Data Structures and Algorithms – W15

NP-completeness

Contents

- Formal notations
- Problem classes:
  - P class
  - NP class
  - NP-complete class
  - NP-hard class

Undecidable problems

- The **halting problem** is a decision problem about properties of computer programs on a fixed Turing-complete model of computation. The question is, given a program and an input to the program, whether the program will eventually halt when run with that input. In this abstract framework, there are no resource limitations of memory or time on the program's execution; it can take arbitrarily long, and use arbitrarily much storage space, before halting. The question is simply whether the given program will ever halt on a particular input.
- The halting problem was one of the first problems proved **undecidable**, which means there is **no computer program** capable of correctly answering the question for **all possible** inputs
- We cannot even solve such a „simple” problme like Colatz problem (another name is „3n+1” problem).

Decision problems vs optimization problems

- *Optimization problems*: each feasible solution has an associated value, and we wish to find the feasible solution with the best value
  - SHORTEST-PATH problem, we are given an undirected graph *G* and vertices *u* and *v*, and we wish to find the path from *u to v* that uses the fewest edges
- *Decision problems*: the answer is simply „yes” or „no” (1 or 0). **NP-completeness** applies directly to decision problems.
  - PATH problem, we are given a undirected graph *G*, vertices *u* and *v*, and an integer *k*, a path exists from *u* to *v* consisting of at most *k* edges
- If an optimization problem is easy its related decision problem is easy as well.
- If an decision problem is hard its related optimization problem is hard too.

Abstract problem

- An ***abstract problem Q*** to be a binary relation on a set *I* of problem instances and a set *S* of problem solutions.
- For example:
  - an instance for SHORTEST-PATH is a triple consisting of a graph and two vertices.
  - a solution is a sequence of vertices in the graph, with perhaps the empty sequence denoting that no path exists.
  - the problem SHORTEST-PATH itself is the relation that associates each instance of a graph and two vertices with a shortest path in the graph that connects the two vertices. Since shortest paths are not necessarily unique, a given problem instance may have more than one solution.

Abstract decision problems

- Problems those having a yes/no solution. In this case, we can view an abstract decision problem as a function that maps the instance set *I* to the solution set {0, 1}.
- For example:
  - a decision problem related to SHORTEST-PATH is the problem PATH: if $i = \langle G, u, v, k \rangle$ is an instance of the decision problem PATH, then PATH(*i*) = 1(yes) if a shortest path from *u* to *v* has at most *k* edges, and PATH(*i*) = 0 (no) otherwise.

- The abstract problem is a mathematical problem. In the case of solving on a computer, you need to concretise how we will represent the problem and its solution.

Encodings

- An **encoding** of a set *S* of abstract objects is a **mapping** *e* from *S* to the set of **binary strings**.
  - encoding the natural numbers *N* = {0, 1, 2, 3, 4,...} as the strings {0, 1, 10, 11, 100,...}
  - Even a compound object can be encoded as a binary string by combining the representations of its constituent parts. Polygons, graphs, functions, ordered pairs, programs - all can be encoded as binary strings
- Unary encoding - "expensive" encodings
  - e.a {1, 11, 111, 1111, ….}
- Other encoding with different base are polynomially related

Definitions

- We call a problem whose instance set is the set of binary strings a ***concrete problem.*** We say that an algorithm solves a concrete problem ***in time O(T(n))*** if, when it is provided a problem instance i of length $n=|i|$, the algorithm can produce the solution in $O(T(n))$ time. A concrete problem is ***polynomial-time solvable,*** therefore, if there exists an algorithm to solve it ***in time $O(n^k)$*** for some constant k.

- The ***complexity class P*** is the set of concrete decision problems that are ***polynomial-time*** solvable.

- All problems presented so far in this course belong to the class of complexity P.

Formal language

- An **alphabet** *S* is a finite set of symbols.
- A **language** *L* **over** *S* is any set of strings made up of symbols from *S*.
  - For example, if *S* = {0, 1}, the set *L* = {10, 11, 101, 111, 1011, 1101, 10001,...} is the language of binary representations of prime numbers.
- We denote the empty string by ε, and the empty language by Ø.
- The language of all strings over *S* is denoted *S**.
  - For example, if *S* = {0, 1}, then *S** = {ε, 0, 1, 00, 01, 10, 11, 000,...} is the set of all binary strings.
- Every language *L* over *S* is a subset of *S**

Formal language (cont.)

- From the point of view of language theory, the set of instances for any decision problem *Q is simply the set* S*, where S = {0, 1}. Since *Q* is entirely characterized by those problem instances that produce a 1 (yes) answer, we can view *Q* as a language *L* over *S* = {0, 1}*, where:*

$$L=\{ x \in S^* : Q(x)=1\}$$

- We say that an algorithm *A* **accepts a string *x*$\in$ *{0, 1}**** if, given input *x*, the algorithm's output *A*(*x*) is 1. The language **accepted by an algorithm *A*** is the set of strings $L = \{x \in \{0, 1\}^*: A(x) = 1\}$, that is, the set of strings that the algorithm accepts. An algorithm A **rejects a string** *x* if *A*(*x*) = 0

Accept/decide a language

- Even if language L is accepted by an algorithm A, the algorithm will not necessarily reject a string $x \notin L$ provided as input to it. For example, the algorithm may loop forever. A language L is **decided by an** algorithm A if every binary string in L is accepted by A and every binary string not in L is rejected by A. A language L is **accepted in polynomial time by an algorithm *A*** if it is accepted by A and if in addition there is a constant k such that for any length-n string $x \in L$, algorithm A accepts x in time $O(n^k)$. A language L is **decided in polynomial time by an algorithm *A*** if there is a constant k such that for any length-n string $x \in \{0, 1\}^*$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$. Thus, to accept a language, an algorithm need only worry about strings in L, but to decide a language, it must correctly accept or reject every string in $\{0, 1\}^*$

- An alternative definition of the complexity class P:

$$P = \{L \subseteq \{0, 1\}^* : \text{there exists an algorithm A that decides L in polynomial time}\}$$

Verification algorithm

- We define a ***verification algorithm as being a two-argument algorithm A, where one argument is an*** ordinary input string *x and the other is a binary string y called a* ***certificate. A two-argument algorithm A verifies an input string x if there exists a certificate y such that A(x, y) = 1. The language verified by a*** verification algorithm *A is*

$$L = \{x \in \{0, 1\}^* : \text{there exists } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}.$$

- An algorithm *A verifies a language L if for any string* $x \in L$*, there is a certificate y that A can use* to prove that $x \in L$*. Moreover, for any string* $x \notin L$*, there must be no certificate proving that* $x \in L$*.*
  - A **hamiltonian cycle of an undirected graph *G*=(*V*, *E*)** is a simple cycle that contains each vertex in *V* . A graph that contains a hamiltonian cycle is said to be **hamiltonian; otherwise, it is nonhamiltonian**
  - *For* example, in the hamiltonian-cycle problem, the certificate is the list of vertices in the hamiltonian cycle. If a graph is hamiltonian, the hamiltonian cycle itself offers enough information to verify this fact. Conversely, if a graph is not hamiltonian, there is no list of vertices that can fool the verification algorithm into believing that the graph is hamiltonian, since the verification algorithm carefully checks the proposed "cycle" to be sure.

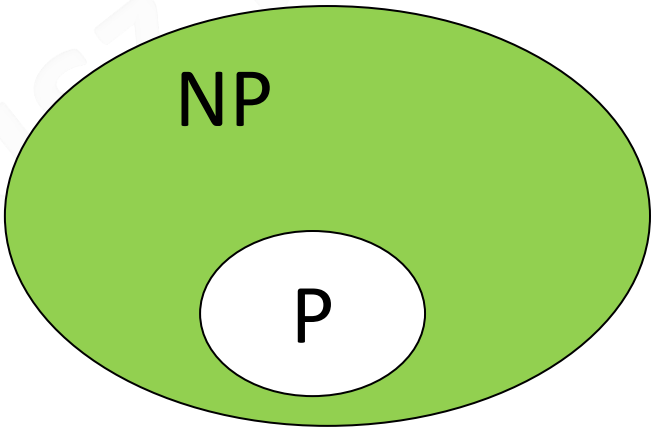
The complexity class NP

- The **complexity class NP** is the class of languages that can be **verified** by a **polynomial-time algorithm.**
- A language L belongs to NP if and only if there exist a two-input polynomial-time algorithm A and constant c such that $L = \{x \in \{0, 1\}^* : \text{there exists a certificate } y \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}$.
- We say that algorithm A **verifies language L in polynomial time**

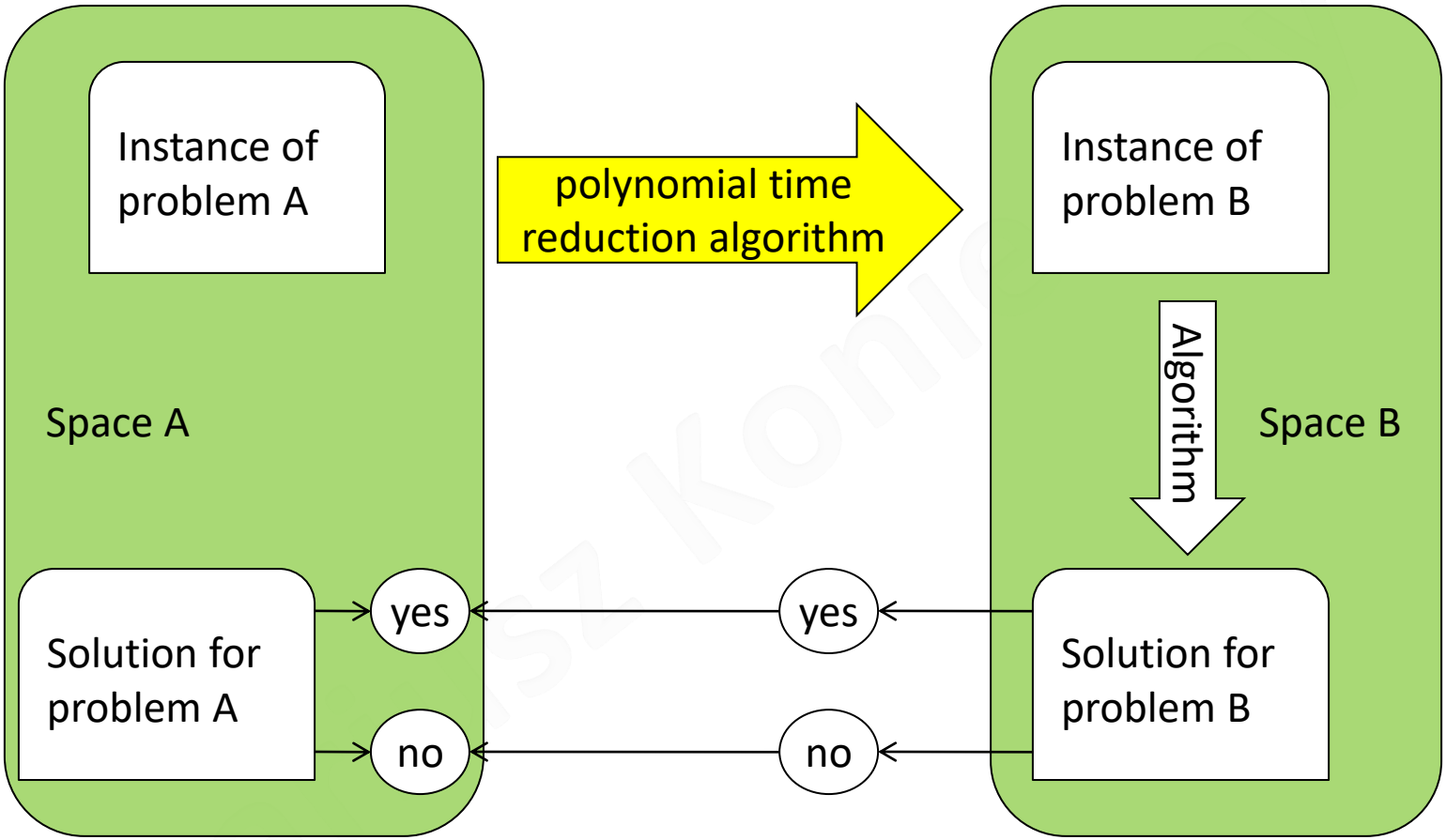
The Question

P = NP ?

(Probably P≠NP)



Reducibility



Reducibility (cont.)

- A language L_1 is **polynomial-time reducible to a language L_2,** written $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f:\{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0, 1\}^*$,

  $x \in L_1$ if and only if $f(x) \in L_2$

- We call the function f the **reduction function,** and a polynomial-time algorithm F that computes f is called a **reduction algorithm**.

Lemma

If $L_1, L_2 \in \{0,1\}^*$ are languages such that $L_1 \leq_P L_2$, then $L_2 \in P$ implies $L_1 \in P$

NP-completness

- A language $L \in \{0, 1\}^*$ is **NP-complete if**

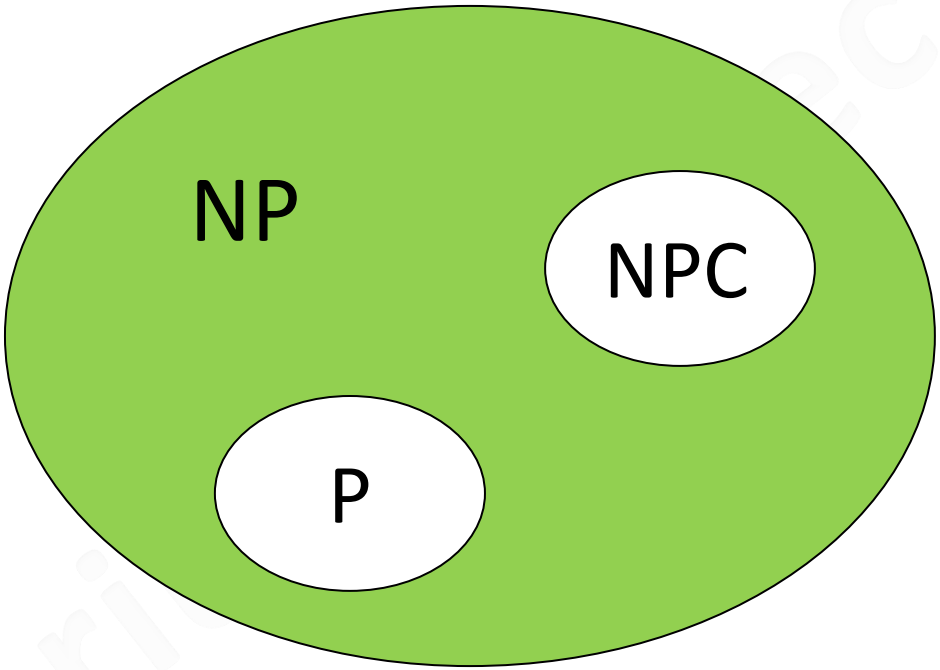
1. $L \in$ NP, and
2. $L' \leq_P L$ for every $L' \in$ NP.

- If a language L satisfies property 2, but not necessarily property 1, we say that L is **NP-hard.**
- We also define NPC to be the class of NP-complete languages.

Theorem

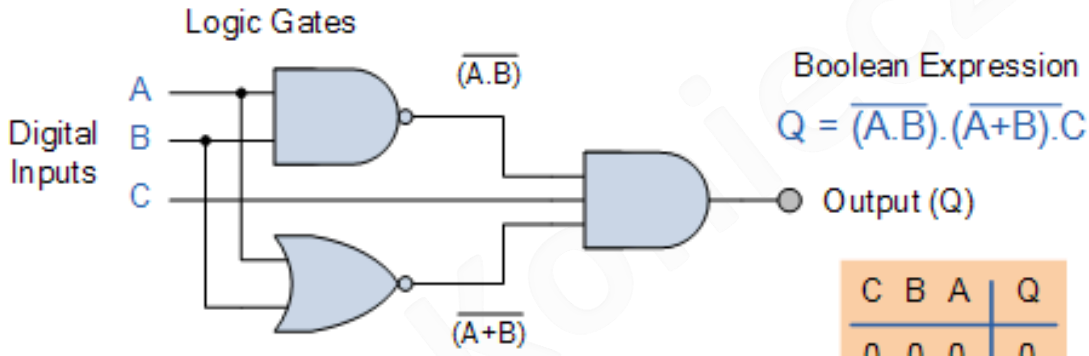
If any NP-complete problem is polynomial-time solvable, then P = NP. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

Probably relationship



NP-complete problems 1/8

- The circuit-satisfiability problem
  - Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?



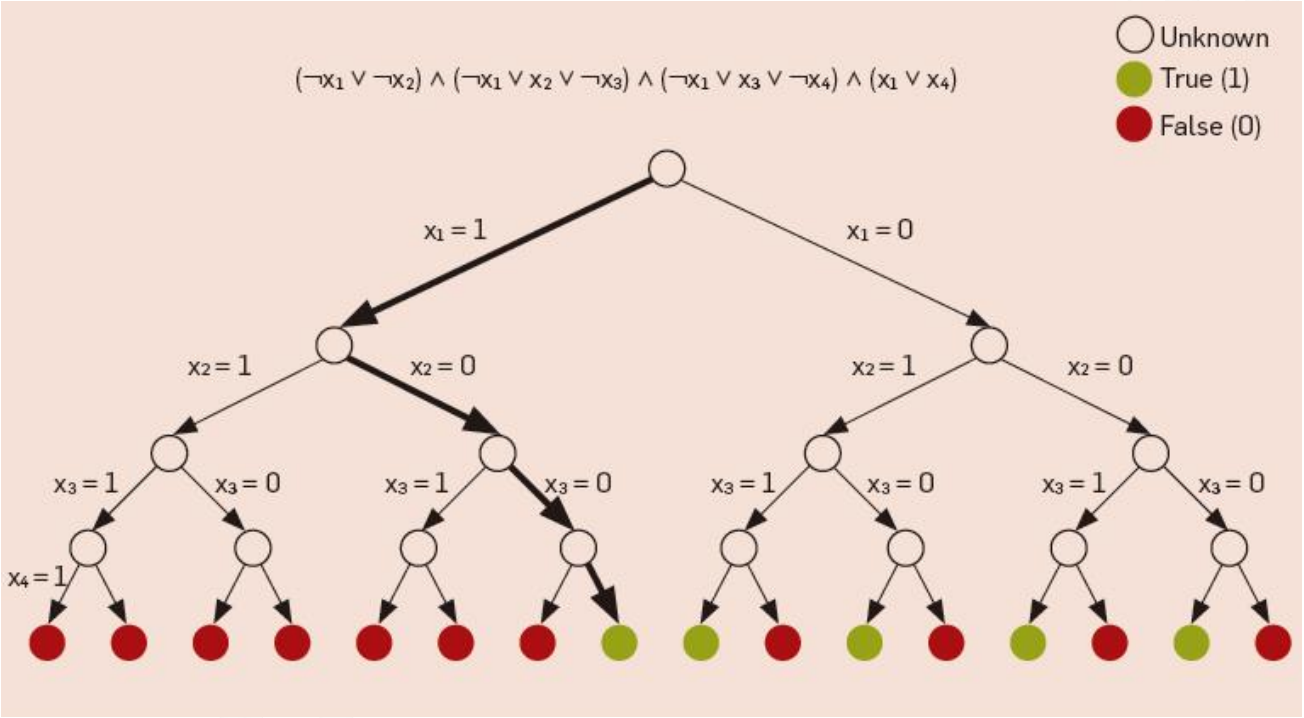
Typical Truth Table

C	B	A	Q
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

https://www.electronics-tutorials.ws/combination/comb_1.html

NP-complete problems 2/8

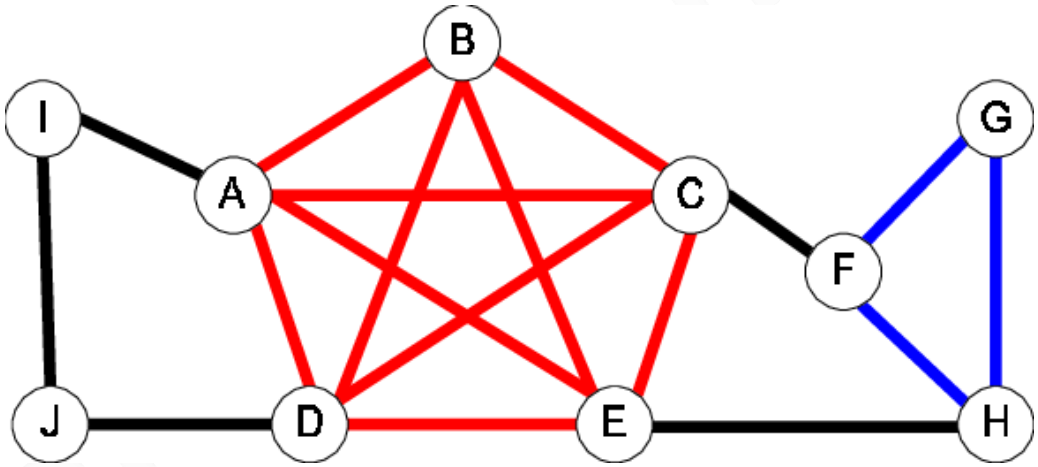
- Formula satisfiability (SAT) – first NP-complete problem
  - Given a boolean formula is satisfiable?



https://cacm.acm.org/magazines/2009/8/34498-boolean-satisfiability-from-theoretical-hardness-to-practical-success/fulltext

NP-complete problems 3/8

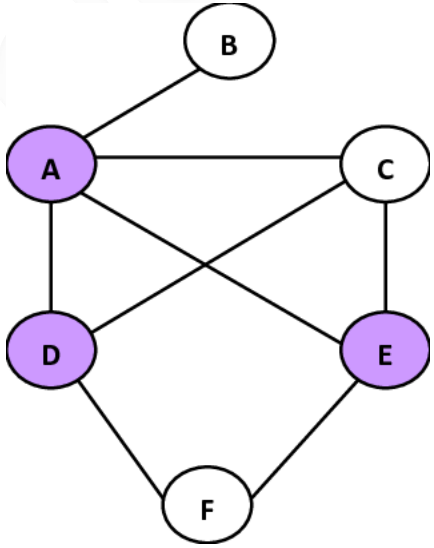
- The clique problem
  - A **clique** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The **size of a clique** is the number of vertices it contains. The **clique problem** is the optimization problem of finding **a clique of maximum size** in a graph. As a decision problem, we ask simply whether a clique of a given size k exists in the graph.



https://www.researchgate.net/figure/Graph-containing-a-clique-of-size-3-FGH-and-a-maximum-clique-of-size-5-ABCDE_fig2_221429651

NP-complete problems 4/8

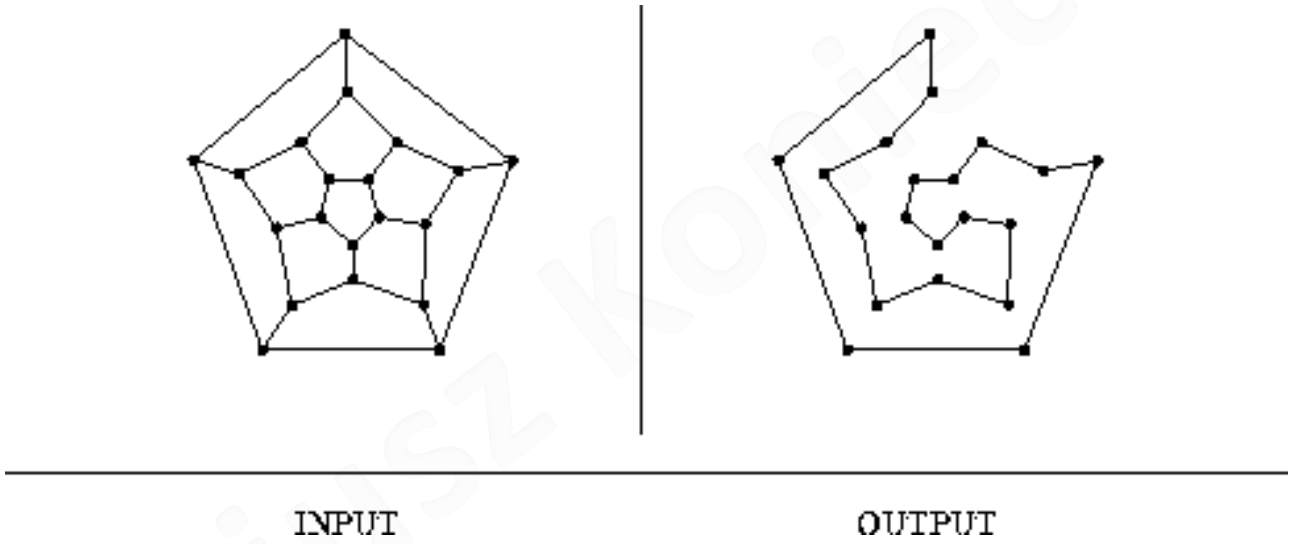
- The vertex-cover problem
  - A **vertex cover** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. The size of a vertex cover is the number of vertices in it. The vertex-cover problem is to find a vertex cover of minimum size in a given graph.



https://www.researchgate.net/figure/An-undirected-graph-G-the-shadowed-nodes-represent-the-vertex-cover_fig1_263008838

NP-complete problems 5/8

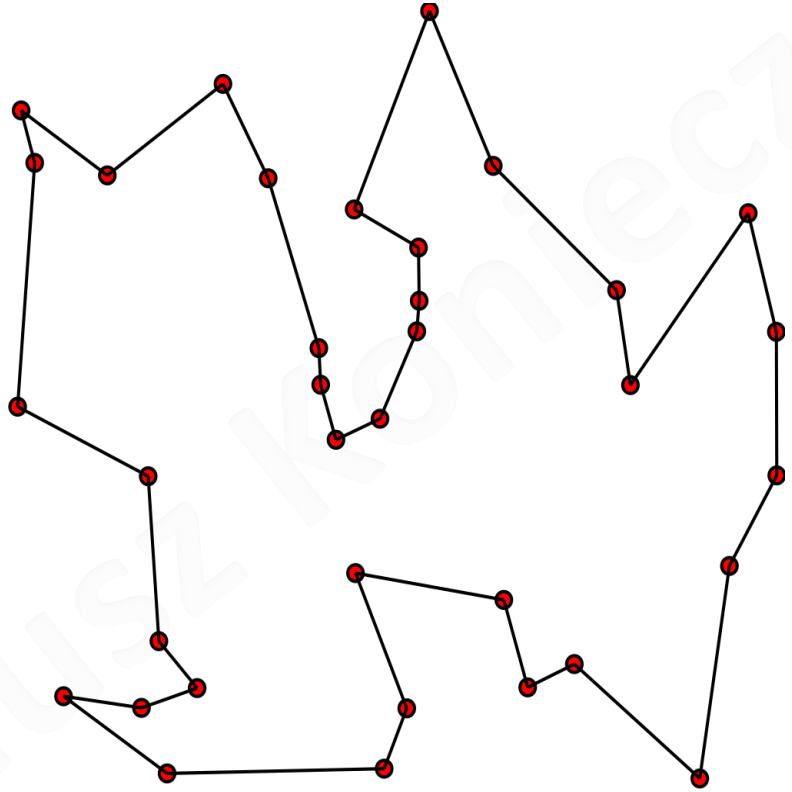
- The hamiltonian-cycle problem – a cycle in which we visit every vertex exactly once.



https://www8.cs.umu.se/kurser/TDBA77/VT06/algorithms/BOOK/BOOK4/NODE176.HTM

NP-complete problems 6/8

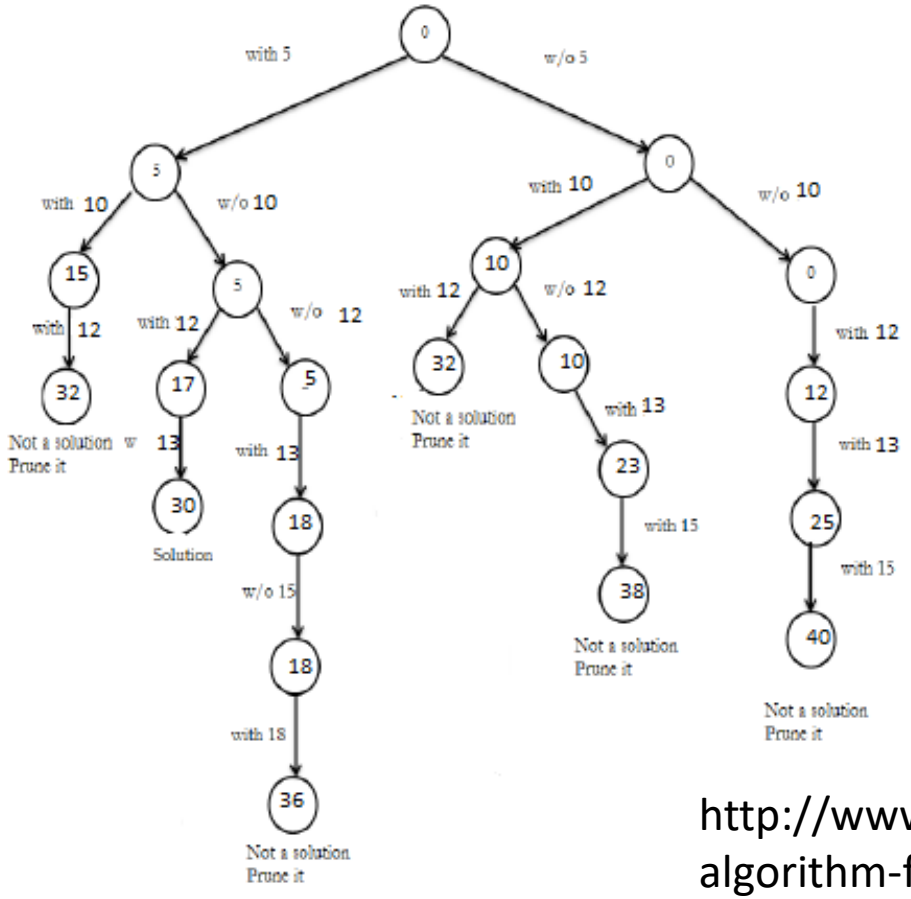
- The traveling-salesman problem – for weighted graph find a cycle with minimal weight, which visit every vertex exactly once.



https://en.wikipedia.org/wiki/Travelling_salesman_problem

NP-complete problems 7/8

- The subset-sum problem
  - In the subset-sum problem, we are given a finite set $S \subseteq N$ and a target $t \in N$. We ask whether there is a subset $S' \subseteq S$ whose elements sum to t.



S={5,10,12,13,15,18}
t=30

http://www.ques10.com/p/17687/write-an-algorithm-for-sum-of-subsets-solve-the-fo/

NP-complete problems 8/8

- There are many other problems which belong to NP-complete problems

- Sources to start:
  - Wikipedia List of NP-complete problems https://en.wikipedia.org/wiki/List_of_NP-complete_problems
  - A compendium of NP optimization problems: http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html
  - Graph of NP-Complete Problems: https://adriann.github.io/npc/npc.html

Solving NP-complete problems

- At present, all known algorithms for NP-complete problems require time that is superpolynomial in the input size, and it is unknown whether there are any faster algorithms.
- The following techniques can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:
  - **Approximation**: Instead of searching for an optimal solution, search for an "almost" optimal one.
  - **Randomization**: Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability.
  - **Restriction**: By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
  - **Parametrization**: Often there are fast algorithms if certain parameters of the input are fixed.
  - **Heuristic**: An algorithm that works "reasonably well" on many cases, but for which there is no proof that it is both always fast and always produces a good result.