P and NP

Complexity Hierarchy

- We have covered different algorithmic techniques for solving different graph problems.
- Solved with time complexity
  - Dijkstra's algorithm: $O(V^2)$ --> $O((V+E)\log V)$
  - Prim's algorithm: $O(V^2)$ --> $O((V+E)\log V)$
  - Bipartite matching: $O((V+E)V)$
  - Finding Euler circuit: $O(V+E)$
  - Finding Hamiltonian cycle: ?
- Questions to ask:
  - Is finding Hamiltonian cycle harder than finding Euler circuit?
  - Is that an inherent phenomenon that graph problems could be solved in polynomial time?

Lower Bound

- The lower bound for a problem is the minimum number of steps that any algorithm must take.
- If we can prove a lower bound for finding Hamiltonian cycle is larger than $\Omega(V + E)$, then we can claim that finding Hamiltonian cycle is harder than finding Euler circuit.
- Thus, proving lower bounds will help us to classify problems into different classes depending on their inherent difficulty.

Tractable Problems

- It has been commonly accepted that a problem is tractable if there exists a polynomial-time algorithm for it.
  - There exists efficient algorithms to solve such problems exactly.
- Can still meaningless practically if the degree of the polynomial is too large, but a theoretical level of discussion ignores the issue of how large a degree is acceptable.
  - Finding shortest paths, minimum spanning tree, bipartite matching, finding Euler circuit are all tractable problems.

Intractable Problems

- Conversely, a problem is intractable if there does not exist a polynomial-time algorithm for it.
  - No efficient algorithms to solve such problems exactly.

- Not knowing a polynomial-time algorithm does not imply that one does not exist.
  - Hamiltonian cycle?

Intractable Problems

- Some problems are provably intractable.

Example:
Suppose that you are given logical expressions involving the set of real numbers, $+, =, <, \exists$, and $\forall$. For example, $\forall x \exists y (y = x + 1)$, $\exists y \forall x (x < y \vee x = y)$. The goal is to design an algorithm to verify whether a given logical expression is true. It has been proved that it takes $\Omega(2^{cn})$ for some fixed constant $c > 0$, no matter what algorithm is used. Thus this problem is certainly intractable.

Intractable Problems

- Some problems are actually unsolvable (undecidable problems). That is, there does not exist any algorithm (no matter what the running time is) to solve the problems exactly.

The well-known example is the halting problem: it can be proved that there does not exist an algorithm that can decide whether a program will loop indefinitely on certain input.

Proof of Halting Problem

- The program to be examined is a function: `void f(char *t);`
- Suppose that the halting problem can be solved by a function:

```
bool halt(char *f_code, char *t);
```

- Construct a function:

```
void modified_halt(char *f_code) {
    if (halt(f_code, f_code)) {
        while (true) { }
    }
    else {
        return;
    }
}
```

- Examine: `halt(modified_halt_code, modified_halt_code)`

Intractable Problems

- So what about those problems that are not known to be polynomial-time solvable but also not known to be intractable?
- Generally, people believe that many of such problems are likely to be intractable except that a proof is missing.
  - Not all, e.g., linear programming was not known to be polynomial time solvable until the ellipsoid method was discovered.
- Stephen Cook discovered a strong reason to suspect that some of these problems should be intractable. These are the so-called NP-complete problems.
  - A proof is still missing.

Some NP-Complete Problems

- **Partition Problem**: Given a set A of integers, is there a subset $B \subseteq A$ such that $\sum_{i \in B} i = \sum_{i \in A \setminus B} i$?
- **Knapsack Problem**: Given a knapsack of capacity S, n items of volumes v_i and weights w_i, and an integer K, is there a packing of total weight at least K.
- **Integer programming**: Given a system of linear equations, is there a solution such that all variables have integral values?

Some NP-Complete Problems

- **Hamiltonian Cycle**: Given an undirected graph, is there a cycle that goes through each vertex exactly once?
- **Independent Set**: A subset of vertices of a graph is an independent set if no two of them are adjacent. Given a graph G and an integer K, is there an independent set of G of at least K vertices?
- **Vertex Cover:** A vertex cover of a graph is a subset of vertices such that every edge is incident to a vertex in this subset. Given a graph G and an integer K, is there is vertex cover of at most K vertices?

Definition of NP (nondeterministically polynomial-time solvable)

- First, assume a computer that can make random choices (nondeterministic Turing machine). We call such a computer a nondeterministic computer. An algorithm for a nondeterministic computer may then provide, for every step, a set of possible instructions to execute, and ask the computer to randomly pick one to execute.

- We say that the algorithm solves a problem is there is a sequence of steps that will lead to a solution of the problem. The running time is the number of such steps.

- A problem belongs to the class NP if there exists an algorithm for a nondeterministic computer that solves the problem in time polynomial in the size of the input problem instance.

Characteristics of NP-Complete Problems

- (Verifier-based) Definition of NP: Given any instance of the problem and a candidate solution for that instance, the algorithm take time polynomial in the input size to verify if the candidate solution is correct.

- NP-Complete problems belong to NP.
- Very easy to verify whether a given candidate solution is correct, although we suspect that NP-complete problems are likely to be intractable.

Example 1: Partition problem

- Given a set A of integers, is there a subset $B \subseteq A$ such that $\sum_{i \in B} i = \sum_{i \in A \setminus B} i$?

- A candidate solution is supposed to be a subset B of A. We check the following things:
  1. Check if every integer in B appears in A.
  2. Sum all the integers in B.
  3. Sum all the integers in A
  4. Check if the first sum is exactly half of the second sum.
- Running time is $O(|B| \cdot |A|)$+$O(|B|)$+$O(|A|)$+ $O(1)$ which is $O(|A|^2)$

Example 2: Hamiltonian cycle

- Given an undirected graph, is there a cycle that goes through each vertex exactly once?

- A candidate solution will be a list of vertices which is supposed to represent a cycle visiting each vertex exactly once. We check the following things:
  1. Check if the first and last vertices are identical.
  2. Check if every vertex in the graph appears exactly once in the list, except the first and last vertices which are the same vertex appearing twice.
  3. Check if there is an edge between adjacent vertices in the list.
- Running time is $O(1)+O(|V|)+O(|V|)$, assuming that the graph is represented using adjacency matrix.

Decision Versus Optimization

- A problem is a decision problem if its solution is either "yes" or "no".
- Examples: Hamiltonian cycle, partition problem, is a graph connected?

- A problem is an optimization problem if one has to maximize or minimize an associated objective function value.
- Examples: shortest path, spanning tree, bipartite matching

Relation Between Optimization and Decision

- Example 1: Shortest Path
  - Optimization version: given a weighted graph G and two vertices s and d, find the shortest path from s to d.
  - Decision version: given a weighted graph G, two vertices s and d, and an integer K, is there a path from s to d with cost at most K?

- Example 2: Maximum Bipartite Matching
  - Optimization version: given a bipartite graph G, find the maximum matching.
  - Decision version: given a bipartite graph G and an integer K, is there a matching with size of at least K?

Significance of Decision Problem

- If one has a polynomial-time algorithm for an optimization problem, then it can be used to solve the decision version of the problem in polynomial time.
- Example: Shortest Path
  - If you could solve the optimization version and got a solution of value M, then you could just check to see if $M \leq K$
- Conversely, if the decision version is intractable, then the optimization problem must be intractable.

- Thus, if the goal is to show that a problem is intractable, then it suffices to consider its decision version.

Significance of Decision Problem

- We can also go from decision to optimization:
- Example: Independent Set
  - Decision version: Given a graph G and a number K, does G contain a set of independent vertices of size at least K?
  - Optimization version: Given a graph G, find the largest independent set.
- Suppose we have an algorithm for the decision version, how can we find the largest independent set?
  - Try every possible K – there are only V of them.
  - Or even better use binary search.

Significance of Decision Problem

- The decision version of a problem is easier than (or the same as) the optimization version.

- Strictly speaking, the class NP and P (polynomial-time solvable) are defined to classify decision problems only. But most people will also talk about their optimization versions being in P or NP.

Relation Between P and NP

- **P** is the set of decision problems that can be answered in polynomial time.
- **NP** is the set of decision problems such that a given candidate solution can be verified in polynomial time.

- Theorem: $P \subseteq NP$

The Big Question: $P = NP$?

- If $P = NP$: If the solution to a problem can be verified in polynomial time, it can be found in polynomial time.
- $P = NP$ is unlikely because of the following:
  - Stephen Cook discovered in 1971 an NP problem, such that all other decision problems in NP can be transformed into this particular decision problem.
  - Intuitively, such a problem is the hardest in NP. Hence the name NP-Complete.
  - If there is any polynomial time algorithm for this problem, then all problems in NP can be solved in polynomial time.

$P = NP?$

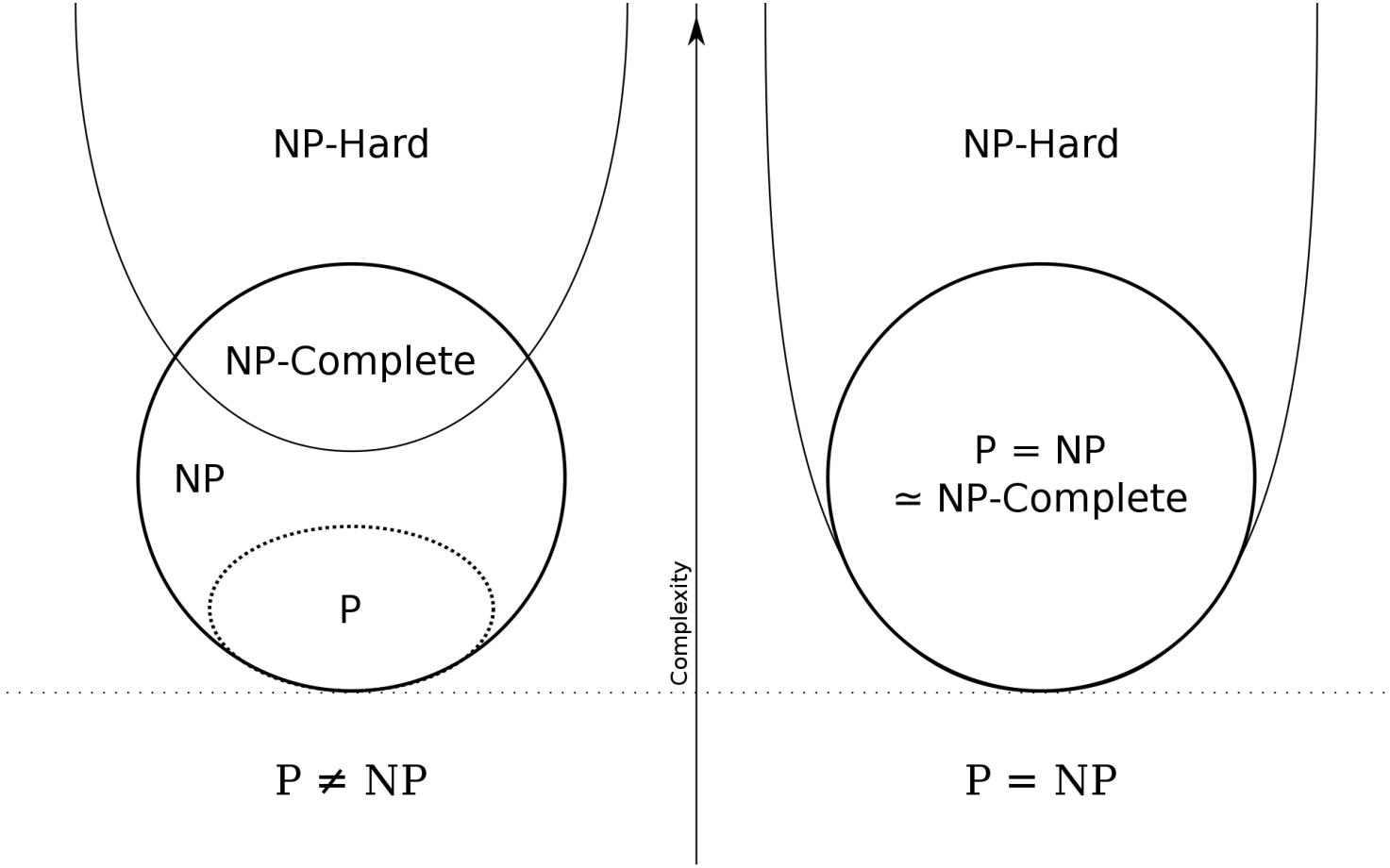

Figure from Wikipeida

Polynomial-time Reducibility

- We say that problem X is polynomial-time reducible to problem Y if there is an algorithm $\mathcal{A}$ such that given any instance x of X, $\mathcal{A}$ outputs an instance y of Y in time polynomial in size of x. Moreover, the answer for x is "yes" iff the answer for y is "yes".

- We write $X \leq_p Y$

Significance of Polynomial-time Reducibility

- If $X \leq_p Y$ and there is a polynomial-time algorithm $\mathcal{B}$ to solve problem Y, then one can solve problem X as follows.
  - Given any instance x of X, run algorithm $\mathcal{A}$ on x to generate an instance y of Y. Then run algorithm $\mathcal{B}$ on y and output the result.

- Conversely, if $X \leq_p Y$ and X is known to be intractable, then Y must also be intractable.

First NP-Complete Problem: SAT

- Given any boolean variable u, both u and $\bar{u}$ are called literals. A clause of literals $x_1, x_2, \ldots, x_k$ is a disjunction of them: $x_1 \vee x_2 \vee \cdots \vee x_k$. Note that two literals in a clause may be identical or two literals may also be complement of each other.
- A conjunctive normal form or CNF is a conjunction of clauses $C_1, C_2, \ldots, C_m$: $C_1 \wedge C_2 \wedge \cdots \wedge C_m$. A CNF is satisfiable if there is a truth assignment of the boolean variables such that the CNF returns 1.
- The satisfiability problem or SAT is: given a CNF, is it satisfiable?

Cook's Theorem

- SAT is NP-Complete:

1. $SAT \in NP$.
2. For all problem $X \in NP, X \leq_P SAT$.

NP-Complete Problems

- Given a decision problem X that is in NP, if we can prove that:

$$SAT \leq_p X$$

- then X is exactly as hard as SAT:
  - X can be solved in polynomial time iff SAT can be solved in polynomial time;
  - X is intractable iff SAT is intractable.

- These problems are all called NP-complete problems.

Proof of Reducibility

- To prove that problem X is polynomial-time reducible to problem Y
- Step 1: design an algorithm that maps each instance of X to an instance of Y
- Step 2: show that the answer to each instance of X is “yes” iff the answer to the corresponding instance in Y is “yes”.

Examples of Reducibility
Independent Set $\leq_p$ Clique

- Independent Set
  - Optimization version: Given a graph G, find the largest independent set.
  - Decision version: Given a graph G and a number K, does G contain a set of independent vertices of size at least K?

- Clique
  - Optimization version: Given a graph G, find the largest clique.
  - Decision version: given G and an integer K, is there a clique of at least K vertices?

Given an undirected graph G, a subgraph H is a clique if there is an edge between every pair of vertices in H. That is, H is a complete graph.

Examples of Reducibility
Independent Set $\leq_p$ Clique

- Step 1: design an algorithm that maps each instance of Independent Set problem to an instance of Clique problem

Construct a graph $\bar{G}$. $\bar{G}$ contains the same set of vertices as G. There is an edge between every pair of vertices in $\bar{G}$ that are not adjacent in G. Then $\bar{G}$ and the integer K form an instance of the Clique problem:

Is there clique of at least K vertices in $\bar{G}$?

Examples of Reducibility
Independent Set $\leq_p$ Clique

- Step 2: show that the answer to the Independent Set problem is “yes” iff the answer to the corresponding Clique problem is “yes”.

- (-->) If there is an independent set H of at least K vertices in G, then any two vertices in H are adjacent in $\bar{G}$ by construction. Thus, the vertices in H form a clique of at least K vertices in $\bar{G}$.
- (<--) if there is a clique I of at least K vertices in $\bar{G}$, then no two vertices in I are adjacent in G by construction. Thus, the vertices in I form an independent set of at least K vertices in G.

NP-Complete Problems

- A decision problem Z is NP-complete if
  - Z is in NP, and
  - Every problem Y in NP is reducible to Z in polynomial time, i.e., $Y \leq_p Z$
- To prove a decision problem X is NP-complete:
  - Show that X is in NP, and
  - Reduce SAT or any known NP-complete problem Z to X, i.e., $Z \leq_p X$
- To prove a decision problem X is NP-hard:
  - Reduce SAT or any known NP-complete problem Z to X, i.e., $Z \leq_p X$

NP-Complete Problems

- Thousands of other problems have been shown to be NP-complete by reductions from other problems previously shown to be NP-complete.

- If we can solve any NP-complete problem in polynomial time, we can solve all NP-complete problems in polynomial time.

Vertex Cover is NP-Complete

- Given an undirected graph $G = (V, E)$, $S \subseteq V$ is a vertex cover if every edge in G has an endpoint in S.
- Decision version: given a graph G and an integer K, is there a vertex cover of size at most K?
- We will prove that vertex cover is NP-complete.

Vertex Cover $\in$ NP

- A candidate solution S for an instance of Vertex Cover is a subset of vertices of the given graph G.
- We verify if S contains at most K vertices. If yes, we examine each edge in G and check if one of its endpoints is in S.
- This takes $O(|V| \cdot |E|)$ time.

Clique $\leq_p$ Vertex Cover

- Step 1: design an algorithm that maps each instance of Clique problem to an instance of Vertex Cover problem

Given a graph G and an integer K, construct the graph $\bar{G}$ in the same way as the previous example and compute the integer $|V| - K$. This takes $O(|V|^2)$ time.

$\bar{G}$ and the integer $|V| - K$ form an instance of the Vertex Cover problem. That is, is there a vertex cover in $\bar{G}$ of size at most $|V| - K$?

Clique $\leq_p$ Vertex Cover

- Step 2: show that answer for the Clique instance is "yes" iff the answer for the Vertex Cover instance is "yes".

- (-->)
- Let C be a clique of size at least K in G. Then we claim that $V \setminus C$ is vertex cover of size at most $|V| - K$ in $\bar{G}$.
- For each edge $\{u, v\}$ in $\bar{G}$, u and v are not adjacent in G. So u or v does not belong to C. Thus, u or v belongs to $V \setminus C$. So $V \setminus C$ is a vertex cover.

Clique $\leq_p$ Vertex Cover

- (<--)
- Let S be a vertex cover of $\bar{G}$ of size at most $|V| - K$. We claim that $V \setminus S$ is a clique in G of size at least K. That is, for any two vertices u and v in $V \setminus S$, u and v must be adjacent in G.
- Otherwise, u and v are not adjacent in G and so u and v are adjacent in $\bar{G}$. Thus, u or v belongs to S as S is a vertex cover of G. But then u or v does not belong to $V \setminus S$, contradiction.

Computers and Intractability: A Guide to the Theory of NP-Completeness