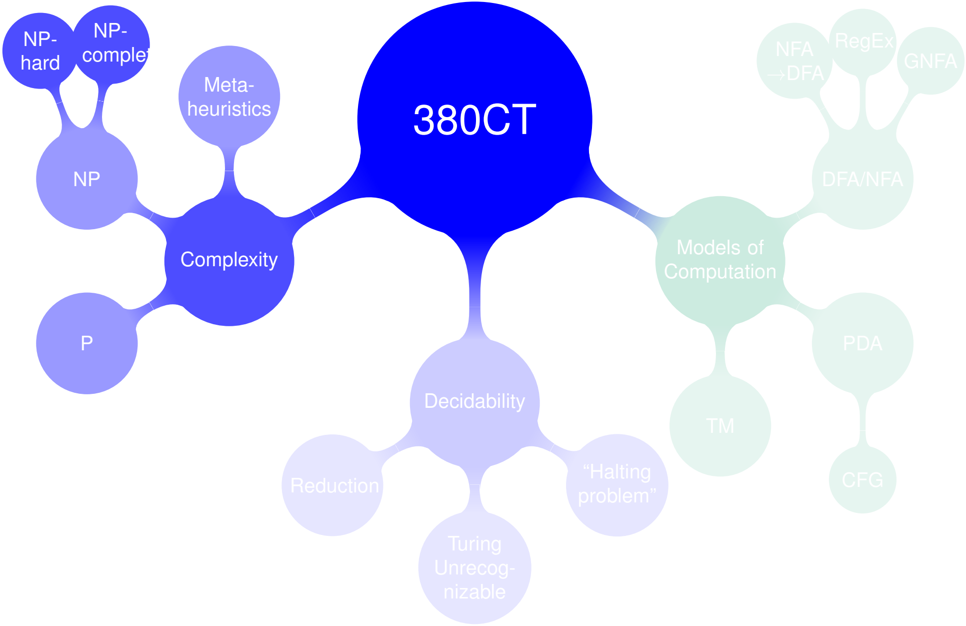
NP-Completeness

Dr Kamal Bentahar

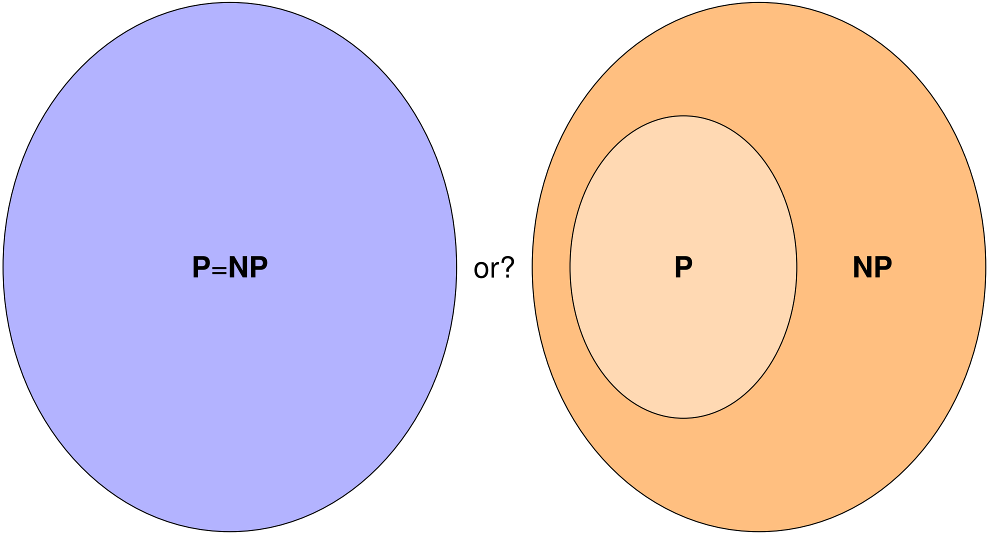
School of Computing, Electronics and Mathematics
Coventry University

Week 10 – 26/03/2019



380CT
Complexity
NP
NP-hard
NP-complete
Meta-heuristics
P
Decidability
Reduction
Turing Unrecog-nizable
"Halting problem"
Models of Computation
DFA/NFA
NFA →DFA
RegEx
GNFA
PDA
CFG
TM

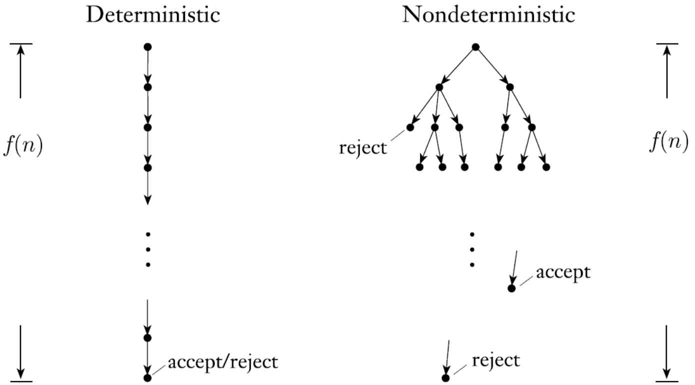
Last time...



Time complexity

The **time complexity** of a decider (TM that always halts) is the maximum number of steps that it makes on **any** input of length *n*.

For nondeterministic TMs consider **all the branches** of its computation.



The class **P**

Let $t \colon \mathbb{N} \to \mathbb{R}^+$ be a function.

Time complexity class

Define the **time complexity class** $TIME(t(n))$ to be the collection of all languages that are decidable by an $O(t(n))$ time TM.

The class **P**

P is the class of languages that are decidable in polynomial time on a deterministic TM.

$$\mathbf{P} = TIME(1) \cup TIME(n) \cup TIME(n^2) \cup TIME(n^3) \cup \cdots .$$

The class NP

Nondeterministic Polynomial time complexity class

$NTIME(t(n)) = \{$Language decided by an $O(t(n))$ time non-deterministic TM$\}$.

The class NP

NP is the class of languages that have polynomial time verifiers.

Equivalently: the class of languages that are decidable in polynomial time on a non-deterministic TM.

$$\mathbf{NP} = NTIME(1) \cup NTIME(n) \cup NTIME(n^2) \cup NTIME(n^3) \cup \cdots .$$

The satisfiability problem

Recall:

- Boolean variables (*True*/*False*)
- Logic operations ($\wedge, \vee, \neg$)
- Boolean formula, e.g.

$$x$$
$$x \wedge y$$
$$x \vee \neg y$$
$$\bar{x} \wedge (x \vee y)$$
$$(y \vee \bar{z}) \wedge (x \vee y)$$

- “**Satisfiable**” if formula can be *True* for some variables assignment.

The satisfiability problem (SAT)

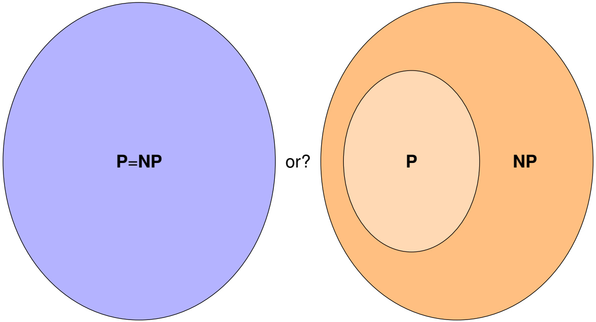
$$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$$

Link between *SAT* and the "**P** vs **NP**" question

Theorem (Cook 1971)

$$\mathit{SAT} \in \mathbf{P} \qquad \Longleftrightarrow \qquad \mathbf{P} = \mathbf{NP}$$

$\rightarrow$ if we can decide *SAT* efficiently then we can also efficiently decide any **NP** problem.



History of *SAT*

- **Stephen Cook (1971):** any problem in **NP** is transformable to *SAT* in polynomial time.
  Efficient solution to *SAT* $\implies$ Efficient solution to every problem in **NP**.
- **Richard Karp (1972):** listed 21 problems all transformable into each other in polynomial time.
- **Garey and Johnson (1979):** book *"Computers and Intractability: A Guide to the theory of NP-Completeness"* lists 320 problems, all transformable into each other in polynomial time.

- These "**NP-complete**" problems are the "hardest in **NP**."
- If any **NP-complete** problem is not in **P** then all of them are not in **P**.
  ($\implies$ **P** $\neq$ **NP**).

Reducibility

Idea: transform a given problem A to another S, such that an algorithm for S could be used as a **subroutine** to solve A.

Example

Let $S = \{x_1, \ldots, x_n\}$ be a set of integers.

Partition Problem (PP):

Can S be partitioned into two subsets with the same sum?

Subset-Sum Problem (SSP):

Can a subset of S sum to a given target t?

Given a set S for **PP**, we can transform it into an **SSP** instance as follows:

- Calculate $t = (x_1 + \cdots + x_n)/2$.
- The **SSP** instance is $\langle S, t \rangle$.

Solving **PP** has been **reduced** to solving **SSP**.

Reducibility

Polytime computable functions

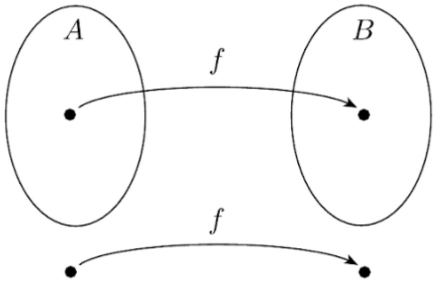
A function $f\colon \Sigma^* \to \Sigma^*$ is a polytime **computable function** if some polytime TM exists that, on input w, halts with just $f(w)$ on its tape.

The function f “efficiently transforms” the encodings of the two problems.

Polytime reducibility between problems

A language A is polytime **reducible** to a language B if a polytime computable function $f\colon \Sigma^* \to \Sigma^*$ exists such that

$$w \in A \quad \iff \quad f(w) \in B \quad \text{for all } w \in \Sigma^*$$



Reducibility

We write $A \leq_P B$ and read it "A **is (polytime) reducible to** B."
If B is known to have a polytime solution then we can construct a polytime solution to A too. So

$$A \leq_P B \text{ and } B \in \mathbf{P} \implies A \in \mathbf{P}$$

In other words, if A can be reduced to an "easy" problem B then A is also "easy."

NP-Completeness and **NP**-Hardness

NP-Hardness

A language is **NP-hard** if every problem in **NP** is polytime reducible to it.

NP-Completeness

A language is **NP-complete** if it satisfies two conditions:

1. it is in **NP**,
2. it is **NP-hard**.

The word "**complete**" is used to to mean that a solution to any problem can be applied to all others in the class.

Example **NP-complete** problems

The Cook-Levin Theorem

SAT is **NP-complete.**

- **Constraint Satisfaction**: SAT, 3SAT
- **Numerical Problems**: Subset Sum, Max Cut
- **Sequencing**: Hamilton Circuit, Sequencing
- **Partitioning**: 3D-Matching, Exact Cover
- **Covering**: Set Cover, Vertex Cover, Feedback Set, Clique Cover, Chromatic Number, Hitting Set
- **Packing**: Set Packing

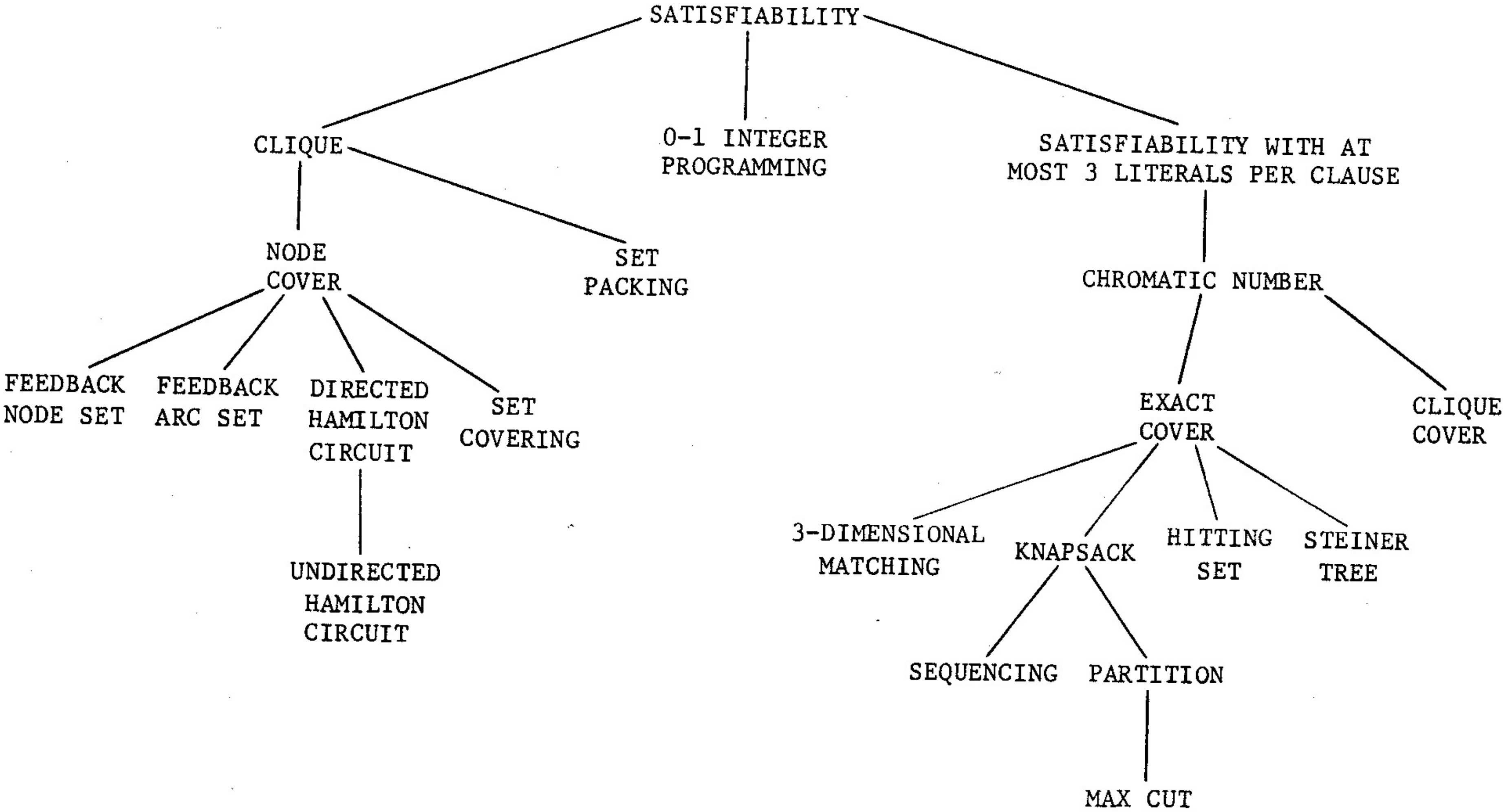


FIGURE 1 - Complete Problems

Main Theorem. All the problems on the following list are complete.

1. SATISFIABILITY
COMMENT: By duality, this problem is equivalent to determining whether a disjunctive normal form expression is a tautology.

2. 0-1 INTEGER PROGRAMMING
INPUT: integer matrix C and integer vector d
PROPERTY: There exists a 0-1 vector x such that $Cx = d$.

3. CLIQUE
INPUT: graph G, positive integer k
PROPERTY: G has a set of k mutually adjacent nodes.

4. SET PACKING
INPUT: Family of sets $\{S_j\}$, positive integer ℓ
PROPERTY: $\{S_j\}$ contains ℓ mutually disjoint sets.

5. NODE COVER
INPUT: graph G', positive integer ℓ
PROPERTY: There is a set $R \subseteq N'$ such that $|R| \leq \ell$ and every arc is incident with some node in R.

6. SET COVERING
INPUT: finite family of finite sets $\{S_j\}$, positive integer k
PROPERTY: There is a subfamily $\{T_h\} \subseteq \{S_j\}$ containing $\leq k$ sets such that $\cup T_h = \cup S_j$.

7. FEEDBACK NODE SET
INPUT: digraph H, positive integer k
PROPERTY: There is a set $R \subseteq V$ such that every (directed) cycle of H contains a node in R.

8. FEEDBACK ARC SET
INPUT: digraph H, positive integer k
PROPERTY: There is a set $S \subseteq E$ such that every (directed) cycle of H contains an arc in S.

9. DIRECTED HAMILTON CIRCUIT
INPUT: digraph H
PROPERTY: H has a directed cycle which includes each node exactly once.

10. UNDIRECTED HAMILTON CIRCUIT
INPUT: graph G
PROPERTY: G has a cycle which includes each node exactly once.

11. SATISFIABILITY WITH AT MOST 3 LITERALS PER CLAUSE
INPUT: Clauses $D_1, D_2, \ldots, D_r$, each consisting of at most 3 literals from the set $\{u_1, u_2, \ldots, u_m\} \cup \{\bar{u}_1, \bar{u}_2, \ldots, \bar{u}_m\}$
PROPERTY: The set $\{D_1, D_2, \ldots, D_r\}$ is satisfiable.

12. CHROMATIC NUMBER
INPUT: graph G, positive integer k
PROPERTY: There is a function $\phi: N \to Z_k$ such that, if u and v are adjacent, then $\phi(u) \neq \phi(v)$.

13. CLIQUE COVER
INPUT: graph G', positive integer ℓ
PROPERTY: N' is the union of ℓ or fewer cliques.

14. EXACT COVER
INPUT: family $\{S_j\}$ of subsets of a set $\{u_i, i = 1,2,\ldots,t\}$
PROPERTY: There is a subfamily $\{T_h\} \subseteq \{S_j\}$ such that the sets T_h are disjoint and $\cup T_h = \cup S_j = \{u_i, i = 1,2,\ldots,t\}$.

15. HITTING SET
INPUT: family $\{U_i\}$ of subsets of $\{s_j, j = 1,2,\ldots,r\}$
PROPERTY: There is a set W such that, for each i, $|W \cap U_i| = 1$.

16. STEINER TREE
INPUT: graph G, $R \subseteq N$, weighting function $w: A \to Z$, positive integer k
PROPERTY: G has a subtree of weight $\leq k$ containing the set of nodes in R.

17. 3-DIMENSIONAL MATCHING
INPUT: set $U \subseteq T \times T \times T$, where T is a finite set
PROPERTY: There is a set $W \subseteq U$ such that $|W| = |T|$ and no two elements of W agree in any coordinate.

18. KNAPSACK
INPUT: $(a_1, a_2, \ldots, a_r, b) \in Z^{n+1}$
PROPERTY: $\sum a_j x_j = b$ has a 0-1 solution.

19. JOB SEQUENCING
INPUT: "execution time vector" $(T_1, \ldots, T_p) \in Z^p$,
"deadline vector" $(D_1, \ldots, D_p) \in Z^p$
"penalty vector" $(P_1, \ldots, P_p) \in Z^p$
positive integer k
PROPERTY: There is a permutation π of $\{1,2,\ldots,p\}$ such that

$$\left(\sum_{j=1}^{p} [\text{if } T_{\pi(1)} + \cdots + T_{\pi(j)} > D_{\pi(j)} \text{ then } P_{\pi(j)} \text{ else } 0]\right) \leq k \quad .$$

20. PARTITION
INPUT: $(c_1, c_2, \ldots, c_s) \in Z^s$
PROPERTY: There is a set $I \subseteq \{1,2,\ldots,s\}$ such that $\sum_{h \in I} c_h = \sum_{h \notin I} c_h$.

21. MAX CUT
INPUT: graph G, weighting function $w: A \to Z$, positive integer W
PROPERTY: There is a set $S \subseteq N$ such that

$$\sum_{\substack{\{u,v\} \in A \\ u \in S \\ v \notin S}} w(\{u,v\}) \geq W \quad .$$

It is clear that these problems (or, more precisely, their encodings into Σ^*), are all in *NP*. We proceed to give a series of explicit reductions, showing that SATISFIABILITY is reducible to each of the problems listed. Figure 1 shows the structure of the set of reductions. Each line in the figure indicates a reduction of the upper problem to the lower one.

To exhibit a reduction of a set $T \subseteq D$ to a set $T' \subseteq D'$, we specify a function $F: D \to D'$ which satisfies the conditions of Lemma 2. In each case, the reader should have little difficulty in verifying that F does satisfy these conditions.

SATISFIABILITY $\propto$ 0-1 INTEGER PROGRAMMING

$$c_{ij} = \begin{cases} 1 & \text{if } x_j \in C_i \\ -1 & \text{if } \bar{x}_j \in C_i \\ 0 & \text{otherwise} \end{cases} \qquad \begin{array}{l} i = 1,2,\ldots,p \\ j = 1,2,\ldots,n \end{array}$$

$b_i = 1 - (\text{the number of complemented variables in } C_i)$,
$i = 1,2,\ldots,p$.

SATISFIABILITY $\propto$ CLIQUE

$N = \{\langle\sigma,i\rangle \mid \sigma \text{ is a literal and occurs in } C_i\}$

$A = \{\{\langle\sigma,i\rangle,\langle\delta,j\rangle\} \mid i \neq j \text{ and } \sigma \neq \bar{\delta}\}$

$k = p$, the number of clauses.

CLIQUE $\propto$ SET PACKING

Assume $N = \{1,2,\ldots,n\}$. The elements of the sets $S_1, S_2, \ldots, S_n$ are those two-element sets of nodes $\{i,j\}$ not in A.

$S_i = \{\{i,j\} \mid \{i,j\} \notin A\}$, $i = 1,2,\ldots,n$

$\ell = k$.

How do we show a problem is **NP-complete**?

Step 1: show it is in **NP**

Example (SSP is in NP – Proof using a verifier)

On input $\langle\langle S, t\rangle, c\rangle$ where c is a subset of S:

- Test whether c is a collection of numbers that sum to t
- Test whether S contains all the numbers in c
- If both pass, accept; otherwise, reject

Example (SSP is in NP – Proof using nondeterminism)

On input $\langle S, t\rangle$:

- Non-deterministically select a subset c of the numbers in S
- Test whether c is a collection of numbers that sum to t
- If test passes, accept; otherwise, reject

How do we show a problem is **NP-complete**?

Step 2: show how problems in **NP** reduce to it

Sufficient to show $SAT \leq_P SSP$.

Example (SSP is NP-complete)

ϕ: Boolean formula with:

- variables $x_1, \ldots, x_k$
- and clauses $c_1, \ldots, c_k$.

Convert ϕ to an SSP instance $\langle S, t \rangle$ where: the elements of S and the number t are the rows in the following table are expressed in ordinary decimal notation

- S contains a pair (y_i, z_i) for each x_i
- Decimal representation is two parts (two complete rows)

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

How do we show a problem is **NP-complete**?

1. Assess the size of the input instance in terms of natural parameters.
2. Define a certificate and the checking procedure for it.
3. Analyze the running time of the checking procedure, using the same natural parameters.
4. Verify that this time is polynomial in the input size.

How do we show a problem A is **NP-complete**?

1. Prove that A is in **NP**.
2. Reduce a known **NP-complete** problem to A:
   1. Define the reduction: how a typical instance of the known **NP-complete** problem is mapped to an instance of A.
   2. Prove that the reduction maps 'yes' (resp. 'no') instances of the **NP-complete** problem to a 'yes' (resp. 'no') instance of A.
   3. Verify that the reduction can be carried out in polynomial time.

For **NP-hardness** we do not need step 1.

Optimization problems

A decision problem has a *true* or *false* answer, whereas an optimization problem involves maximizing or minimizing a function of several parameters.

Optimization Problems

Maximize or minimize a function of the input variables.

- **NP** and **NP-complete** only apply to **decision problems**.
- Optimization version of a **NP-complete** problem is at least as hard.
- It is **NP-hard** (**NP-hard** problems do not need to be decision problems).

Useful strategies for tackling **NP-hard** problems

1. Find tractable special cases which can be solved quickly.
2. Try **(meta-)heuristics** (fast, but not always correct).
3. Try exponential time algorithms better than exhaustive search.