7 Introduction to NP-Completeness

7.1 Introduction

[GJ, 1]

So far we have discussed efficient algorithms for selected problems. In this section we will discuss problems for which no efficient algorithm is known.
From now on we will consider

- Polynomial time algorithm is “practical”.
- Exponential and worse is “impractical”.

Consider the following problem:

Travelling salesman problem (TSP) Given is an undirected, weighted graph $G = (V, E)$. Find a cycle containing every vertex exactly once (tour) with the smallest possible length.

What if we cannot find a good (polynomial-time) algorithm to solve the problem?

- It would be great to prove that there exists no efficient algorithm. This is rarely feasible for “real world” problems.
- Instead we often prove that the problem is “NP-hard”
  - This does not say that it is impossible to solve the problem in polynomial time
  - But many smart people tried and failed

In this part of the course we will:

- Introduce theory of NP-completeness
- Introduce the famous P = NP open problem
- Learn how to prove that a problem is NP-complete

Why is this important?

- If you know that the problem is NP-hard, you know that it is quite unlikely that you find an efficient algorithm to solve it.
- You are justified to use other methods to cope with the problem:
  - heuristics
  - approximation algorithms
  - integer programming
  - backtracking (exhaustive search)

7.2 Optimization versus decision problems

NP-completeness theory is built for so called “decision” problems.

Optimization problem: find a an object optimizing some function.
TSP – find a tour with the smallest possible length

Decision problem: problem that has yes/no answer.
TSP-D – given is a value B. Is there a tour with length at most B?

Observation: If the cost function is easy to evaluate, then decision problem is no harder than corresponding optimization problem.

For example: If we can find the minimum length of a TSP tour, then we can compare it to value of B and thus solve the decision problem.

Note: Often the reverse is also true (i.e. if we can solve the decision problem in polynomial time then we can solve optimization problem in polynomial time as well.

7.3 Class P (polynomial)

[BB 12.5.1 or CLRS2 34.1]

Recall what does "running time" mean?

Running time of an algorithm A is a function of the size of the input instances, where $T_A(n)$ is the largest time required to solve instance an of $\boxed{\text{size}}$ n.

Size of an instance is the number of bits needed to encode the instance.

Definition 1. *Decision problem Q belongs to the* **class of problems** P *iff there exists a polynomial-time algorithm solving problem Q.*

Example 1: Recall Bentley's problem. Is it in class P? No! It is not a decision problem!

Example 2: Reformulate Bentley's problem as a decision problem: Given an array $A[1..n]$ of integers and an integer B. Is there a subarray with sum at lest B? Is it in class P? Yes!

Example 3: Is the decision version of coin changing problem in class P?

- Decision version of the coin changing problem: Given are n coin denominations, sum S and number B. Is it possible to pay out sum S with at most B coins?
- How many bits are needed to encode input? $O(\log n + \sum_{i=1}^{n} \log c_i + \log S + \log B)$
- What is the running time of the best algorithm we know? $O(nS)$
- Is it polynomial in size of the input? No! Because $S = 2^{\log S}$. Therefore the running time is exponential!

So we are unable to prove that the problem is in P!

7.4 Non-deterministic algorithms

[BB 12.5.6]

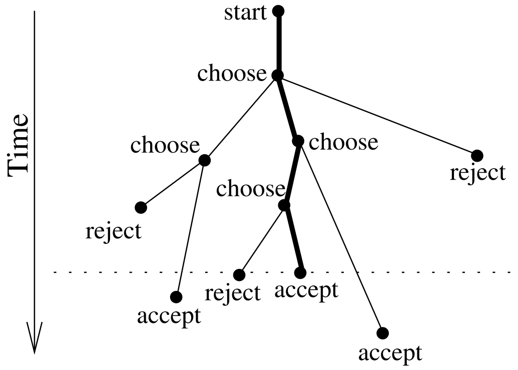
(only defined for decision problems)

We will add the following constructs to our pseudocodes:

- `accept` – finish computation and answer "yes"
- `reject` – finish computation and answer "no"

- `choose k between i and j` – set k to a value between i and j so that the program flow gets to the "accept" instruction in the shortest possible way (or arbitrarily, if the answer is no)

  Note that we cannot implement "choose" on any regular computer.



Example: Non-deterministic algorithm for TSP-D (assume $V = \{1, 2, \ldots, n\}$, adjacency matrix representation)

```
function TSP-D
  visited[i]:=false for all vertices;
  last_visited:=1; visited[1]:=true;
  length:=0;
  repeat n-1 times
    choose next_visited between 1 and n;
    if visited[next_visited] then reject;
    //we cannot visit a single vertex twice
    visited[next_visited]:=true;
    length:=length+w(last_visited,next_visited);
    last_visited:=next_visited;
  length:=length+w(last_visited,1);
  if length<=B then accept;
               else reject;
```

Running time of non-deterministic algorithm

Accepting computation is a computation that ends by "accept" instruction

Running time of non-deterministic algorithm A on instance x is the running time of the shortest possible accepting computation for x (undefined if x is rejected).

Running time of a non-deterministic algorithm A is a function $T_A(n)$. $T_A(n)$ is the largest running time over all "yes" instances of size n.

Note: Non-determinism is one of the most important yet least practical concepts in computer science. It was first introduced in formal languages (see cs360/cs365).

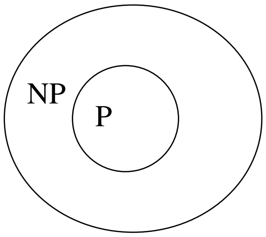
7.5 Class NP (non-deterministic polynomial)

[GJ 2.3, BB 12.5.1-12.5.2]

Definition 2. *A decision problem Q belongs to the class of problems NP iff there exists a polynomial-time non-deterministic algorithm solving problem Q.*

Note:

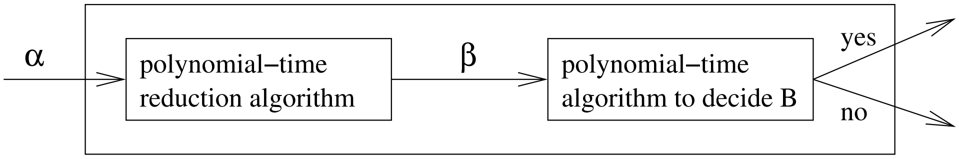
- All problems in P are in NP (we just do not use "choose"). Therefore P $\subseteq$ NP.
- Are there problems which are in NP but not in P?
  - This is famous P $\overset{?}{=}$ NP problem
  - General "hunch" is $P \neq$ NP (but nobody was able to prove this so far)



7.6 Reductions

[CLRS2 34 intro, BB 12.5.2]

Before we have seen that sometimes one problem can be solved by using another problem. Consider two decision problems A and B.


α
polynomial–time reduction algorithm
β
polynomial–time algorithm to decide B
yes
no

polynomial–time algorithm to decide A

- If problem B can be solved in polynomial time then problem A can be solved in polynomial time
- If there is no polynomial-time algorithm to solve A then there is no polynomial-time algorithm to solve B

We have reduced problem A to problem B.

Definition 3. *We say that a decision problem A is* polynomially many-one reducible *to problem B ($A \leq_p B$) if there exists a function f computable in polynomial time that:*

- *maps every instance x of A to an instance $f(x)$ of B and*
- *x is a yes-instance of A if and only if $f(x)$ is a yes-instance of B.*

Example 1: Consider Hamiltonian circuit problem:

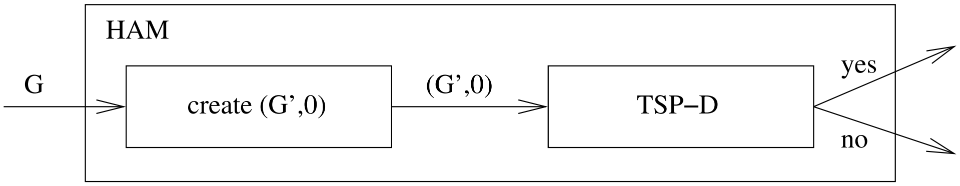
HAM: Given an undirected graph G. Is there a tour in G passing through every vertex exactly once?

Task: Show that HAM$\leq_p$TSP-D (i.e., TSP-D can be used to solve HAM).

Solution: Take a graph G which is an input of HAM. We want to create an input for TSP-D problem which is a pair (undirected graph G',threshold). Create a complete graph G' with the following weighting function:

- $w(u, v) = 0$, if (u, v) is an edge in G,
- $w(u, v) = 1$ otherwise.

Note, that the graph G has a tour if and only if graph G' has a tour of total length at most 0 and thus we can use TSP-D with input $(G', 0)$ to solve problem HAM.



Example 2: [GJ, 3.1.3] Consider the following two problems:

3-SAT: 3-Satisfiability. Consider a set of boolean variables $U = (u_1, u_2, \dots, u_m)$ and a logical formula of the form:

$$(a_{1,1} \vee a_{1,2} \vee a_{1,3}) \wedge (a_{2,1} \vee a_{2,2} \vee a_{2,3}) \wedge \dots \wedge (a_{n,1} \vee a_{n,2} \vee a_{n,3}),$$

where $a_{i,j}$ is either a variable u_i from U or its negation $\neg u_i$. ($a_{i,j}$ is called *literal*)

Is there an assignment of variables with values *true* and *false* so that the formula is satisfied?

Example:

- Formula
$$(u_1 \vee \neg u_3 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_4)$$
is satisfiable (for example, an assignment $u_1 = \textit{true}, u_2 = \textit{true}, u_3 = \textit{true}, u_4 = \textit{true}$ satisfies the formula).
- Formula
$$(\neg u_1 \vee \neg u_1 \vee \neg u_1) \wedge (u_1 \vee \neg u_2 \vee \neg u_2) \wedge (u_1 \vee u_2 \vee \neg u_3) \wedge (u_1 \vee u_2 \vee u_3)$$
cannot be satisfied.

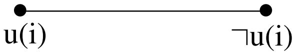
VC: Vertex Cover. Given is a pair (G, K), where $G = (V, E)$ is an undirected graph and K is a number. Does there exist a subset of at most K vertices V' such that for each edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$ (i.e., every edge is "covered" by the set V').

Task: Show that 3-SAT$\leq_p$VC.

In other words: We are given a formula and we want to construct a graph G so that graph G has a vertex cover of size K if and only if the formula is satisfiable.

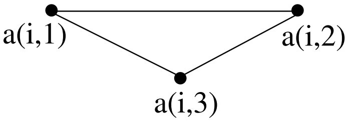
Construction of the graph: Construct graph G as follows:

- **Type (1)** For every variable u_i: (2 vertices, 1 edge)

  To cover the edge, at least one of the vertices u_i and $\neg u_i$ must be in the cover.

- **Type (2)** For every clause $(a_{i,1} \vee a_{i,2} \vee a_{i,3})$

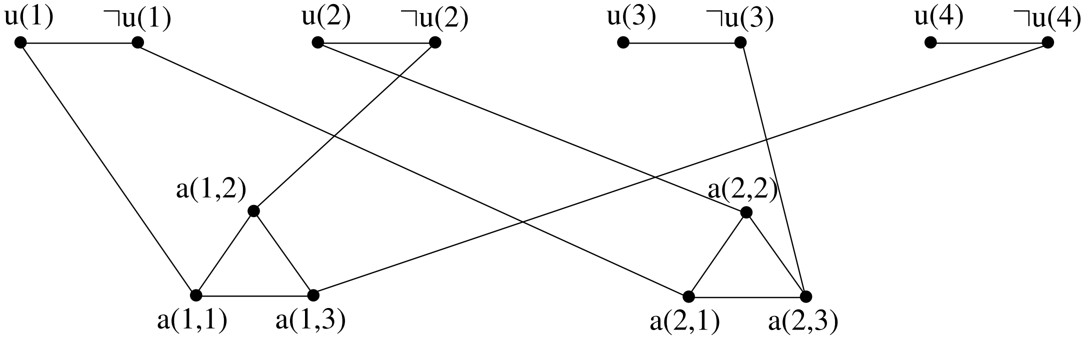
  

  To cover edges in the triangle, at least two of the vertices $a_{i,1}$, $a_{i,2}$, $a_{i,3}$ must be in the cover.

- **Type (3)** If literal $a_{i,j} = u_k$ then connect vertices $(a_{i,j}, u_k)$.
  If literal $a_{i,j} = \neg u_k$ then connect vertices $(a_{i,j}, \neg u_k)$.

Every vertex cover of this graph must have at least $m + 2n$ vertices.

Example: $(u_1 \vee \neg u_2 \vee \neg u_4) \wedge (\neg u_1 \vee u_2 \vee \neg u_3)$



Idea: vertex of type (1) is in the vertex cover iff the corresponding literal is true.

Lemma 1. *A 3-SAT formula is satisfiable iff the graph constructed as described above has a vertex cover of size at most $m + 2n$.*

Proof.

- (⇒) (If formula is satisfiable, then graph has a vertex cover of size $m + 2n$)

  Take assignment of variables that satisfies the formula:

  $$\left.\begin{array}{l}\bullet \text{ If } u_i = \textit{true}\text{, put } u_i \text{ to VC}\\ \bullet \text{ If } u_i = \textit{false}\text{, put } \neg u_i \text{ to VC}\end{array}\right\}\text{covers edges of type (1)}$$

  $$\left.\begin{array}{l}\bullet \text{ For every clause:}\\ \qquad\bullet \text{ select one satisfied literal}\\ \qquad\bullet \text{ add vertices corresponding to the other two VC}\end{array}\right\}\text{covers edges of type (2)}$$

  - This vertex cover is of size $m + 2n$
  - The only edges which are possibly not covered are of type (3). An edge of type (3) can represent:
    - satisfied literal – then it will be covered by a vertex of type (1)
    - unsatisfied literal – it will be covered by a vertex of type (2)

- (⇐) (If graph has a vertex cover of size $m+2n$, then formula is satisfiable)

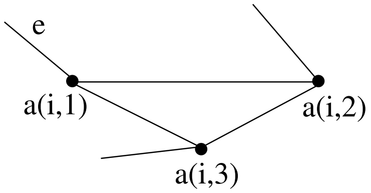
  Take the vertex cover of the graph. As we have shown before:

  - From each pair of vertices of type (1) at least one must be selected
  - From each triple of vertices of type (2) at least two must be selected

  This requires $m+2n$ vertices in the vertex cover. Therefore

  - From each pair of vertices of type (1) **exactly** one must be selected
  - From each triple of vertices of type (2) **exactly** two must be selected

  Let us define assignment of variables such that $u_i = \textit{true}$ iff vertex cover contains vertex u_i. We will prove that this assignment satisfies the formula.

  Assume to the contrary that there exists a clause that is not satisfied. Consider corresponding triple of type (2) vertices. One of them is not in the vertex cover. WLOG assume it is $a_{i,1}$. Then edge e must be covered by type (1) vertex on its other side. But that means that literal $a_{i,1}$ is satisfied, which is contradiction.

□

7.7 NP-hard and NP-complete problems

[GJ, 2.6]

- We have defined relation $A \leq_p B$, which intuitively means: the problem B is **harder** than (or at least as hard as) problem A
- What about problems that are **hardest in NP** – harder than any other problem in NP?

Definition 4. *Problem Q in* NP-hard *iff for any problem $R \in$ NP, $R \leq_p Q$. If a NP-hard problem Q is in NP, we say it is* NP-complete.

Recall

- If $A \leq_p B$ and we can solve B in poly-time, then we can solve A in poly-time as well
- If somebody would solve an NP-complete problem in polynomial time, then P = NP.
- This is why proving that a problem is NP-complete automatically establishes that it would be hard to solve in polynomial time.

SAT: SATISFIABILITY Consider a set of boolean variables $(u_1, \ldots, u_m)$ and a logical formula f.
Problem: Is there an assignment of the variables so that f is satisfied?

Theorem 1 (Cook's Theorem). *SAT is NP-complete.*

Sketch of the proof: We need to prove:

1. SAT∈NP –or–
   There exists a non-deterministic polynomial algorithm solving SAT.
2. SAT is NP-hard –or–
   For any problem Q in NP, $Q \leq_p$SAT

1. SAT∈NP

```
for i:=1 to m do
  choose u[i] between 0 and 1; // 0 means false,
                               // 1 means true

evaluate formula f with assignment
(u[1],u[2],...,u[m])

if f is satisfied then ACCEPT
                  else REJECT
```

2. SAT is NP-hard Consider a problem $Q \in$ NP
$\Longrightarrow$there is a poly-time non-deterministic algorithm solving Q

How do we express such algorithm?

- Each memory cell (register) stores number of fixed size
  (registers $R_1, R_2, \ldots$)
- Program is fixed and has constant number of lines
  (all lines are numbered)
- At the beginning, input is stored in the first n registers
  (n is the size of the input)
- Program runs for at most $p(n)$ steps
  and accesses at most first $q(n)$ registers
  ($p(n)$ and $q(n)$ are polynomials in n)
- Instruction set contains following instructions:
  - ACCEPT
  - REJECT
  - GOTO m
  - IF $R_\ell = 0$ THEN GOTO m
  - CHOOSE R_l BETWEEN 0 AND 1
  - basic arithmetic operations
    (e.g. $R_\ell := R_u + R_v$, $R_\ell := R_u * R_v$)
  - some mechanism for addressing any register up to $q(n)$
    (this is somewhat complicated)

You probably would not like to program in such simple language
BUT if needed, we can program efficiently in it.

To prove $Q \leq_p$SAT, we want:

- Given a program A solving Q in poly-time and instance $x = x_1, x_2, \ldots, x_n$.
- Construct a large logical formula f that "simulates" program A on input x;
- A can reach ACCEPT $\Longleftrightarrow$ f is satisfiable

Variables of the formula:

- $Q[i,k]$ – at time i, the program is executing line k
- $S[i,j,k]$ – at time i, register j has value k

Formula f will be a conjunction ("AND") of several groups of smaller formulas; all of these smaller formulas must be satisfied to satisfy f

1. "At each time i, the program is executing exactly one line."
$\neg(Q[i,k] \wedge Q[i,\ell])$ for all i, and $k \neq \ell$

2. "At each time i, each register contains a single value."
$\neg(S[i,j,k] \wedge S[i,j,l])$ for all i,j, and $k \neq \ell$

3. At time 0:

- Program is executing line 1: $Q[0,1]$
- First n registers hold values $x_1,\ldots,x_n$:
  $S[0,1,x_1] \wedge S[0,2,x_2] \wedge \ldots \wedge S[0,n,x_n]$
- Other registers hold 0:
  $S[0,n+1,0] \wedge S[0,n+2,0] \wedge \ldots \wedge S[0,q(n),0]$

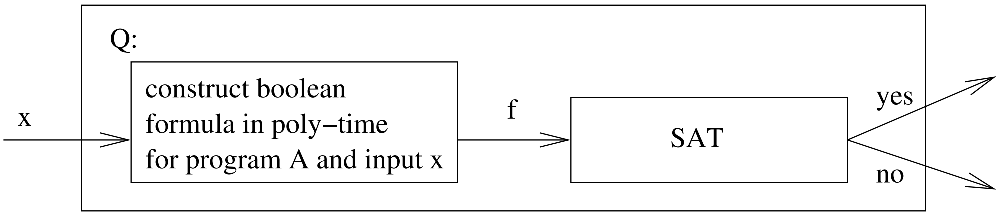
4. "After $p(n)$ time steps program has entered a line with ACCEPT command"
$Q[p(n),k_1] \vee Q[p(n),k_2] \vee \ldots$
where $k_1,k_2,\ldots$ are the lines containing "ACCEPT"

5. "For each time $0 \leq i < p(n)$, the state of the computer changes between time i and $i+1$ according to the program."

Contents of line k	**Formula**
ACCEPT or REJECT	$Q[i,k] \Rightarrow Q[i+1,k]$
GOTO ℓ	$Q[i,k] \Rightarrow Q[i+1,\ell]$
IF $R_\ell = 0$ THEN GOTO m	$Q[i,k] \wedge S[i,\ell,0] \Rightarrow Q[i+1,m]$ $Q[i,k] \wedge \neg S[i,\ell,0] \Rightarrow Q[i+1,k+1]$
CHOOSE R_ℓ	$Q[i,k] \Rightarrow Q[i+1,k+1] \wedge$ $(S[i+1,\ell,0] \vee S[i+1,\ell,1])$
...and so on for other instructions	

SAT is NP-hard: summary The above mentioned algorithm constructs for a given algorithm A and input x formula f:

- Algorithm runs in polynomial time in n.
- Resulting formula is of polynomial size in n.
- f is satisfiable $\Longleftrightarrow$ A accepts x



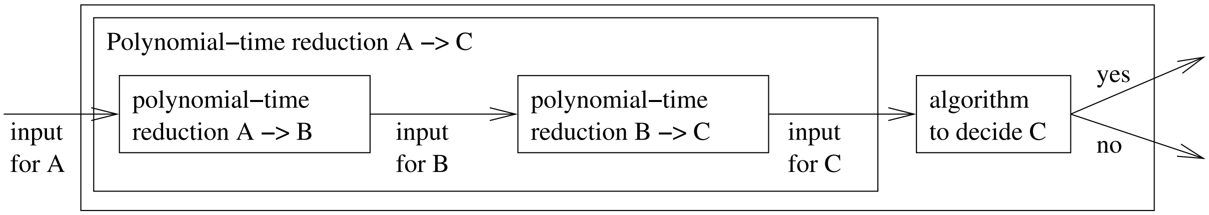
$\Longrightarrow \boxed{\text{we proved: } Q \leq_p \text{SAT for any } Q \in NP}$

7.8 How to prove other problems are NP-complete?

- We have proved that SAT is NP-complete by reducing any problem $Q \in$ NP to SAT ($Q \leq_p$SAT)
- Thus if SAT can be solved in polynomial time then **every problem in NP** can be solved in polynomial time.

Lemma 2. *If $A \leq_p B$ and $B \leq_p C$, then $A \leq_p C$ (i.e., $\leq_p$ is transitive).*

Proof.


Polynomial–time reduction A –> C
input for A
polynomial–time reduction A –> B
input for B
polynomial–time reduction B –> C
input for C
algorithm to decide C
yes
no

□

Corollary 1. *If N is NP-complete problem and $N \leq_p Q$, then Q is NP-hard.*

Proof. For any problem $U \in$ NP, $U \leq_p N$ and $N \leq_p Q$ and therefore $U \leq_p Q$. □

Recall: Problem Q is NP-complete iff

- $Q \in$ NP, and
- Q is NP-hard.

To prove that problem Q is NP-hard we can:

1. Choose a problem N which we know is NP-complete (such as SAT)
2. Show $N \leq_p Q$:
   - Give a poly-time algorithm transforming instance x of N to instance $f(x)$ of Q
   - Show: if x is a "yes" instance of N, then $f(x)$ is a "yes" instance of Q.
   - Show: if x is a "no" instance of N, then $f(x)$ is a "no" instance of Q.
     OR
     if $f(x)$ is a "yes" instance of Q, then x is a "yes" instance of N.
3. Conclude that since N is NP-complete, Q must by NP-hard.

To prove that $Q \in$ NP and thus finishing the proof that Q is NP-complete we can:

4a. Design a poly-time non-deterministic algorithm that solves Q

Alternative: poly-time verification [CLRS2 34.2]

- Consider a salesperson (not a traveling one) wants to sell you a big graph, claiming it is Hamiltonian.
- You say you cannot verify that claim because no efficient algorithm for that task is known.
- The salesperson provides you with the order of vertices on the Hamiltonian cycle.
- Now you can easily verify in polynomial time that this order of vertices indeed specifies a Hamiltonian cycle and you purchase the graph.

Definition 5. *A* verification algorithm *for problem Q is a two-argument algorithm $A(x, y)$ where*

- *x is an instance of a problem Q (let $|x| = n$)*
- *y is a string (called a* certificate*) size of which is bounded by polynomial $q(n)$*

with the following properties:

- *Running time of A is polynomial in n.*
- *If x is "no" instance of the problem Q, $A(x, y) =$ "no" for every certificate y.*
- *If x is "yes" instance of the problem Q, there exists a certificate y such that $A(x, y) =$ "yes".*

Example: Certificate for the HAM problem is a list of vertices in the order on the Hamiltonian cycle. Clearly, it has polynomial size. The verification algorithm checks that the list contains each vertex of the graph exactly once and that every two adjacent vertices are connected by an edge. If the graph is hamiltonian, the correct list of vertices will pass the verification algorithm. For a non-hamiltonian graph no list of vertices will pass verification.

Lemma 3. *Problem Q is in NP iff there exists a verification algorithm for Q.*

Proof.
(⇒) Certificate for input x is a list of all `choose` command choices in the shortest accepting computation on instance x.
(⇐) Non-deterministically generate certificate and run verification algorithm. □

Alternative way to prove $Q \in NP$

4b. Define a polynomial-size certificate

5b. Give polynomial-time verification algorithm

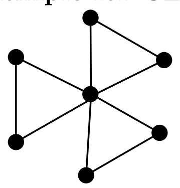
You are free to choose either of those methods (4a or 4b and 5b)

7.9 Seven basic NP-complete problems

Before proceeding with further NP-completeness proofs, we need "arsenal"of basic NP-completeness problems.

SAT	Instance:	Boolean formula f
	Problem:	Can f be satisfied?
3-SAT	Instance:	Boolean formula f in the form $(a_{1,1} \vee a_{1,2} \vee a_{1,3}) \wedge \ldots \wedge (a_{n,1} \vee a_{n,2} \vee a_{n,3})$
	Problem:	Can f be satisfied?
VC	Instance:	Graph $G = (V, E)$; number K
	Problem:	Is there a set of vertices V' of size $\leq K$ s.t. for any $e = (u, v) \in E$, $u \in V'$ or $v \in V'$?
HAM	Instance:	Graph $G = (V, E)$
	Problem:	Is there a Hamiltonian cycle in G?
TSP-D	Instance:	Weighted graph $G = (V, E)$; number K
	Problem:	Is there a tour with length $\leq K$?
CLIQUE	Instance:	Graph $G = (V, E)$; number K
	Problem:	Does G contain a complete subgraph with $\geq K$ vertices?
SUBSET-SUM	Instance:	n numbers $s_1, s_2, \ldots, s_n$; target t
	Problem:	Is there a subset of the numbers $s_1, \ldots, s_n$ with sum exactly t?

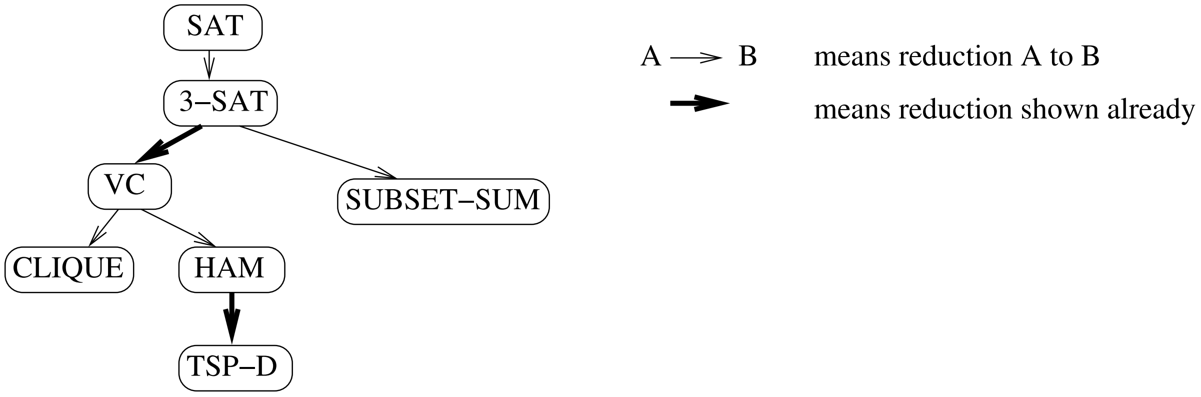
Example for CLIQUE: The following graph contains a clique of size 3 but not a clique of size 4.



Example for SUBSET-SUM: Consider numbers 2,3,3,5

- If $t = 7$, "yes" instance (2+5)
- If $t = 13$, "yes" instance (2+3+3+5)
- If $t = 12$, "no" instance

Sequence of reductions to prove NP-completeness:



7.10 More NP-completeness proofs

Subset-sum is NP-complete

1. By reduction from 3-SAT
2. **Want:** Given a formula $f = (a_{1,1} \vee a_{1,2} \vee a_{1,3}) \wedge \ldots \wedge (a_{n,1} \vee a_{n,2} \vee a_{n,3})$, construct an instance of SUBSET-SUM (set of numbers and target)

   Idea: We will have 2 numbers for every variable – one for the case when variable is true and one for the case when variable is false.

   Target must enforce choosing exactly one of those two.

   Details:

   – For every variable u_i create two $m + n$ digit numbers v_i and v_i' of base 10 as follows:

	u_1	u_2	$\ldots$	u_i	$\ldots$	u_m	C_1	C_2	$\ldots$	C_n
v_i (corresponds to u_i):	0	0	$\ldots$	1	$\ldots$	0	$C_k = 1$ iff u_i is in clause C_k			
v_i' (corresponds to $\neg u_i$):	0	0	$\ldots$	1	$\ldots$	0	$C_k = 1$ iff $\neg u_i$ is in clause C_k			

   – If we choose exactly the numbers corresponding to some satisfying truth assignment, we get the following sum:

u_1	u_2	$\ldots$	u_m	C_1	C_2	$\ldots$	C_n
1	1	$\ldots$	1	≥ 1	≥ 1	$\ldots$	≥ 1
				≤ 3	≤ 3	$\ldots$	≤ 3

- Choose target:

u_1	u_2	$\dots$	u_m	C_1	C_2	$\dots$	C_n
1	1	$\dots$	1	4	4	$\dots$	4

- Include for every clause the following numbers:

	u_1	u_2	$\dots$	u_m	C_1	C_2	$\dots$	C_i	$\dots$	C_n
s_i	0	0	$\dots$	0	0	0	$\dots$	1	$\dots$	0
s_i'	0	0	$\dots$	0	0	0	$\dots$	2	$\dots$	0

  These numbers we can use to "complete" the sum in column c_i to 4 if we already have sum 1, 2, or 3.

Example: Formula $(u_1 \vee \neg u_2 \vee \neg u_3) \wedge (\neg u_1 \vee \neg u_2 \vee \neg u_3)$

		u_1	u_2	u_3	C_1	C_2
→	v_1	1	0	0	1	0
	v_1'	1	0	0	0	1
→	v_2	0	1	0	0	0
	v_2'	0	1	0	1	1
	v_3	0	0	1	0	0
→	v_3'	0	0	1	1	1
	s_1	0	0	0	1	0
→	s_1'	0	0	0	2	0
→	s_2	0	0	0	0	1
→	s_2'	0	0	0	0	2
	target	1	1	1	4	4

Show: If f is satisfiable then there exists a subset with sum equal to target

Proof. Take satisfying assignment and choose the subset of numbers as follows:

- v_i if $v_i = \mathit{true}$; otherwise v_i'
- if clause C_i has
  - ∗ 1 satisfied literal: s_i, s_i'
  - ∗ 2 satisfied literals: s_i'
  - ∗ 3 satisfied literals: s_i

The subset selected in this way has sum equal to target:

- For every variable we have either v_i or v_i', therefore column u_i sums to 1
- Since every clause is satisfied, column C_i sums to 4

□

Show: If there is a subset with sum equal target, then f is satisfiable.

Proof.

- No carry-overs between columns can occur (because every column sums to less than 10 even if we use all numbers)
- From every pair v_i, v_i' exactly one must be in the subset (because column v_i sums to 1).
  Create a truth assignment as follows:
  - ∗ $u_i = \mathit{true}$ if v_i was chosen
  - ∗ $u_i = \mathit{false}$ if v_i' was chosen

- To achieve sum 4 in column C_i there must be some v_i or v_i' contributing to sum in this column. This gives us a literal satisfying clause C_i.

  Since each clause is satisfied, f is satisfiable.

□

3. Since 3-SAT is NP-complete, SUBSET-SUM is NP-hard

4b. Certificate is the subset achieving the target sum. Length of certificate is clearly polynomial in the size of the input.

5b. Verification algorithm

  - verifies that the certificate specifies a subset of the input list of numbers
  - verifies that the sum of the subset is equal to the target

Steps 4b and 5b imply that SUBSET-SUM is in NP and step 3 implies it is NP-hard. Therefore it is NP-complete.

Coin-changing is NP-complete

COIN-CHANGING (decision version) Given m coin denominations, sum S and number B. Is it possible to pay out sum S with at most B coins?

1. Prove by reduction from SUBSET-SUM (SUBSET-SUM$\leq_p$COIN-CHANGING)

2. **Given:** Instance of SUBSET-SUM $s_1, s_2, \ldots, s_n$, target t.

   Want: set of coins, target sum S, number of coins B

   Recall: In coin changing problem, each coin can be used more than once!

   Coin values: Let $M = \max\{s_1, s_2, \ldots, s_n\}$

   For every number s_i create two coins c_i and c_i':

	value	s_1	$\ldots$	s_i	$\ldots$	s_m
	(base $2nM$)	(base $2n$)		(base $2n$)		(base $2n$)
c_i	s_i	0	$\ldots$	1	$\ldots$	0
c_i'	0	0	$\ldots$	1	$\ldots$	0

   Size of numbers c_i, c_i':

   - column "value" has $\log n + \log M + 1$ bits,
   - each other column has $\log n + 1$ bits.

   Therefore the size of the number is polynomial in the size of the SUBSET-SUM instance.

   Intuition:

   - Choosing coin c_i means including s_i in the subset
   - Choosing coin c_i' means NOT including s_i in the subset

   Finish instance of COIN-CHANGING:

   - number of coins B set to be n
   - target sum S:

value	s_1	$\ldots$	s_i	$\ldots$	s_m
(base $2nM$)	(base $2n$)		(base $2n$)		(base $2n$)
t	1	$\ldots$	1	$\ldots$	1

Show: If there exists a subset with sum t then it is possible to pay out the sum S with at most n coins.

Proof. Given the subset, choose coins as follows:

- If s_i is chosen, take coin c_i
- If s_i is not chosen, take coin c'_i

Observe

- we use n coins
- sum in the first column is t
- sum in every other column is 1

□

Show: If it is possible to pay out sum S with at most n coins, there exists a subset with sum t.

Proof.

- Since we use at most n coins to pay S, there are no "overflows" between columns.
- Thus for every pair c_i, c'_i exactly one is chosen and is used exactly once (because the sum must have 1 in column s_i)
- Construct a subset of $s_1, s_2, \ldots, s_n$ as follows: s_i is in the subset iff coin c_i was chosen.
- The first column assures that the sum of the elements in the subset is t.

□

3. Since SUBSET-SUM is NP-complete, and SUBSET-SUM$\leq_p$COIN-CHANGING, COIN-CHANGING is NP-hard.

4b. **Certificate:** the number of coins of each denomination used to pay out the sum.

Each of these numbers is at most B and there are m of them, therefore the size of the certificate is polynomial.

5b. **Verification:**

- check that the number of coins used it at most B
- check that the sum of coins used is S

We have proved that COIN-CHANGING is in NP and therefore it is NP-complete.