an undirected graph, determine if there exists an independent set of size k.
- An independent set is a subset of vertices such that no two vertices in the subset are connected by an edge.
- The problem can take different forms:
  - Is the graph G an independent set?
  - Is there an independent set within G?
  - Is there an independent set within G of size k?
  - Which nodes make such an independent set?

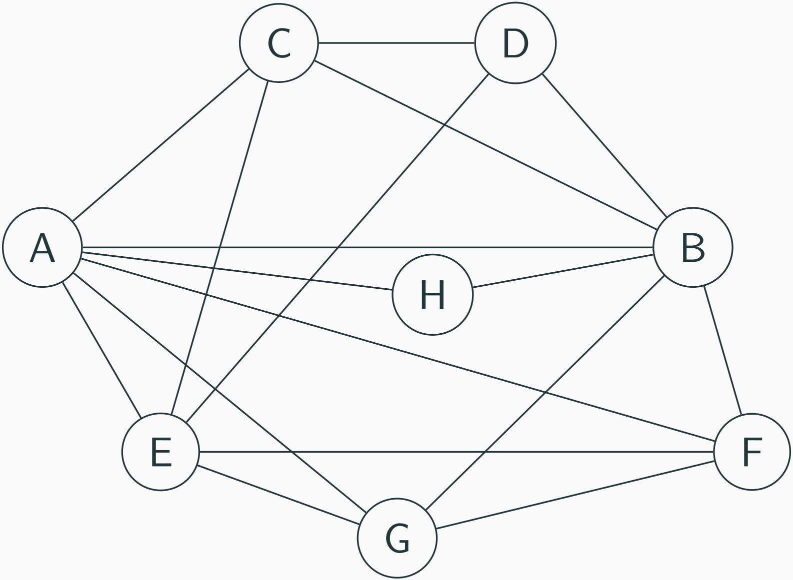
The Independent Set Problem

- Given an undirected graph, determine if there exists an independent set of size k.
- An independent set is a subset of vertices such that no two vertices in the subset are connected by an edge.
- The problem can take different forms:
  - Is the graph G an independent set?
  - Is there an independent set within G?
  - Is there an independent set within G of size k?
  - Which nodes make such an independent set?
- No known polynomial-time solution exists for the last three forms.

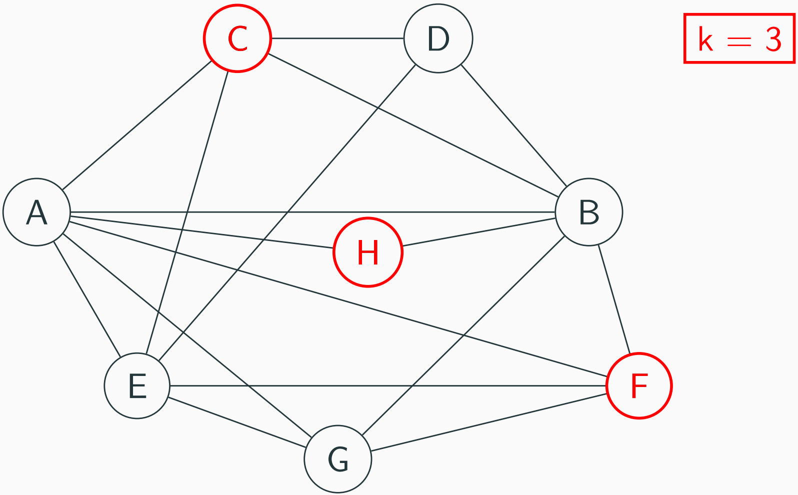
The Independent Set Problem

- Given an undirected graph, determine if there exists an independent set of size k.
- An independent set is a subset of vertices such that no two vertices in the subset are connected by an edge.
- The problem can take different forms:
  - Is the graph G an independent set?
  - Is there an independent set within G?
  - Is there an independent set within G of size k?
  - Which nodes make such an independent set?
- No known polynomial-time solution exists for the last three forms.
- **Brute Force Solution**: Check all possible subsets of vertices of size k to see if they form an independent set, which takes exponential time.

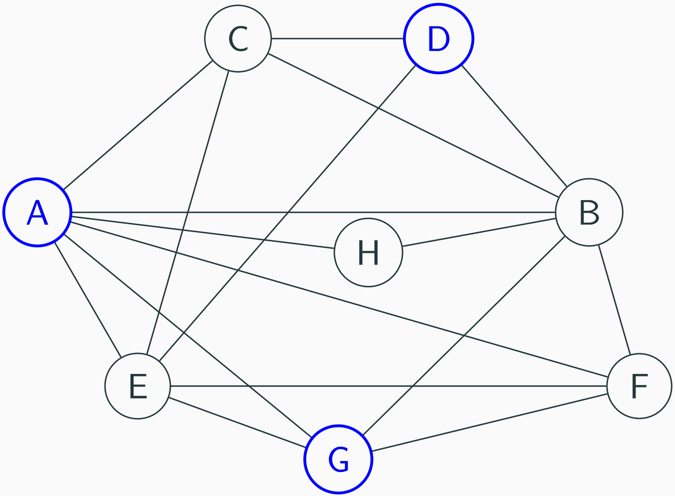
The Independent Set Problem: Example



The Independent Set Problem: Example

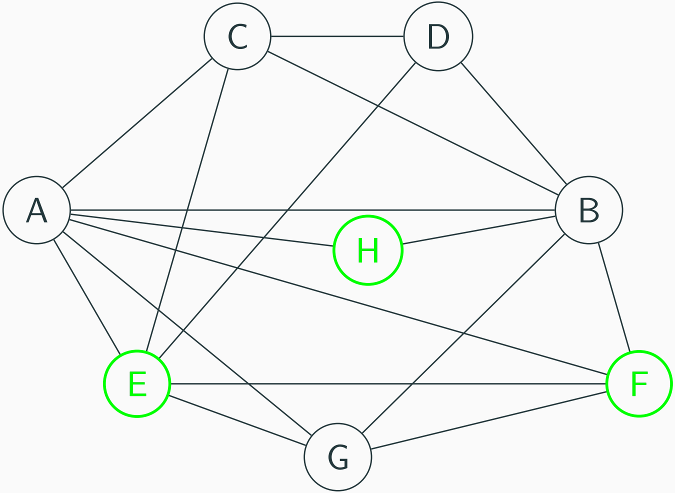


The Independent Set Problem: Example



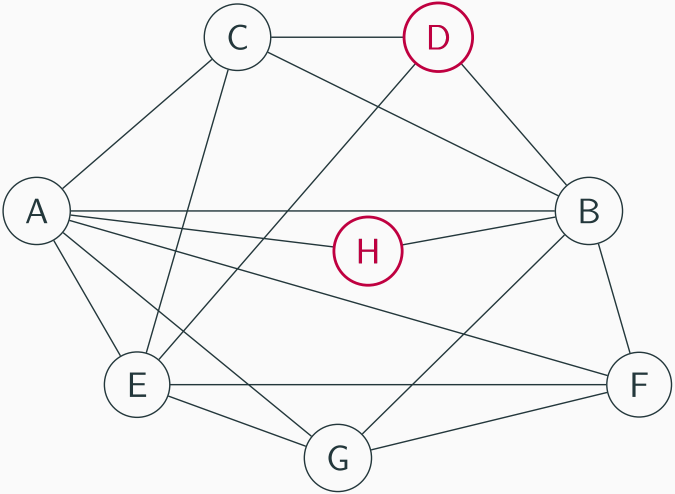
$\boxed{k = 3}$

The Independent Set Problem: Example



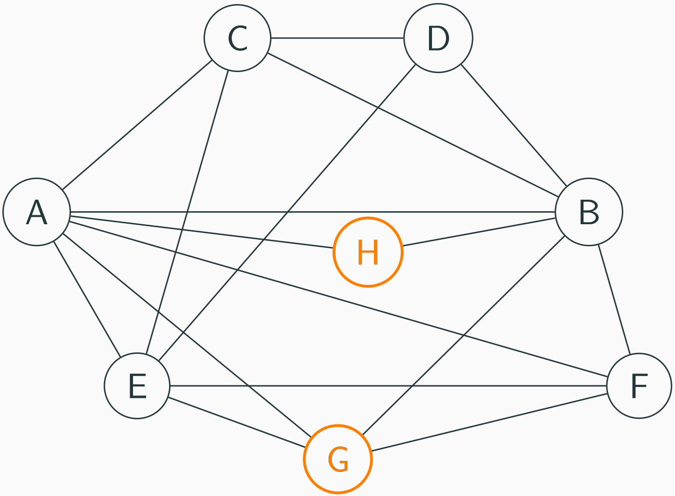
$\boxed{k = 3}$

The Independent Set Problem: Example



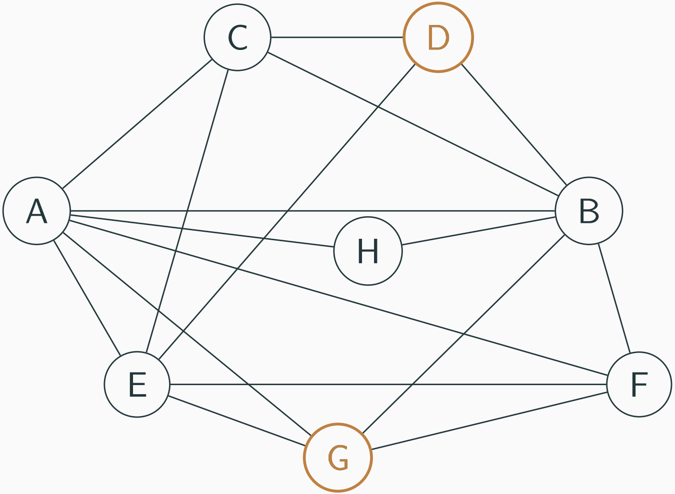
$\boxed{k = 2}$

The Independent Set Problem: Example



$\boxed{k = 2}$

The Independent Set Problem: Example



$\boxed{k = 2}$

The Independent Set Problem: NP Complete

- We want to show that the Independent Set Problem problem is NP Complete
- This means showing that it is both NP and NP hard
- To show it is NP means that if a certificate is generated for the solution we can verify it with polynomial efficiency
- To show it is NP-Hard means to show that every NP-problem can be polynomially reduced to it
    - In practice, we don't need to show reduction for every problem
    - We only need to show this for a well known NP-hard problem
    - We'll assume the clique problem is NP-hard and we show the reduction for it

Reduction: Clique Problem to Independent Set

- **Reduction:** Transform an instance of the Clique problem into an instance of the Independent Set problem.

Reduction: Clique Problem to Independent Set

- **Reduction:** Transform an instance of the Clique problem into an instance of the Independent Set problem.
- Given a graph G and an integer k, the Clique problem asks if there is a clique of size k in G.

Reduction: Clique Problem to Independent Set

- **Reduction:** Transform an instance of the Clique problem into an instance of the Independent Set problem.
- Given a graph G and an integer k, the Clique problem asks if there is a clique of size k in G.
- Construct a new graph G' by taking the complement of G (i.e., create G' where two vertices are adjacent in G' if and only if they are not adjacent in G).

Reduction: Clique Problem to Independent Set

- **Reduction:** Transform an instance of the Clique problem into an instance of the Independent Set problem.
- Given a graph G and an integer k, the Clique problem asks if there is a clique of size k in G.
- Construct a new graph G' by taking the complement of G (i.e., create G' where two vertices are adjacent in G' if and only if they are not adjacent in G).
- In the new graph G', an independent set of size k in G' corresponds to a clique of size k in G.

Reduction: Clique Problem to Independent Set

- **Reduction:** Transform an instance of the Clique problem into an instance of the Independent Set problem.
- Given a graph G and an integer k, the Clique problem asks if there is a clique of size k in G.
- Construct a new graph G' by taking the complement of G (i.e., create G' where two vertices are adjacent in G' if and only if they are not adjacent in G).
- In the new graph G', an independent set of size k in G' corresponds to a clique of size k in G.
- Therefore, solving the Independent Set problem on G' can be used to solve the Clique problem on G.

Reduction: Clique Problem to Independent Set

- **Reduction:** Transform an instance of the Clique problem into an instance of the Independent Set problem.
- Given a graph G and an integer k, the Clique problem asks if there is a clique of size k in G.
- Construct a new graph G' by taking the complement of G (i.e., create G' where two vertices are adjacent in G' if and only if they are not adjacent in G).
- In the new graph G', an independent set of size k in G' corresponds to a clique of size k in G.
- Therefore, solving the Independent Set problem on G' can be used to solve the Clique problem on G.
- This reduction shows that the Independent Set problem is NP-hard because the Clique problem is NP-hard.

Reduction: Time Complexity of Constructing the Complement Graph

- Constructing a new graph G' by taking the complement of G involves:

Reduction: Time Complexity of Constructing the Complement Graph

- Constructing a new graph G' by taking the complement of G involves:
    - Initializing G' with the same set of vertices as G.

Reduction: Time Complexity of Constructing the Complement Graph

- Constructing a new graph G' by taking the complement of G involves:
    - Initializing G' with the same set of vertices as G.
    - For each pair of vertices (u, v):
        - Check if there is an edge between u and v in G.
        - If there is no edge in G, add an edge between u and v in G'.

Reduction: Time Complexity of Constructing the Complement Graph

- Constructing a new graph G' by taking the complement of G involves:
  - Initializing G' with the same set of vertices as G.
  - For each pair of vertices (u, v):
    - Check if there is an edge between u and v in G.
    - If there is no edge in G, add an edge between u and v in G'.
- Time Complexity Analysis:
  - There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of vertices to check.
  - Each check and potential edge addition takes $O(1)$ time.

Reduction: Time Complexity of Constructing the Complement Graph

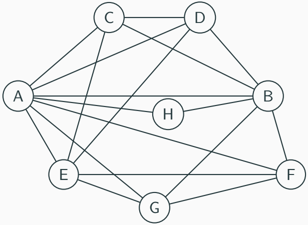
- Constructing a new graph G' by taking the complement of G involves:
  - Initializing G' with the same set of vertices as G.
  - For each pair of vertices (u, v):
    - Check if there is an edge between u and v in G.
    - If there is no edge in G, add an edge between u and v in G'.
- Time Complexity Analysis:
  - There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of vertices to check.
  - Each check and potential edge addition takes $O(1)$ time.
- Overall Time Complexity: $O(n^2)$.

Reduction: Time Complexity of Constructing the Complement Graph

- Constructing a new graph G' by taking the complement of G involves:
  - Initializing G' with the same set of vertices as G.
  - For each pair of vertices (u, v):
    - Check if there is an edge between u and v in G.
    - If there is no edge in G, add an edge between u and v in G'.
- Time Complexity Analysis:
  - There are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of vertices to check.
  - Each check and potential edge addition takes $O(1)$ time.
- Overall Time Complexity: $O(n^2)$.
- This means our reduction is a **polynomial reduction**

Reduction: Clique to Independent Set

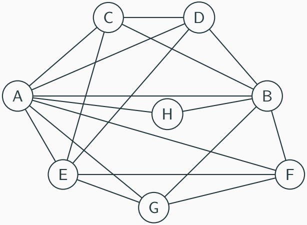
- Start with the original graph G for the Clique problem.



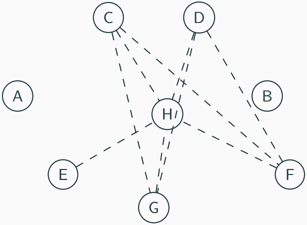
Graph G (Clique problem)

Reduction: Clique to Independent Set

- Start with the original graph G for the Clique problem.
- Create the complement graph G':
  - Two vertices are connected in G' if and only if they are not connected in G.



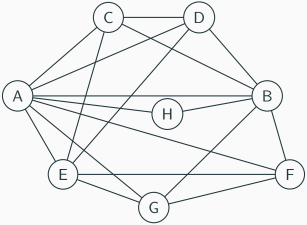
Graph G (Clique problem)



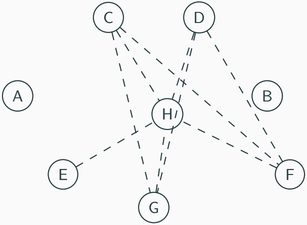
Graph G' (Independent Set problem)

Reduction: Clique to Independent Set

- Start with the original graph G for the Clique problem.
- Create the complement graph G':
  - Two vertices are connected in G' if and only if they are not connected in G.
- An independent set in G' corresponds to a clique in G.



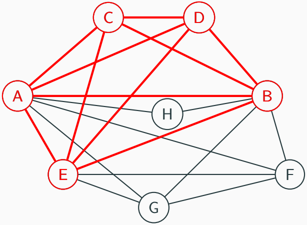
Graph G (Clique problem)



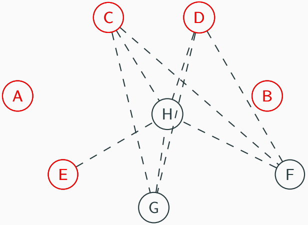
Graph G' (Independent Set problem)

Reduction: Clique to Independent Set

- Start with the original graph G for the Clique problem.
- Create the complement graph G':
  - Two vertices are connected in G' if and only if they are not connected in G.
- An independent set in G' corresponds to a clique in G.
  - For example, the independent set (A, B, C, D, E) in G' corresponds to the clique (A, B, C, D, E) in G.



Graph G (Clique problem)



Graph G' (Independent Set problem)

Reduction: Note

- In this particular example reduction is symmetric

Reduction: Note

- In this particular example reduction is symmetric
  - If *clique* $<_p$ *independetset* and *independetset* $<_p$ *clique*

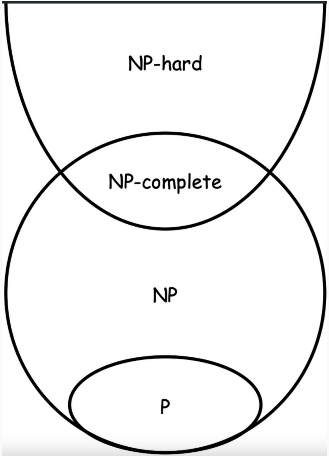
Reduction: Note

- In this particular example reduction is symmetric
  - If *clique* $<_p$ *independetset* and *independetset* $<_p$ *clique*
  - This is not the general case
    - $X <_p Y \not\Rightarrow Y <_p X$

Conclusion

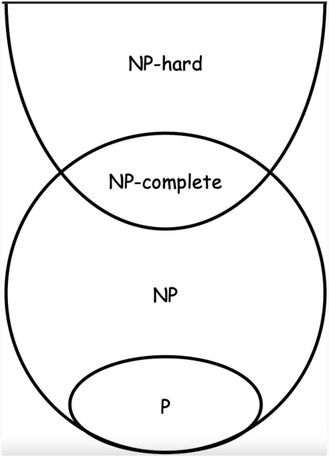
P vs NP

- There are a number of questions that are still open in the P vs NP domain



P vs NP

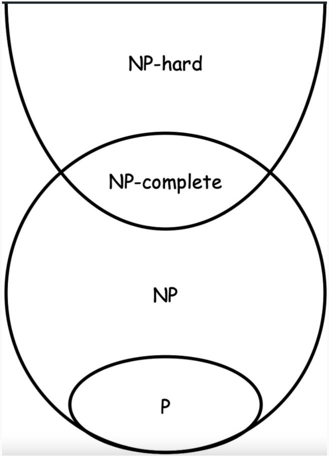
- There are a number of questions that are still open in the P vs NP domain



- $P = NP$?

P vs NP

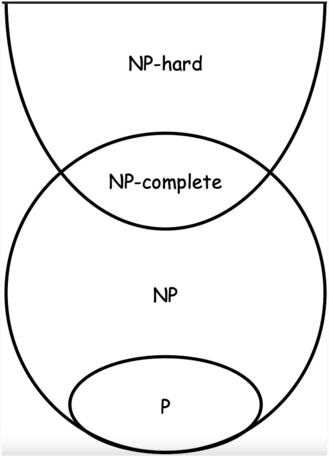
- There are a number of questions that are still open in the P vs NP domain



- $P = NP$?
    - Part of the millennium prize challenges

P vs NP

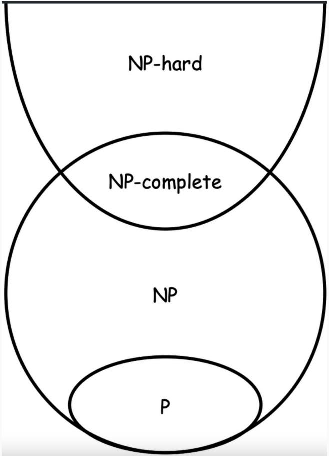
- There are a number of questions that are still open in the P vs NP domain



- $P = NP$?
  - Part of the millennium prize challenges
- $NP = co\text{-}NP$?

P vs NP

- There are a number of questions that are still open in the P vs NP domain



- $P = NP$?
  - Part of the millennium prize challenges
- $NP = co\text{-}NP$?
- $P = NP \cap co\text{-}NP$?
  - **EXP**: Verification takes exponential time
  - **PSpace** - Problems that can be verified in polynomial space given unlimited time
  - **BPP** - Problems that can be solved with probabilistic algorithm in polynomial time
  - **BQP** - Solvable via quantum computing
  - **Others**: EXSpace, 2-Exp and even unsolvable

Conclusion

- Understanding computational intractability and reductions helps classify problems and develop efficient algorithms.
- The study of NP-complete problems and their relationships with other problems is crucial in computer science.
- In practice, we use reduction to show that a problem is difficult to solve of practical purposes (i.e., intractable)
- An entire sub-field of CS/Math, called Computational complexity theory, is dedicated to studying these issues