In the name of God

Part 2. Complexity Theory

2.2. Complexity of Problems

Spring 2010

Instructor: Dr. Masoud Yaghini

Easy vs. Difficult Problems

- **Tractable or Easy Problems**
  - The problems that are solvable by **polynomial-time algorithms** are **tractable**, or **easy**
- **Intractable or Difficult Problems**
  - The problems that require **super-polynomial time** are **intractable**, or **hard**.

Decision & Optimization Problems

- **Decision Problems**
  - Given an input and a question regarding a problem, determine if the answer is **yes** or **no**
- **Example: Prime number decision problem.**
  - Is a given number Q a prime number?
  - It will return *yes* if the number Q is a prime one, otherwise the *no* answer is returned.

Decision & Optimization Problems

- **Optimization Problems**
  - Find a solution with the “best” value
- **Example: Traveling Salesman Problem.**
  - “find the optimal Hamiltonian tour that optimizes the total distance,”

Decision & Optimization Problems

- An optimization problem can always be reduced to a decision problem.
- **Example: Optimization versus decision problem.**
  - The TSP can reduced to a decision problem: “given an integer *D*, is there a Hamiltonian tour with a distance less than or equal to *D*?”

Class P Problems

- **Class P Problems**
  - The family of problems where a known deterministic polynomial-time algorithm exists to solve the problem.
  - They can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.

Class P Problems

- **Some problems of class P**
  - shortest path problems
  - maximum flow network
  - minimum spanning tree
  - maximum bipartite matching
  - linear programming continuous

Nondeterministic Polynomial Algorithms

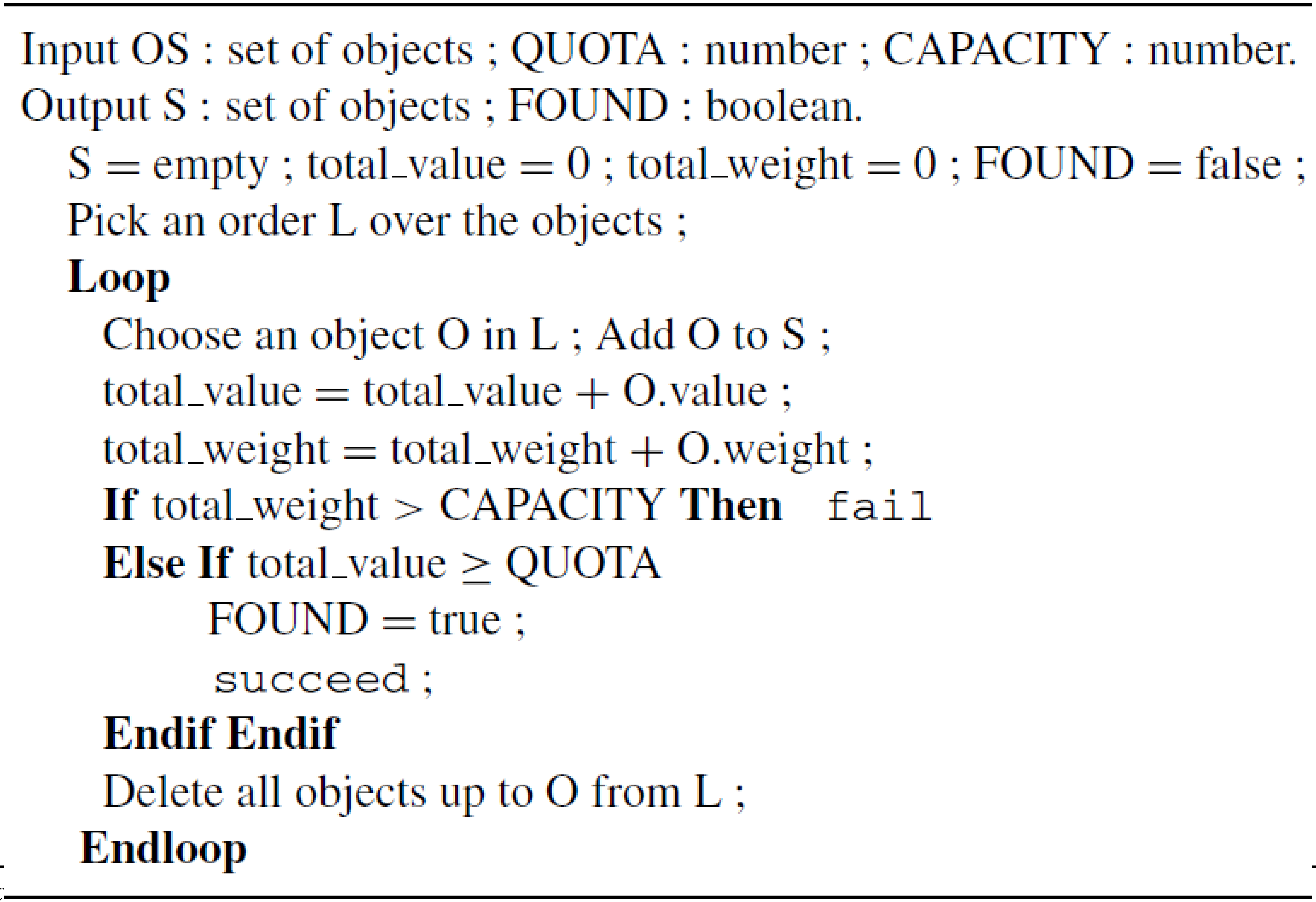
- Nondeterministic algorithm = two stage procedure:
- 1) Nondeterministic ("guessing") stage:
  - generate randomly an arbitrary string that can be thought of as a candidate solution ("certificate")
- 2) Deterministic ("verification") stage:
  - take the certificate and the instance to the problem and returns YES if the certificate represents a solution
- **NP algorithms (Nondeterministic polynomial)**
  - verification stage is polynomial

Nondeterministic Polynomial Algorithms

- **Example: Nondeterministic algorithm for the 0–1 knapsack problem.**
  - The 0–1 knapsack decision problem:
    - Given a set of *N* objects.
    - Each object *O* has a specified weight and a specified value.
    - Given a capacity, which is the maximum total weight of the knapsack, and a quota, which is the minimum total value that one wants to get.
    - The 0–1 knapsack decision problem consists in finding a subset of the objects whose **total weight** is at most **equal to the capacity** and whose total value is **at least equal** to the specified **quota**.

Nondeterministic Polynomial Algorithms

- Nondeterministic algorithm for the knapsack problem



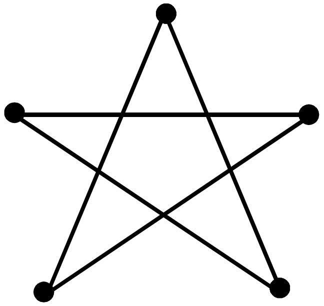
```
Input OS : set of objects ; QUOTA : number ; CAPACITY : number.
Output S : set of objects ; FOUND : boolean.
  S = empty ; total_value = 0 ; total_weight = 0 ; FOUND = false ;
  Pick an order L over the objects ;
  Loop
    Choose an object O in L ; Add O to S ;
    total_value = total_value + O.value ;
    total_weight = total_weight + O.weight ;
    If total_weight $>$ CAPACITY Then  fail
    Else If total_value $\geq$ QUOTA
          FOUND = true ;
          succeed ;
    Endif Endif
    Delete all objects up to O from L ;
  Endloop
```

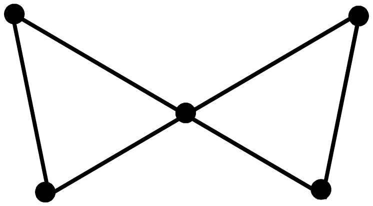
Class NP Problems

- **Class NP Problems**
  - NP problems stands for **Nondeterministic Polynomial-time Problems**
  - The set of all decision problems that can be solved by a **nondeterministic algorithm**.
    - **i.e., verifiable in polynomial time**
  - If we were somehow given a solution, then we could verify that the solution is correct in time polynomial in the size of the input to the problem.
  - **Common error:** NP does **not** mean “**non-polynomial**”

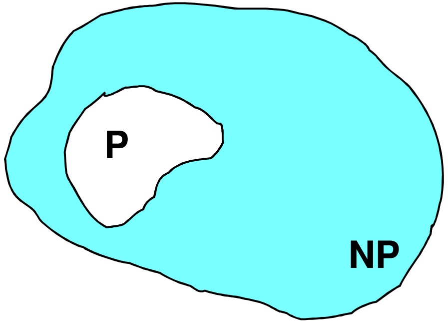
Example: Hamiltonian Cycle

- **Given:** a directed graph $G = (V, E)$, determine a simple cycle that contains each vertex in V
  - Each vertex can only be visited once
- **Certificate**:
  - Sequence: $\langle v_1, v_2, v_3, \ldots, v_{|V|} \rangle$





Is P = NP?



- Any problem in P is also in NP:

$$P \subseteq NP$$

- The big (and open question) is whether NP $\subseteq$ P or P = NP
  - i.e., if it is always easy to check a solution, should it also be easy to find a solution?
  - Obviously, for each problem in P we have a nondeterministic algorithm solving it.
- Most computer scientists believe that this is false but we do not have a proof …

Class NP-Complete Problems

- Class NP-Complete Problems
  - NP-Complete problems stands for **Nondeterministic Polynomial-time Complete Problems**
  - The NP-complete problems are the hardest problems in NP
  - The problems that no one can solve them in a polynomial-time
  - If a polynomial deterministic algorithm exists to solve an NP-complete problem, then all problems of class NP may be solved in polynomial time.

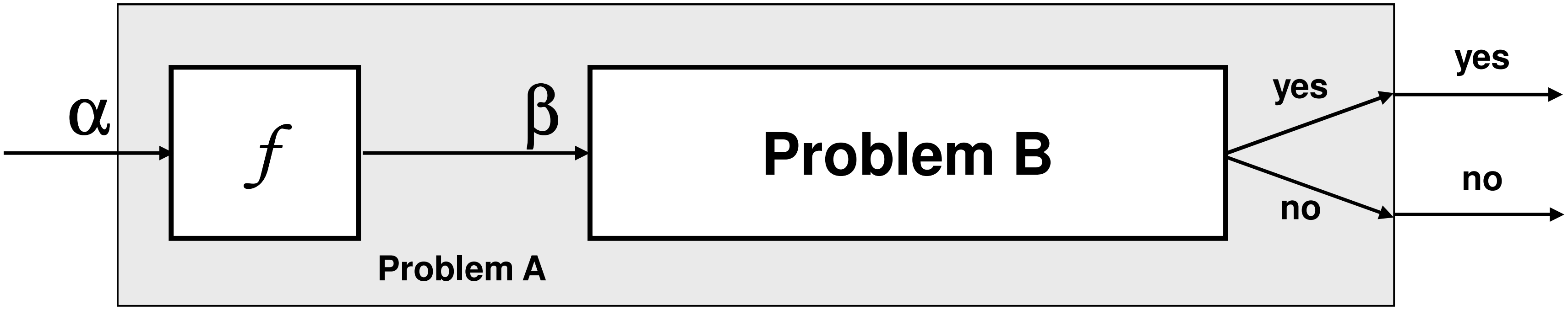
Reductions

- A problem A can be **reduced** to another problem B if any instance of A can be rephrased to an instance of B, the solution to which provides a solution to the instance of A
  - This rephrasing is called a **transformation**
- **Intuitively**: If A reduces in **polynomial time** to B, A is "no harder to solve" than B
- **Example**: lcm(m, n) = m * n / gcd(m, n),

  lcm(m,n) (as A) problem is reduced to gcd(m, n) (as B) problem

Reductions

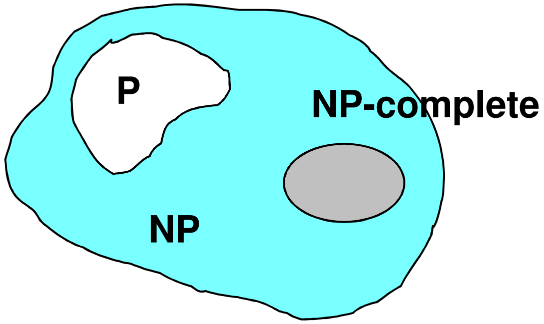
- Reduction is a way of saying that one problem is “easier” than another.
- We say that problem A is easier than problem B,
  (i.e., we write “A $\leq_p$ B”)
- **Idea**: transform the inputs of A to inputs of B



Polynomial Reductions

- Given two problems A, B, we say that A is polynomially reducible to B ($A \leq_p B$) if:
  1. There exists a function *f* that converts the input of A to inputs of B in **polynomial time**
  2. $A(i) = YES \Leftrightarrow B(f(i)) = YES$

NP-Completeness (formally)



- A problem B is **NP-complete** if:

  (1) $B \in$ **NP**

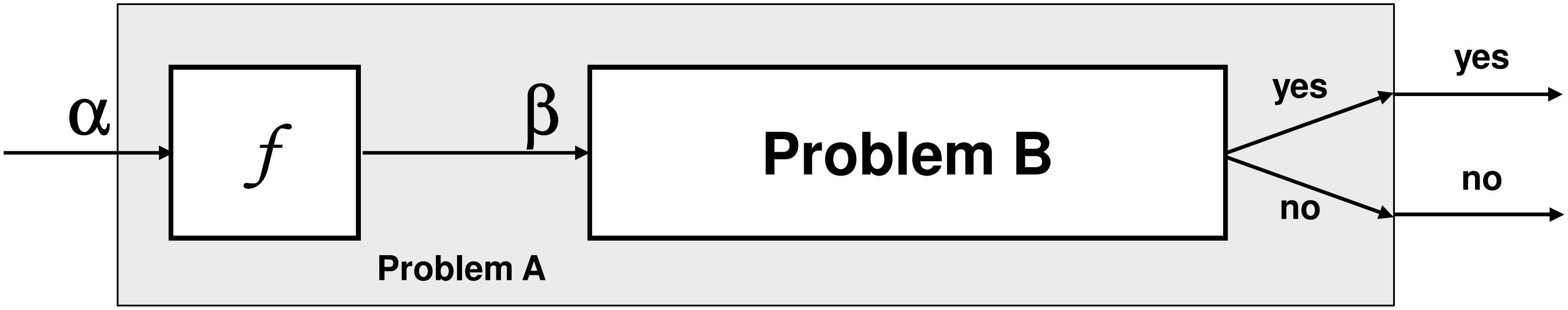
  (2) $A \leq_p B$ for all $A \in$ **NP**

- If B satisfies only property (2) we say that B is **NP-hard**

- No polynomial time algorithm has been discovered for an **NP-Complete** problem

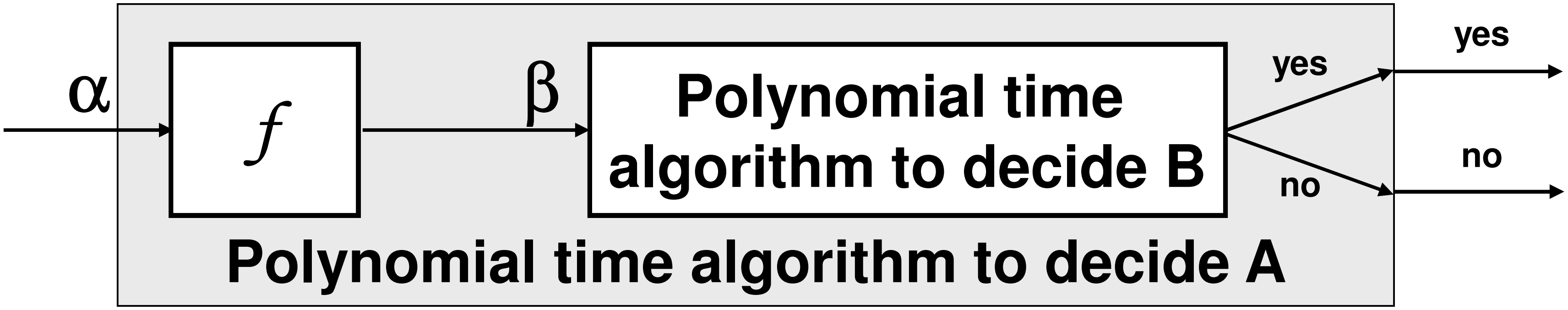
- No one has ever proven that no polynomial time algorithm can exist for any **NP-Complete** problem

Implications of Reduction



- If $A \leq_p B$ and $B \in P$, then $A \in P$
- if $A \leq_p B$ and $A \notin P$, then $B \notin P$

Proving Polynomial Time



1. Use a **polynomial time** reduction algorithm to transform A into B
2. Run a known **polynomial time** algorithm for B
3. Use the answer for B as the answer for A

Proving NP-Completeness

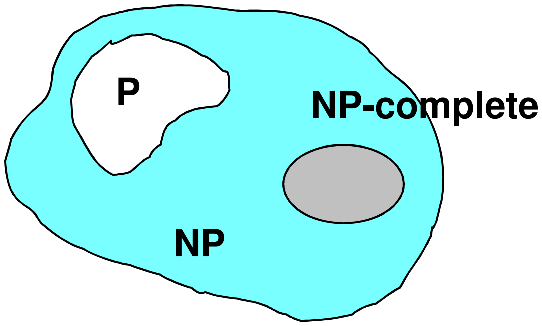
Theorem: If A is NP-Complete and $A \leq_p B$

$\Rightarrow$ B is **NP-Hard**

In addition, if $B \in NP$

$\Rightarrow$ B is **NP-Complete**

Revisit "Is P = NP?"



Theorem: If any NP-Complete problem can be solved in polynomial time ⇒ then P = NP.

Relation among P, NP, NPC

- $P \subseteq NP$ (Sure)
- $NPC \subseteq NP$ (sure)
- $P = NP$ (or $P \subset NP$, or $P \neq NP$) ???
- $NPC = NP$ (or $NPC \subset NP$, or $NPC \neq NP$) ???

P & NP-Complete Problems

- **Shortest simple path**
  - Given a graph G = (V, E) find a **shortest** path from a source to all other vertices
  - Polynomial solution: O(VE)
- **Longest simple path**
  - Given a graph G = (V, E) find a **longest** path from a source to all other vertices
  - NP-complete

P & NP-Complete Problems

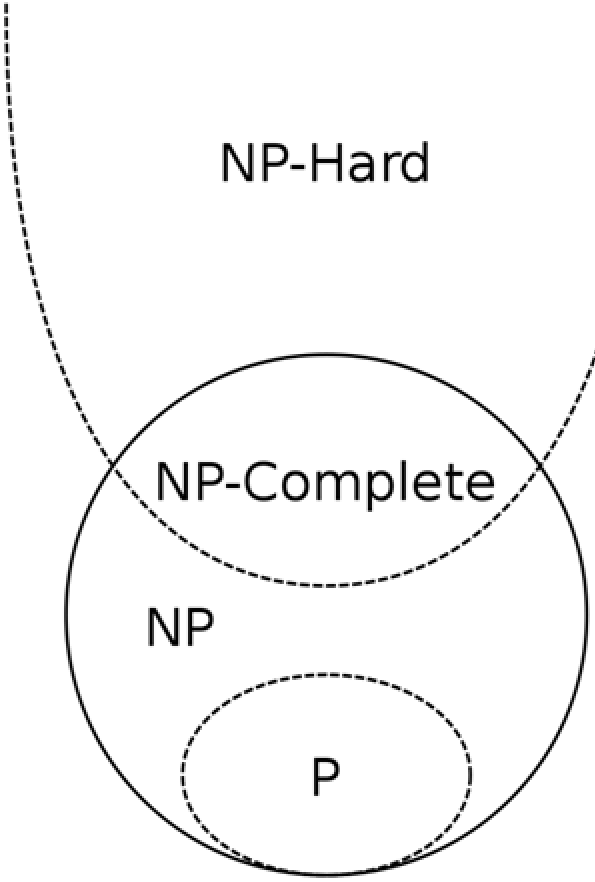
- **Euler tour**
  - G = (V, E) a connected, directed graph find a cycle that traverses each edge of G exactly once (may visit a vertex multiple times)
  - Polynomial solution O(E)
- **Hamiltonian cycle**
  - G = (V, E) a connected, directed graph find a cycle that visits each vertex of G exactly once
  - NP-complete

NP-Hard Problems

- **NP-Hard problems**
  - NP-hard stands for **Nondeterministic Polynomial-time Hard**
  - Most of the real-world optimization problems are NP-hard for which provably efficient algorithms do not exist.
  - They require **exponential time** to be solved in optimality.
  - Metaheuristics constitute an important alternative to solve this class of problems.
  - NP-hard problems may be of any type: decision problems, search problems, or optimization problems.

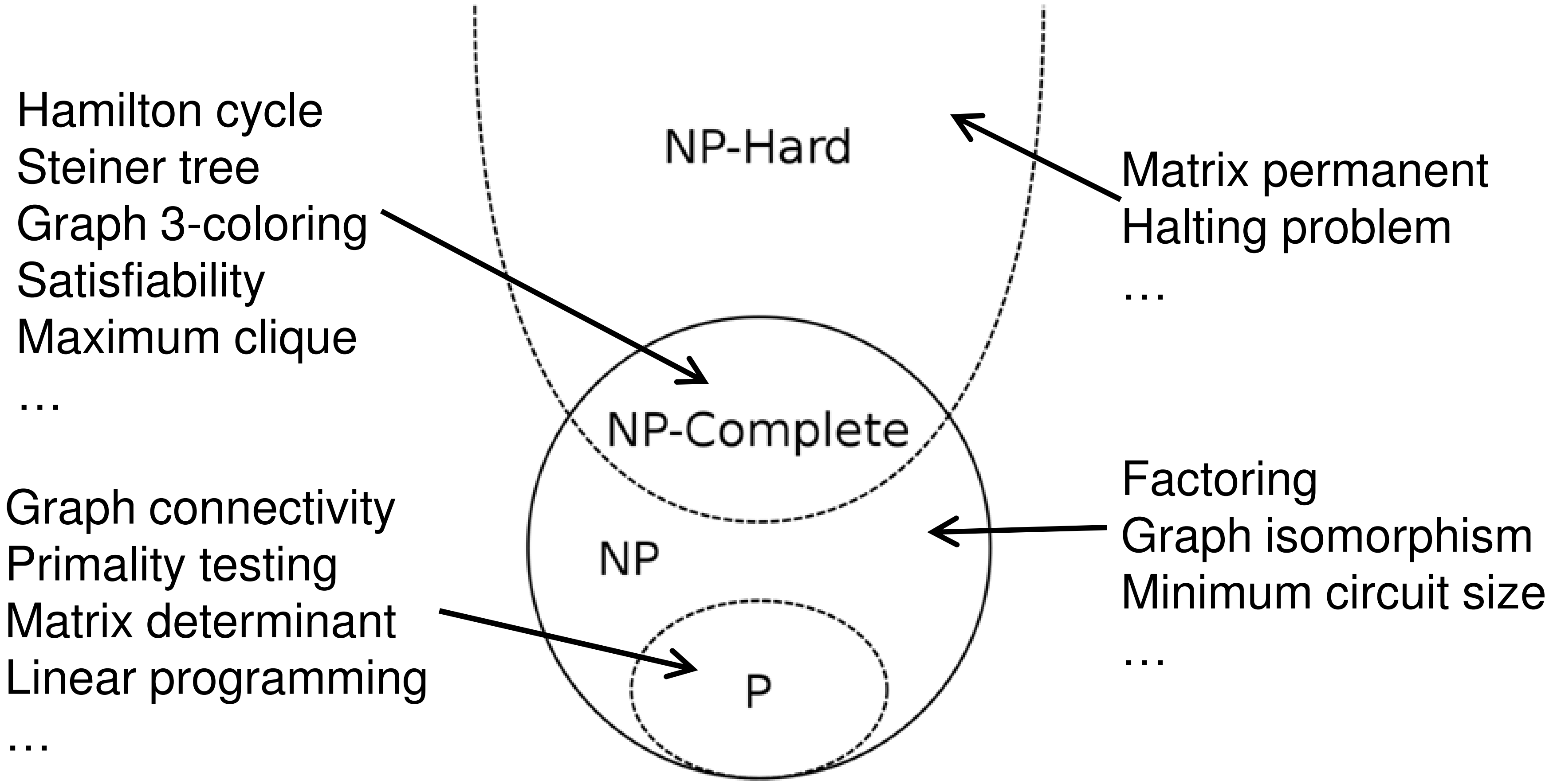
NP-Hard Problems

- NP-hard problems do not necessarily belong to NP.
- An NP-hard problem that is in NP is said to be NP-complete.



NP-Hard Problems

- Some examples



Hamilton cycle
Steiner tree
Graph 3-coloring
Satisfiability
Maximum clique
…

Matrix permanent
Halting problem
…

Graph connectivity
Primality testing
Matrix determinant
Linear programming
…

Factoring
Graph isomorphism
Minimum circuit size
…

Some NP-hard problems

- Sequencing and scheduling problems
  - such as flow-shop scheduling, job-shop scheduling, or open-shop scheduling.
- Assignment and location problems
  - such as quadratic assignment problem (QAP), generalized assignment problem (GAP), location facility, and the p-median problem.
- Grouping problems
  - such as data clustering, graph partitioning, and graph coloring.
- Routing and covering problems
  - such as vehicle routing problems (VRP), set covering problem (SCP), Steiner tree problem, and covering tour problem (CTP).
- Knapsack and packing/cutting problems, and so on.

NP-hard Problems

- Integer programming models belong in general to the NP-hard class.
- Unlike LP models, IP problems are difficult to solve because the feasible region is not a convex set.

Complexity of Problems

- To become a good algorithm designer, you must understand the basics of the theory of NP-completeness.
- If you can establish a problem as NP-hard, you provide good evidence for its intractability.
- As an engineer, you would then do better spending your time developing an approximation algorithm, rather than searching for a fast algorithm that solves the problem exactly.
- Thus, it is important to become familiar with this remarkable class of problems.

NP-naming convention

- **NP-complete -** means problems that are 'complete' in NP, i.e. the most difficult to solve in NP
- **NP-hard** - stands for 'at least' as hard as NP (but not necessarily **in** NP);
- **NP-easy** - stands for 'at most' as hard as NP (but not necessarily **in** NP);
- **NP-equivalent** - means equally difficult as NP, (but not necessarily **in** NP);

References

References

- Thomas H. Cormen et al., **Introduction to Algorithms**, Second Edition, The MIT Press, 2001. (Chapter 34)
- El-Ghazali Talbi, **Metaheuristics : From Design to Implementation**, John Wiley & Sons, 2009. (Chapter 1)

The End