NP-completeness

Chapter 34

Sergey Bereg
Examples

Some problems admit polynomial time algorithms, i.e. $O(n^k)$ running time where $n$ is the input size. We will study a class of NP-complete problems which will be defined later. No polynomial time algorithm is known for any NP-complete problem.

**Shortest vs. longest simple paths.** Given a directed graph $G = (V,E)$ with non-negative edge weights, the shortest path from between two vertices can be found in $O(VE)$ time. Finding the longest simple path is a NP-complete problem.

What if we modify Bellman-Ford algorithm by changing “$>$” to “$<$” in the relaxation condition?

```
RELAX(u, v, w)
// relax edge (u, v) using weight w(u, v)
1   if d[v] > d[u] + w(u, v) then
2       d[v] = d[u] + w(u, v)
3       π[v] = u
```
Examples

Some problems admit \textit{polynomial time algorithms}, i.e. $O(n^k)$ running time where $n$ is the input size. We will study a class of \textit{NP-complete problems} which will be defined later. No polynomial time algorithm is known for \textit{any} NP-complete problem.

\textbf{Shortest vs. longest simple paths}. Given a directed graph $G = (V, E)$ with non-negative edge weights, the \textit{shortest} path from between two vertices can be found in $O(VE)$ time. Finding the \textit{longest} simple path is a NP-complete problem.

There is a graph $G$ and a vertex $s \in G$ such that a modified Bellman-Ford algorithm does not find the longest simple paths.
Definitions

An abstract problem $Q$ is a binary relation on a set $I$ of problem instances and a set $S$ of problem solutions.

Example. An instance for Shortest-Path is a graph $G = (V, E)$ and two vertices. A solution is an ordered sequence of vertices (can be empty sequence if no path exists). A given instance may have more than one solution.

A decision problem is a problem having yes/no solution. So $S = \{0, 1\}$. Many problems are optimization problems where some value must be minimized or maximized. Usually it is easy to cast an optimization problem as a decision problem.

Example. Decision problem for Shortest-Path is the problem Path: Given a graph $G = (V, E)$, vertices $u$ and $v$, and an integer $k$, is there a path of from $u$ to $v$ consisting at most $k$ edges?
Encodings

*Encoding* of a set $S$ is a mapping $e : S \rightarrow \{0, 1\}^*$ from $S$ to the set of binary strings. *Example:* $S = \mathbb{N}$, then binary code $e : \mathbb{N} \rightarrow \{0, 1\}^*$.

A problem whose instance set is $\{0, 1\}^*$ is called a *concrete problem*. We say that an algorithm *solves* a concrete problem in time $O(T(n))$ if it takes $O(T(n))$ time for a problem instance $i$ of length $n = |i|$. If $T(n) = O(n^k)$ then we say that the problem is *polynomially solvable*.

*Complexity class* $\mathbb{P}$ is a class of concrete decision problems that are polynomially solvable.
Formal language framework

- An **alphabet** $\Sigma$ is a finite set of symbols.
- A **language** $L$ over $\Sigma$ is any set of strings made up of symbols from $\Sigma$.
- **Example**: if $\Sigma = \{0, 1\}$, the set $L = \{10, 11, 101, 111, \ldots\}$ is the language of binary representations of prime numbers.
- We denote the **empty string** by $\varepsilon$ and the **empty language** by $\emptyset$.
- So, $L \subseteq \Sigma^*$ where $\Sigma^*$ is the language of all strings on $\Sigma$.

Operations on languages:
- **union** $L_1 \cup L_2$,
- **intersection** $L_1 \cap L_2$,
- **complement** of $L$ is $\Sigma^* - L$,
- **concatenation** of $L_1$ and $L_2$ is $\{x_1x_2 : x_1 \in L_1, x_2 \in L_2\}$,
- **closure or Kleene star** $L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$ where $L^k$ is the language obtained by concatenating $L$ to itself $k$ times.
Formal language framework

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

Question: Is it true that $A$ rejects a string $x \notin L$?
• An algorithm $A$ *accepts* a string $x \in \{0,1\}^*$ if, given input $x$, the algorithm’s output is 1, i.e. $A(x) = 1$.

• $A$ *rejects* $x$ if $A(x) = 0$.

• The language *accepted* by an algorithm $A$ is the set of strings $L = \{x \in \{0,1\}^* : A(x) = 1\}$.

Question: Is it true that $A$ rejects a string $x \notin L$?

No, $A$ may not reject a string $x \notin L$ (*$A$ may loop forever on $x$*).
A language $L$ is *decided* by an algorithm $A$ if $A$ accepts/rejects every string $x$ in/not in $L$.

A language $L$ is *accepted in polynomial time* by an algorithm $A$ if, for any $x \in L$, $|x| = n$, it accepts $x \in L$ in $O(n^k)$ time for some fixed $k$.

A language $L$ is *decided in polynomial time* by an algorithm $A$ if, for any $x \in \{0, 1\}^*$, $|x| = n$, it decides whether $x \in L$ in $O(n^k)$ time for some fixed $k$.

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

**Theorem.** $P = \{L \subseteq \{0, 1\}^* : \exists A \text{ that accepts } L \text{ in polynomial time}\}$. 
Theorem. \( P = \{ L \subseteq \{0, 1\}^* : \exists A \text{ that accepts } L \text{ in polynomial time} \} \).

Proof. If a language \( L \) is decided by a polynomial-time algorithm \( A \), then \( L \) is accepted by \( A \).

Suppose that a language \( L \) is accepted by a polynomial-time algorithm \( A \) with running time at most \( T(n) = cn^k \).

We use a “simulation” argument and create a new algorithm \( A' \).
For an input string \( x \), the algorithm \( A' \):

1. Computes the length \( n = |x| \).
2. Simulates the action of \( A \) only for time \( T(n) \).
   If \( A \) accept/rejects \( x \) then \( A' \) reports it and stops.
3. Rejects \( x \).
Verification algorithms

**Hamiltonian cycle problem** HAM-CYCLE. Given an undirected graph $G$, is there a simple cycle that contains all the vertices of $G$?

An algorithm: encode graph with adjacency matrix, the number of vertices $m \geq \sqrt{n}$. There are $m!$ permutations and the running time to check them is at least $\Omega(m!) = \Omega((\sqrt{n})!) = \Omega(2^{\sqrt{n}})$. Not polynomial time.

To verify a solution of HAM-CYCLE we need a *certificate*, for example, the hamiltonian cycle. A *verification algorithm* has two arguments - input string and certificate string. A two-argument algorithm $A$ verifies an input string $x$ if there is a certificate $y$ such that $A(x, y) = 1$.

The *language verified* by a verification algorithm $A$ is

$$L = \{x \in \{0, 1\}^* : \exists y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}.$$
Is there a walk along a closed path on dodecahedron such that every vertex is visited one time?
**Complexity class NP**

*Complexity class NP* is the class of languages that can be verified by a polynomial time algorithm. So, \( L \in \text{NP} \) if and only if there exist a two-input polynomial-time algorithm \( A \) and a constant \( c \) such that

\[
L = \{ x \in \{0,1\}^* : \exists y \in \{0,1\}^* \text{ such that } |y| = O(|x|^c) \text{ and } A(x,y) = 1 \}.
\]

We say that algorithm \( A \) verifies \( L \) in *polynomial time*.

*Example:* \( \text{HAM-CYCLE} \in \text{NP} \).

If \( L \in \text{P} \) then \( L \in \text{NP} \) since there is a polynomial-time algorithm to decide \( L \) (the verification algorithm ignores the second argument). Thus, \( \text{P} \subseteq \text{NP} \).

Whether \( \text{P} = \text{NP} \) is the famous problem.

It is not even known if \( \text{NP} \) is closed under complement.

That is, does \( L \in \text{NP} \) imply \( \overline{L} \in \text{NP} \)?

*Complexity class co-NP* is the set of languages \( L \) such that \( \overline{L} \in \text{NP} \).

So, the question is \( \text{NP} = \text{co-NP} \)?
Classes $P$, $NP$, and $co-NP$

Since $P$ is closed under complement, we have $P \subseteq NP \cap co-NP$.

There are 4 possible relations between $P$, $NP$ and $co-NP$. 

- $P = NP = co-NP$
- $NP = co-NP$
- $P\subseteq NP \cap co-NP$
- $NP \subseteq P \cap co-NP$
Reducibility

A language $L_1$ is \textit{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 $f(x) \in L_2$.

$f$ is the \textit{reduction function} and a polynomial-time algorithm $F$ that computes $f$ is called a \textit{reduction algorithm}.

\textit{Example}. \textsc{Equal-Numbers} $\leq_P$ \textsc{Same-Points}.

\textsc{Equal-Numbers}: Given $n$ integer numbers, are there two equal numbers?

\textsc{Same-Points}: Given $n$ points in the plane with integer coordinates, are there two coinciding points?
Reducibility

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

Proof. Consider the algorithm for $L_1$ that is combination of the reduction algorithm and the algorithm for $L_2$. 
A language $L$ is **NP-complete** if

1. $L \in \text{NP}$, and
2. $L' \leq_p L$ for every $L' \in \text{NP}$.

We say that a language is **NP-hard** if it satisfies condition 2 (1 may be satisfied or not). Any NP-complete problem is NP-hard.

Example of NP-hard problem that is not NP-complete (does not satisfy the condition 1): halting problem (given a program and its input, will it run forever?).

**Theorem.** If any NP-complete problem is solvable in polynomial time, then $P = \text{NP}$. Equivalently, if some problem in $\text{NP}$ is not solvable in polynomial time, then no NP-complete problem is polynomial-time solvable.
How most theoretical computer scientists view the relationships among P, NP, and NPC.
Circuit Satisfiability

A boolean combinatorial circuit is composed of AND, OR, and NOT gates connected by wires. Circuit input is a wire that is not the output of a gate. Circuit output is a wire that is not the input of a gate; it is unique. A truth assignment of a circuit is a set of boolean input values.

\[
\begin{array}{c|c}
  x & \neg x \\
  0 & 1 \\
  1 & 0 \\
\end{array}
\quad
\begin{array}{c|c|c}
  x & y & x \land y \\
  0 & 0 & 0 \\
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  1 & 1 & 1 \\
\end{array}
\quad
\begin{array}{c|c|c}
  x & y & x \lor y \\
  0 & 0 & 0 \\
  0 & 1 & 1 \\
  1 & 0 & 1 \\
  1 & 1 & 1 \\
\end{array}
\]

A boolean combinatorial circuit is satisfiable if it has a truth assignment that causes the output to be 1. This assignment is called a satisfying assignment.
Example. $x_1 = 1, x_2 = 1, x_3 = 0$ is a *satisfying assignment* and the circuit is satisfiable.
Circuit Satisfiability

Circuit-SAT problem. Given a boolean combinatorial circuit, is it satisfiable?

Lemma. Circuit-SAT is in NP.

Proof. Make a verification algorithm $A(x, y)$ where $x$ is an encoding of circuit and $y$ is the certificate (satisfying assignment). $A$ must be a polynomial-time algorithm.

Idea: for each logic gate $A$ computes its output based on inputs. At any moment, there is a gate with “ready” inputs. $A$ returns the final output (of entire circuit).
Lemma. \textsc{Circuit-SAT} is NP-hard.
Lemma. \textsc{Circuit-SAT} is \textit{NP}-hard.

Proof. Let \( L \) be a language in \textit{NP} and let \( A \) be a verification algorithm for \( A \). We want to design a polynomial-time algorithm that maps every \( x \in \{0,1\}^* \) to a circuit \( C = f(x) \) such that \( x \in L \) if and only if \( C \in \textsc{Circuit-SAT} \).

Idea: model the execution of the algorithm (code for \( A \), program counter, machine state, and working storage) with a circuit \( C \). There are \( T(n) = O(n^k) \) steps in the execution. We put \( T(n) \) copies of the circuit \( M \) that implements the computer hardware. We “run” the algorithm on the certificate \( y \). \( \square \)
Boolean formula satisfiability

An instance of Circuit-SAT can be transformed into a boolean formula \( \phi \) composed of

1. \( n \) boolean variables \( x_1, \ldots, x_n \)
2. \( m \) boolean connectives: any boolean function with one or two inputs and one output, such as \( \land \) (AND), \( \lor \) (OR), \( \neg \) (NOT), \( \rightarrow \) (implication. like \( \geq \)), \( \leftrightarrow \) (if and only if. like \( = \))
3. parentheses

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( y )</td>
<td>( x \rightarrow y )</td>
<td>( x \leftrightarrow y )</td>
</tr>
<tr>
<td>0 0</td>
<td>1 1</td>
<td>1 1</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>1 1</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 0</td>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

Example: \( \phi = ((x_1 \rightarrow x_2) \lor \neg((\neg x_1 \leftrightarrow x_3) \lor x_4)) \land \neg x_2. \)
A truth assignment for a boolean function \( \phi \) is a set of values of variables of \( \phi \) and a satisfying assignment is a truth assignment that causes it to evaluate to 1. If a satisfying assignment exists then \( \phi \) is satisfiable.

SAT problem. Given a boolean formula, is it satisfiable?

**Theorem.** SAT is NP-complete.

**Proof.** 1. We show that SAT is in NP: certificate is the truth assignment that satisfies the formula. Can be checked in polynomial time.
2. We show that Circuit-SAT ≤ P SAT. Assign a variable for each wire. Formula for SAT is the AND of the circuit output and the conjunction of clauses describing the operation of each gate. The formula can be produced in polynomial time. It is satisfiable if and only if the circuit is satisfiable.
A literal in a boolean formula is a variable or its negation. A boolean formula is in conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each of which is the OR of one or more literals. A boolean formula is in 3-conjunctive normal form, or 3-CNF, if each clause has exactly 3 distinct literals.

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

3-CNF-SAT, or 3-SAT, asks whether a given 3-CNF is satisfiable.

**Theorem.** 3-SAT is NP-complete.

**Proof.** 3-SAT\(\in\)NP is similar to the proof that SAT\(\in\)NP (every instance of 3-SAT is an instance of SAT).
We prove that $\text{SAT} \leq_{\text{p}} \text{3-SAT}$.

Let $\phi$ be an input formula for SAT, for example
$\phi = ((x_1 \rightarrow x_2) \lor \neg((\neg x_1 \leftrightarrow x_3) \lor x_4)) \land \neg x_2$.

Construct a binary “parse” tree. Introduce new variables $y_i$. 
3-CNF

Transform the tree into a formula $\phi'$:

$$
\phi' = y_1 \land (y_1 \leftrightarrow (y_2 \land \neg x_2)) \\
\land (y_2 \leftrightarrow (y_3 \lor y_4)) \\
\land (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
\land (y_4 \leftrightarrow \neg y_5) \\
\land (y_5 \leftrightarrow (y_6 \lor x_4)) \\
\land (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
$$
3-CNF

Make truth table for each clause $\phi'_i$.

Convert $\neg\phi'_i$ using 0’s in the table.

For example $\phi'_1 = y_1 \leftrightarrow (y_2 \land \neg x_2)$

\[
\begin{array}{ccc|c}
  y_1 & y_2 & x_2 & y_1 \leftrightarrow (y_2 \land \neg x_2) \\
  1   & 1   & 1   & 0 \\
  1   & 1   & 0   & 1 \\
  1   & 0   & 1   & 0 \\
  1   & 0   & 0   & 0 \\
  0   & 1   & 1   & 0 \\
  0   & 1   & 0   & 1 \\
  0   & 0   & 1   & 1 \\
  0   & 0   & 0   & 1 \\
\end{array}
\]

Then $\neg\phi'_1 = (y_1 \land y_2 \land x_2) \lor (y_1 \land \neg y_2 \land x_2) \lor (y_1 \land \neg y_2 \land \neg x_2) \lor (\neg y_1 \land y_2 \land \neg x_2)$
3-CNF

\[ \neg \phi_1' = (y_1 \land y_2 \land x_2) \lor (y_1 \land \neg y_2 \land x_2) \lor (y_1 \land \neg y_2 \land \neg x_2) \lor (\neg y_1 \land y_2 \land \neg x_2) \]

Applying DeMorgan’s laws

\[ \phi_1' = (\neg y_1 \lor \neg y_2 \lor \neg x_2) \land (\neg y_1 \lor y_2 \lor \neg x_2) \land (\neg y_1 \lor y_2 \lor x_2) \land (y_1 \lor \neg y_2 \lor x_2) \]

This way we create a CNF formula \( \phi'' \) so that each clause has at most 3 literals. We change those with 1 and 2 literals.
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 its vertices. *Clique Problem* is to find a clique of maximum size in $G$. Decision problem:

$\text{CLIQUE} = \{ \langle G, k \rangle : G \text{ is a graph with a clique of size } k \}$.

Naive algorithm needs $\Omega(k^2(\binom{|V|}{k}))$ time.

**Theorem.** The clique problem is NP-complete.

**Proof.** $\text{CLIQUE} \in \text{NP}$ if we choose the clique as the certificate.

$\text{CLIQUE}$ is NP-hard if we show $3\text{-SAT} \leq_P \text{CLIQUE}$. 

Oct 2017
Clique Problem

The reduction algorithm begins with an instance of 3-SAT problem \( \phi = C_1 \land C_2 \land \cdots \land C_k \). We construct a graph \( G(V, E) \). For each clause \( C_r = (l^r_1 \lor l^r_2 \lor l^r_3) \), we place 3 vertices \( v^r_1, v^r_2, v^r_3 \). We put an edge between \( v^r_i \) and \( v^s_j \) if \( r \neq s \) and literals of \( v^r_i \) and \( v^s_j \) are not contradictory.

Example: \( \phi = (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_2 \lor x_3) \).
We show that the transformation of $\phi$ is a reduction.

Suppose that $\phi$ has a satisfying assignment. Then each clause $C^r$ contains at least one literal $l_i^r$ assigned to 1. These vertices form a clique of size $k$ in $G$.

Conversely, suppose that $G$ has a clique $V'$ of size $k$. No edges in $G$ connect vertices in the same triple, so $V'$ contains exactly one vertex per triple. We assign 1 to each literal in the clique (we do not assign 1 to a literal and its complement). We assign any value to a variable whose literals are not in clique. All clauses are satisfied and $\phi$ is satisfied. □
Vertex-Cover Problem

A vertex cover of a graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices such that, if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both).

The vertex-cover problem is to find a vertex cover of minimum size in $G$.

Decision problem:

\[
\text{VERTEX-COVER} = \{\langle G, k \rangle : \text{graph } G \text{ has a vertex cover of size } k\}. 
\]

**Theorem.** The vertex-cover problem is NP-complete.

**Proof.** $\text{VERTEX-COVER} \in \text{NP}$ if we choose the vertex cover as the certificate.

$\text{VERTEX-COVER}$ is NP-hard if we show $\text{CLIQUE} \leq_{P} \text{VERTEX-COVER}$. 

Oct 2017
We define the *complement* of $G$ as $\overline{G} = (V, \overline{E})$ where

$$\overline{E} = \{(u, v) : u, v \in V, u \neq v \text{ and } (u, v) \notin E\}.$$  

An instance of $\langle G, k \rangle$ of the clique problem reduces to an instance $\langle \overline{G}, |V| - k \rangle$ of the vertex-cover problem. Indeed, a set $V' \subseteq V, |V'| = k$ is the clique of $G$ if and only if $V - V'$ is a vertex cover of $\overline{G}$. The reduction can be done in polynomial time.

$G$, clique $\{a, b, c, d\}$

$\overline{G}$, vertex cover $\{e, f\}$
**vertex-cover problem**

**Approx-Vertex-Cover**

1. $C = \emptyset$
2. $E' = E[G]$
3. while $E' \neq \emptyset$
4. let $(u, v)$ be any edge of $E'$
5. $C = C \cup \{u, v\}$
6. remove from $E'$ every edge incident on either $u$ or $v$
7. return $C$

**Theorem.** Approx-Vertex-Cover computes a 2-approximation in $O(V + E)$ time.
Hamiltonian-cycle Problem

A *hamiltonian cycle* in a graph \( G = (V, E) \) is a simple cycle that contains all vertices of \( G \). \( G \) is *hamiltonian* if it contains a hamiltonian cycle. *Hamiltonian-cycle problem* is to detect if a graph \( G \) is hamiltonian.

\[
\text{HAM-CYCLE} = \{ \langle G \rangle : G \text{ is a hamiltonian graph} \}.
\]

**Theorem.** The hamiltonian-cycle problem is NP-complete.

**Proof.** \( \text{HAM-CYCLE} \in \text{NP} \) if we choose the hamiltonian cycle as the certificate.

\( \text{HAM-CYCLE} \) is NP-hard if we show \( \text{VERTEX-COVER} \leq_p \text{HAM-CYCLE} \). Use *widgets* and *selector vertices* to transform \( G \) to \( G' \). Polynomial time. \( G \) has a vertex cover \( V^* \) if and only if \( G' \) has a hamiltonian cycle. \( \square \)
Traveling-salesman problem

Let $G$ be a complete graph with integer cost $c(i, j)$ to travel from city $i$ to $j$. Traveling-salesman problem is to find a minimum-cost tour (hamiltonian cycle).

![Graph with cities a, b, c, and d connected by edges with costs 1, 2, 3, 4, 5, and a traveling-salesman tour acbda of costs 7.]

TSP = $\{\langle G, c, k \rangle : G$ is a complete graph and has a traveling-salesman tour of cost at most $k\}$.

**Theorem.** The traveling-salesman problem is NP-complete.

**Proof.** TSP $\in$ NP if we choose the tour as the certificate.

Hardness: HAM-CYCLE $\leq_P$ TSP.
Given a finite set $S \subset \mathbb{N}$ (natural numbers or positive integers) and a target $t$, the \textit{subset-sum problem} is to find a subset $S' \subseteq S$ whose elements sum to $t$.

Example: $S = \{1, 3, 3, 9, 18\}$, $t = 10$. Then $S' = \{1, 9\}$ is a solution.

SUBSET-SUM = \{\langle S, t \rangle : \text{there exists a subset } S' \subseteq S \text{ such that } t = \sum_{s \in S'} s\}.

**Theorem.** The subset-sum problem is NP-complete.

**Proof.** SUBSET-SUM $\in$ NP if we choose $S'$ as the certificate. Hardness: 3-SAT $\leq_p$ SUBSET-SUM.
Subset-Sum Problem

Reduction. \( \phi = C_1 \land C_2 \land C_3 \land C_4 \) where \( C_1 = (x_1 \lor \neg x_2 \lor \neg x_3) \), \( C_2 = (\neg x_1 \lor \neg x_2 \lor \neg x_3) \), \( C_3 = (\neg x_1 \lor \neg x_2 \lor x_3) \), and \( C_4 = (x_1 \lor x_2 \lor x_3) \).

\[
\begin{array}{cccccccc}
   & x_1 & x_2 & x_3 & C_1 & C_2 & C_3 & C_4 \\
v_1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\
v_1' & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\
v_2 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
v_2' & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
v_3 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
v_3' & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
 s_1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
s_1' & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\
s_2 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
s_2' & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\
s_3 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
s_3' & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\
s_4 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
s_4' & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
t & 1 & 1 & 1 & 4 & 4 & 4 & 4
\end{array}
\]