15 - Theory of P, NP, NP-Complete

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

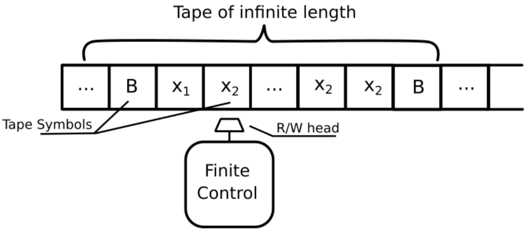
Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 21-25 March 2022

Table of contents

Turing machine

Turing Machine:



A Turing machine is a mathematical model of computation that defines an abstract machine that manipulates symbols on a strip of tape according to a table of rules. Despite the model's simplicity, given any computer algorithm, a Turing machine capable of implementing that algorithm's logic can be constructed. (*wikipedia*)

Turing Machine (2)

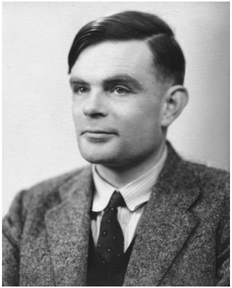
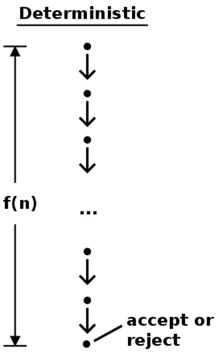


Figure: Alan Mathison Turing, (23 June 1912 7 June 1954), an English mathematician, logician, cryptanalyst, and computer scientist.

Deterministic algorithm (1)

Definition

A deterministic algorithm is an algorithm that, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states



Deterministic algorithm (2)

Example: Sequential search.

Given an array of n integers $(a_1, a_2, \ldots, a_n)$. We want to find the maximum of the array.

Algorithm 1 Finding maximum of an array of integers

```
procedure MAX(A[1..n])
    max ← a_1
    for i = 2 to n do
        if a_i > max then
            max ← a_i
        end if
    end for
end procedure
```

Time complexity: $\mathcal{O}(n)$

Nondeterministic algorithm (1)

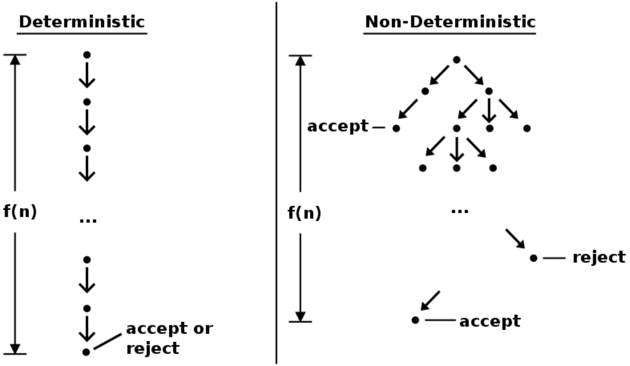
Definition

A nondeterministic algorithm is a two-stage procedure that takes as its input an instance I of a decision problem and does the following.

- Nondeterministic ("guessing") stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I.
- Deterministic ("verification") stage: A deterministic algorithm takes both I and S as its input and outputs yes if S represents a solution to instance I. (If S is not a solution to instance I, the algorithm either returns no or is allowed not to halt at all.)

A nondeterministic algorithm solves a decision problem if and only if "*for every yes instance of the problem it returns yes on some execution*", i.e. we require a nondeterministic algorithm to be capable of 'guessing' a solution at least once and to be able to verify its validity. (Also, no 'yes' output on instance with 'no' answer)

Nondeterministic algorithm (2)



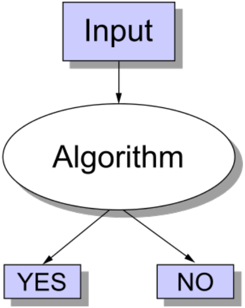
Nondeterministic algorithm (3)

Example: Nondeterministic Turing Machine

Decidability and undecidability

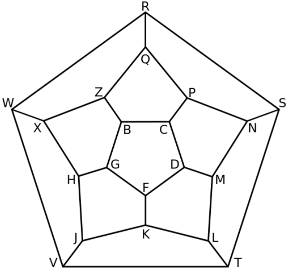
Decision problems

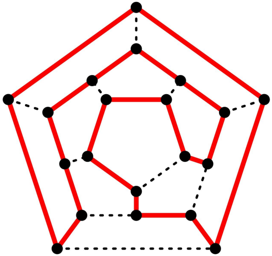
Decision problems are problems with yes/no answers.



Example of decision problems

1. Hamiltonian cycle problem: Determine whether a given graph has a Hamiltonian circuit a path that starts and ends at the same vertex and passes through all the other vertices exactly once.





2. Decision version of TSP problem: Given a positive integer ℓ, the task is to decide whether the graph has a tour of at most ℓ.

Decidable & undecidable problems

Does *every* decision problem can be solved in polynomial time?

Definition

Decision problems that can be solved by an algorithm is called decidable problems. Decision problems that cannot be solved at all by any algorithm is called undecidable problems.

- **Decidable problem**: if there is a Turing machine which halts on every input with an answer "yes" or "no"
- **Undecidable problem**: if we cannot construct an algorithm that can answer the problem correctly in finite time, i.e. there will always be a condition that will lead the Turing Machine into an infinite loop without providing an answer at all

Halting problem (1)

Example of decidable problems:

Example of undecidable problems: Halting problem

Problem (Halting problem (Turing, 1936))

Given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

Halting problem (2)

Halting problem is undecidable.

Proof.
For a contradiction, assume that A is an algorithm that solves the halting problem. That is, for any program P and input I:

$$A(P, I) = \begin{cases} 1, & \text{if program } P \text{ halts on input } I \\ 0, & \text{if program } P \text{ does not halt on input } I \end{cases}$$

Consider program P as an input to itself and use the output of algorithm A for pair (P, P) to construct a program Q as follows:

$$Q(P) = \begin{cases} \text{halts}, & \text{if } A(P, P) = 0, \text{ i.e., if program } P \text{ does not halt on input } P \\ \text{does not halt}, & \text{if } A(P, P) = 1, \text{ i.e., if program } P \text{ halts on input } P \end{cases}$$

Substituting Q for P gives:

$$Q(Q) = \begin{cases} \text{halts}, & \text{if } A(Q, Q) = 0, \text{ i.e., if program } Q \text{ does not halt on input } Q \\ \text{does not halt}, & \text{if } A(Q, Q) = 1, \text{ i.e., if program } Q \text{ halts on input } Q \end{cases}$$

Hence, this is a contradiction because neither of the two outcomes for program Q is possible.

Tractability

P problems

Definition (Polynomial problems)

Class P is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called polynomial.

Example of polynomials problems

- Searching $\rightarrow$ $T(n) = \mathcal{O}(n)$, $T(n) = \mathcal{O}(\log n)$
- Sorting $\rightarrow$ $T(n) = \mathcal{O}(n^2)$, $T(n) = \mathcal{O}(n \log n)$
- Matrix multiplication $\rightarrow$ $T(n^3) = \mathcal{O}(n)$, $T(n) = \mathcal{O}(n^{2.83})$

Example of non-polynomials problems

- TSP $\rightarrow$ $T(n) = \mathcal{O}(n!)$
- Integer knapsack problem $\rightarrow$ $T(n) = \mathcal{O}(2^n)$
- Graph coloring problem, etc.

Tractable & intractable problems

Definition

We say that an algorithm solves a problem in polynomial time if its worst-case time efficiency belongs to $\mathcal{O}(p(n))$ where $p(n)$ is a polynomial of the problems input size n.

(*Note that since we are using big-oh notation here, problems solvable in, say, logarithmic time are solvable in polynomial time as well.*)

Problems that can be solved in polynomial time are called tractable, and problems that cannot be solved in polynomial time are called intractable.

- Polynomial-time: $\mathcal{O}(n^k)$, $\mathcal{O}(1)$, $\mathcal{O}(n \log n)$
- Not in polynomial-time: $\mathcal{O}(2^n)$, $\mathcal{O}(n!)$, $\mathcal{O}(n^n)$

Decidable but intractable problems

- Hamiltonian circuit problem: Determine whether a given graph has a Hamiltonian circuit a path that starts and ends at the same vertex and passes through all the other vertices exactly once.
- Traveling salesman problem: Find the shortest tour through n cities with known positive integer distances between them.
- Knapsack problem: Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.
- Partition problem Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.
- Graph-coloring problem: For a given graph, find its *chromatic number*, which is the smallest number of colors that need to be assigned to the graphs vertices so that no two adjacent vertices are assigned the same color.
- Integer linear programming problem: Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and inequalities.

NP problems

NP problems

NP: non-deterministic polynomial (**not** "non-polynomial time algorithm")

Definition (Nondeterministic polynomial algorithms)

Non-deterministic polynomial-time algorithm is a non-deterministic algorithm whose verification stage can be done in polynomial time.

Poly-time verification means:

- Provided a solution candidate, we can check whether the answer is correct/wrong in poly-time.
- Note that this is equivalent to "finding solution in poly-time"

Example: In decision-version of TSP, given TSP solution of a graph, a positive integer k, we can check in poly-time if the solution is a TSP and has weight $\leq k$.

NP problems

Definition (Class NP)

Class NP is the class of decision problems that can be solved by *nondeterministic polynomial algorithms*. This class of problems is called nondeterministic polynomial.

Remark.

- Most decision problems are in NP, and $P \subseteq NP$
  - because, if a problem is in P, we can use the deterministic poly-time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simply ignores string S generated in its nondeterministic ("guessing") stage.
- $NP \not\subseteq P$, because some problems are in NP but not in P.
  - Examples: Hamiltonian circuit problem, decision version of TSP, knapsack, and graph coloring problems, etc.
- Some (rare) problems are not in NP. Example: Halting problem.

Is $P = NP$?

The most important open question of theoretical computer science:

$$P \stackrel{?}{=} NP$$

- $P = NP$ would imply that each of many hundreds of difficult combinatorial decision problems can be solved by a poly-time algorithm (*this is still open despite the efforts of many computer scientists over many years*).
- Many well-known decision problems are known to be "NP-complete" $\rightarrow$ more doubts on the possibility that $P = NP$.

Millennium Prize Problems

Seven well-known mathematical problems selected by the Clay Mathematics Institute in 2000. The Clay Institute has pledged a US$1 million prize for the correct solution of any of the problems.

1. Birch and Swinnerton-Dyer conjecture
2. Hodge conjecture
3. Navier-Stokes existence and smoothness
4. P versus NP problem
5. Poincaré conjecture (solved)
6. Riemann hypothesis
7. Yang-Mills existence and mass gap

NP-Complete problems (1): definition

Informally, an *NP*-complete problem is a problem in *NP* that is as difficult as any other problem in this class because, by definition, any other problem in *NP* *can be reduced* to it in polynomial time.

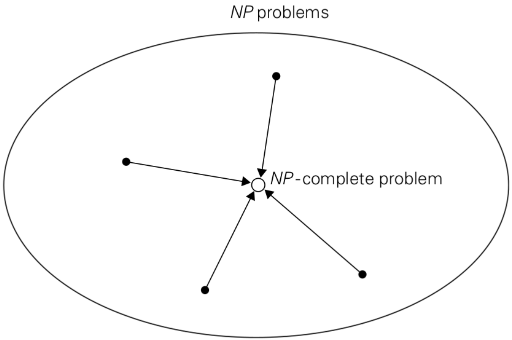


Figure: Notion of an *NP*-complete problem. Polynomial-time reductions of *NP* problems to an NP-complete problem are shown by arrows.

NP-Complete problems (2): polynomial reduction

Definition (Polynomially reducible problems)

A decision problem D_1 is said to be polynomially reducible to a decision problem D_2, if there exists a function t that transforms instances of D_1 to instances of D_2 such that:

- t maps all yes instances of D_1 to yes instances of D_2, and all no instances of D_1 to no instances of D_2
- t is computable by a poly-time algorithm, i.e. $t \in NP$

Implication: If a problem D_1 is polynomially reducible to some problem D_2 that can be solved in poly-time, then problem D_1 can also be solved in poly-time.

$$D_1 \xrightarrow[\text{in } P]{\text{reduced to}} \underset{\text{in } P}{D_2}$$

NP-Complete problems (3): *NPC* definition

Definition (*NP*-Complete problem)

A decision problem D is said to be *NP*-complete if:

- it belongs to class NP
- every problem in NP is polynomially reducible to D

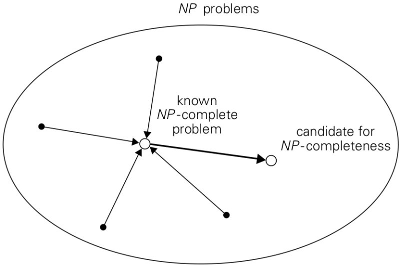


Figure: Proving *NP*-completeness by reduction

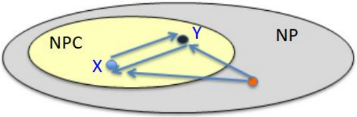
- If X is NPC and X is poly-time solvable, then all NP problems are poly-time solvable;
- i.e. if X is poly-time solvable, then $P = NP$.

List of NP-Complete problems

- Boolean satisfiability problem (SAT)
- *Decision*-version TSP
- Hamiltonian cycle problem
- Partition problem
- Clique problem
- Decision-version of graph coloring problem
- Vertex cover problem
- Decision-version of Knapsack problem

Properties of *NP*-complete problems

- A problem X is *NPC* if any problem in *NP* can be reduced (transformed) to X in poly-time.
- Two problems X and Y in *NPC* can be reduced one to each other in poly-time.
  - X can be reduced to Y in poly-time
  - Y can be reduced to X in poly-time



NP-Complete problems (5): polynomial reduction

How to show that a problem X is *NPC*?

- Show that X is *NP*
- Choose a problem Y from a collection of *NPC* problems
- Construct a reduction algorithm that reduces an instance of problem Y to an instance of problem Z.

NP-Complete problems (6): polynomial reduction

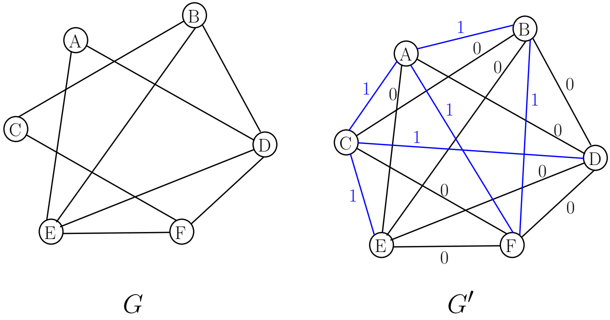
Example: The Hamiltonian circuit problem is polynomially reducible to the decision version of TSP

- **Hamiltonian circuit problem**: Determine whether a given graph has a Hamiltonian circuit – a path that starts and ends at the same vertex and passes through all the other vertices exactly once.
- **TSP-decision problem**: Given a graph and distance between pair of vertice, and a positive integer ℓ, the task is to decide whether the graph has a tour of at most ℓ.

We map a graph G of a given instance of the Hamiltonian circuit problem to a complete weighted graph G representing an instance of the TSP.

Reduction:

- Assign 0 as the weight to each edge in G and adding an edge of weight 1 between any pair of nonadjacent vertices in G.



- G **has Hamiltonian cycle if there exists a cycle in** G' **passing through all vertices exactly once, and that has a length** ≤ 0 (i.e. has a solution for the instance of TSP where $k = 0$).

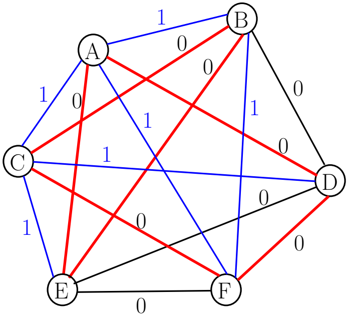
1. **If there is a cycle that passes through all vertices exactly once, and has length** ≤ 0 **in** G', the cycle contains only edges that were originally present in G. (The new edges in G' have weight 1 and hence cannot be part of a cycle of length ≤ 0.)

   $\Rightarrow$ There exists a Hamiltonian cycle in G

2. **If there exists a Hamiltonian cycle in** G, it forms a cycle in G' with length $= 0$, since a weights of all the edges is 0.

   $\Rightarrow$ There exists a solution for TSP in G' with length ≤ 0.

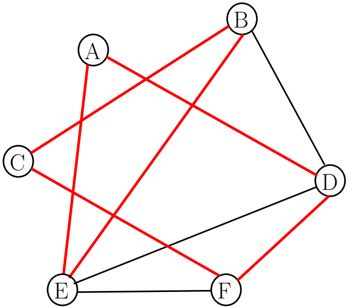
Example:



G'

Figure: G' has a cycle passing through all vertices exactly once with length ≤ 0.

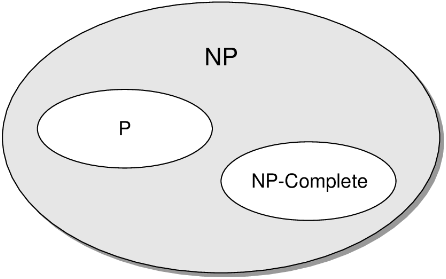
Example:



G

Figure: G' has a cycle passing through all vertices. This is a Hamiltonian cycle in G

P, NP, and NP-Complete diagram



CNF-satisfiability problem

- x_1, x_2, x_3, and x_4 are Boolean variables to be assigned (value 0 or 1)
- $\neg$ means negation (logical *not*)
- $\wedge$ means conjunction (logical *and*)
- $\vee$ means disjunction (logical *or*)
- A literal is a variable or its negation, e.g.: x_i and $\neg x_i$
- A clause is a disjunction ($\wedge$) of literals, e.g.: $x_i \vee x_j$
- Conjunctive Normal Form (CNF) is a conjunction of clauses

  Example: $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3 \vee \neg x_4) \wedge (\neg x_1 \vee x_4)$

CNF-satisfiability problem

Definition

The Satisfiability Problem (SAT) is a classic combinatorial problem. Given a Boolean formula of n variables:

$$f(x_1, x_2, \dots, x_n)$$

The problem is **to find such values of the variables, on which the formula takes on the value True**.

The CNF Satisfiability Problem (CNF-SAT) is a version of the Satisfiability Problem, where the Boolean formula above is specified in the Conjunctive Normal Form (CNF).

CNF-satisfiability problem

Input: Expression over Boolean variables in conjunctive normal form (CNF).

Question: Is the expression satisfiable? i.e., can we give each variable a value (true or false) such that the expression becomes true?

CNF-satisfiability problem

Input: Expression over Boolean variables in conjunctive normal form (CNF).

Question: Is the expression satisfiable? i.e., can we give each variable a value (true or false) such that the expression becomes true?

Example: $(x_1 \lor x_2) \land (\neg x_2 \lor x_3 \lor \neg x_4) \land (\neg x_1 \lor x_4)$

The formula is satisfiable because on $x_1 = \mathit{True}$, $x_2 = \mathit{False}$, $x_3 = \mathit{False}$, and $x_4 = \mathit{True}$, it takes on the value True.

Check it!

CNF-satisfiability problem

Theorem (Cook-Levin Theorem)

CNF-Satisfiability is NP-complete.

Proof. See
`https://en.wikipedia.org/wiki/CookLevin_theorem`

- Most well known is Cooks proof, using Turing machine characterization of *NP*.
- It design a Turing machine that verifies yes-instances of SAT

NP-Hard problems

Definition (NP Hard problem)

A decision problem H is *NP-hard* if for every problem L in NP, there is a polynomial-time many-one reduction from L to H.

- A problem is NP-hard if an algorithm for solving it can be translated into one for solving any NP-problem.
- NP-hard therefore means "at least as hard as any NP-problem" although it might, in fact, be harder.
- NP-hard problems often have exponential-time complexity.

Example: (Non-decision problem) of TSP

Remark. If $P \neq NP$, then NP-hard problems could not be solved in polynomial time.

Diagram of complexity classes