1 Decision Problem as a Subset

**Definition 1** The set of all binary strings, is defined as \{0,1\}∗ = \{0,1,00,01,10,11,000,…\} For any \(\pi \subseteq \Sigma^*\), \(\bar{\pi}\) denotes its complement in the set \(\Sigma^*\).

Next, let’s give following examples of decision problems: Given a graph \(G = (V,E)\),

\[ ST = \{ < G, s, t > : \text{There is a path from } s \text{ to } t \text{ in graph } G \} \]

\(ST\), the complement of \(ST\) is given by:

\[ \bar{ST} = \{ < G, s, t > : \text{There is no path from } s \text{ to } t \text{ in graph } G \} \]

Therefore given any \(<G,s,t>\) finding whether \(<G,s,t>\in ST\) or not is a decision problem.

As another example, consider Linear Programming (LP) problem:

\[ LP = \{ <A,b> : \exists x, \text{s.t. } Ax \leq b \} \]

Note that for LP, \(\Sigma^*\) is the set of all possible \(<A,b>\) and \(\bar{LP}\) is given by

\[ \bar{LP} = \{ <A,b> : \forall x, Ax \not\leq b \} \]

Hence, given any \(<A,b>\in \Sigma^*\) deciding whether \(<A,b>\in LP\) or not is a decision problem.

Similarly, a linear optimization problem may also viewed as a decision problem. Consider following optimization problem:

\[ \max c^T x, \text{ subject to } Ax \leq b. \]

If we define \(\pi\) as

\[ \pi = \{ <A,b,c,t> : \text{There is a solution } x \text{ s.t. } Ax \leq b \text{ and } c^T x \geq t \} \]

and beginning from \(t = -\infty\) decide whether \(<A,b,c,t>\in \pi\) or not, we can find optimal solution \(t^*\) as the point where the answer “switches” from YES to NO.

2 Definition of Polynomial Time

If we denote a computational problem as \(\pi\), then the set of polynomial time problems, denoted by \(\mathcal{P}\), is defined as:

\[ \mathcal{P} = \{ \pi : \text{There is an algorithm to decide } \pi \text{ in polynomial time} \} \]

A necessary and sufficient condition for a problem to be an element of polynomial time is given by

\[ [\pi \in \mathcal{P}] \Leftrightarrow [\exists A \text{ s.t. running time of } A \text{ on all inputs of length } n \text{ is } \leq n^d \text{ and } \pi = \{ x \in \Sigma^* : A(x) = \text{YES} \}] \]

where \(d\) is a constant. The aforementioned algorithm \(A\) is called polynomial time acceptor for \(\pi\).

**Examples:** \(ST\) problem defined in previous part.

\(MWST = \{ <G,W> : \exists \text{ a spanning tree such that weight is } \leq W \}\), where \(MWST\) stands for “minimum cost spanning tree”.
3 Definition of Non-deterministic Polynomial Time

The following is a necessary and sufficient condition for a computational problem \( \pi \) to be an element of \( \mathcal{NP} \):

\[
\pi \in \mathcal{NP} \iff \exists B \text{ with running time } |x|^d \text{ and } \pi = \{ x \in \Sigma^* : \exists c_x, \text{ s.t. } B(x, c_x) = \text{YES} \},
\]

where \( c_x \) is called certificate (proof, hint).

**Example:** \( \pi = \text{SAT} \) (Satisfiability). Before defining SAT problem, we should state following

- Variables: \( x_1, x_2, \ldots, x_n \).
- Literals: \( l_1, l_2, \ldots, l_n \), where \( \forall i, l_i \in \{ x_i, \bar{x}_i \} \).
- Clauses: \( c_1, c_2, \ldots, c_n \), where \( \forall i, c_i = (l_{j_1} \lor \ldots \lor l_{j_n}) \).
- Formula: \( \Phi = c_1 \land \ldots \land c_n \).

Now we are ready to define SAT problem as follows:

\( \text{SAT} = \{ \Phi : \text{We can assign } 0, 1 \text{ to variables of } \Phi \text{ s.t. all clauses are satisfied} \} \),

where by satisfying we mean at least one one clause has a positive literal \( x_i \) assigned to 1, or a negative literal \( \bar{x}_j \) assigned to 0.

**Claim 1** \( \text{SAT} \in \mathcal{NP} \)

**Proof:** We construct an algorithm with following parameters:

**Input:** \(< \Phi, n \text{ bit vector } v >\).

**Procedure:** Check whether \( v \) satisfies \( \Phi \) or not.

**Output:** YES - NO.

A couple of observations next. First of all note that \( B \) is polynomial time. Indeed its running time is linear, since it simply scans thorough vector \( v \) and decides YES if it encounters a 1 and NO otherwise. Furthermore, if \( \Phi \in \text{SAT} \), then \( \exists v, \text{ s.t. } B(\Phi, v) = \text{YES} \), where \( v \) is a satisfying assignment. Moreover, if \( \Phi \not\in \text{SAT} \), then \( \forall v, B(\Phi, v) = \text{NO} \). Using all these there observations, we conclude that \( \text{SAT} \in \mathcal{NP} \).

Note that definition of \( \mathcal{NP} \) is asymmetric, i.e.

\[
\Phi \in \text{SAT} \Rightarrow \exists c_\Phi, \text{ which leads to acceptance.}
\]

\[
\Phi \not\in \text{SAT} \Rightarrow \forall c_\Phi, B \text{ rejects.}
\]

As another example of \( \mathcal{NP} \) problem, consider LP problem. Recall that we have

\( \text{LP} = \{ < A, b > : \text{ system } Ax \leq b \text{ is feasible.} \} \)

Certificate for LP is a feasible solution \( x \). An algorithm \( B \) calculates \( Ax \) and decides whether \( x \) is feasible or not. Running time of \( B \) is quadratic in length of \( x \) (by recalling the complexity of matrix vector multiplication).

Another example of a problem in \( \mathcal{NP} \) is the following:

\( \text{COMPOSITE} = \{ n \in \mathbb{N} : n \text{ is composite} \} \).

A certificate for COMPOSITE problem is \( n_1, n_2 \in \mathbb{N}, \text{ s.t. } n_1, n_2 \not\in \{ 1, n \} \text{ and } n_1n_2 = n \).

4 \( \mathcal{NP}, \text{ CO-} \mathcal{NP} \)

**Definition 2**

\[
[\pi \subseteq \Sigma^*, \text{ s.t. } \pi \in \text{CO-NP}] \iff [\pi \in \mathcal{NP}].
\]
As an example of a computational problem which is in $\text{CO-NP}$, consider $\text{PRIME} = \text{COMPOSITE}$. Since $\text{COMPOSITE} \in \text{NP}$ (as argued above), $\text{PRIME} \in \text{CO-NP}$. Therefore, for any given input $n \in \mathbb{N}$, we have certificate that states that $n$ is not a prime.

As another example of a problem in $\text{CO-NP}$, consider $\text{LP}$ defined as follows:

$$\text{LP} = \{<A, b>: Ax \leq b \text{ is not feasible}\}.$$ 

If $<A, b> \notin \text{LP}$, there exists a short certificate to verify $<A, b> \notin \text{LP}$, which implies that $\text{LP} \in \text{CO-NP}$.

**Claim 2** $\text{LP} \in \text{NP} \cap \text{CO-NP}$.

**Proof:** Consider following systems

$$Ax \leq b, \quad (1)$$

and

$$A^T y = 0, \quad y \geq 0, \quad b^T y < 0. \quad (2)$$

Using Farkas’ lemma, we know that either (1) or (2) is feasible but not both. In the same way as above, we can write a polynomial time acceptor for (2), so we know that detecting the infeasibility of an LP is in $\text{NP}$, or rather $\text{LP} \in \text{NP}$. Hence, we have $\text{LP} \in \text{NP} \cap \text{CO-NP}$.

Before discussing $\text{NP}$–completeness, we state two long standing open problems in computer science:

First, is $P = \text{NP} \cap \text{CO-NP}$?

Second, is $P = \text{NP}$?

It’s clear that $P \subseteq \text{NP} \cap \text{CO-NP}$ and that $P \subseteq \text{NP}$, but it’s unclear whether equality holds.

## 5 $\text{NP}$–Completeness

Intuitively, $\text{NP}$–complete problems may be considered as ‘the hardest’ problems in $\text{NP}$, in the sense that every problem in $\text{NP}$ can be reduced to a $\text{NP}$–complete problem in polynomial time. Before stating the definition of $\text{NP}$–completeness, we need following definition.

**Definition 3** $\pi, \pi \subseteq \Sigma^*$, $\pi'$ is reducible to $\pi$, denoted as $\pi' \leq \pi$, provided that

$$\exists f : \Sigma^* \rightarrow \Sigma^*, \text{ s.t. } x \in \pi' \iff f(x) \in \pi \text{ and } f \text{ runs in polynomial time}.$$ 

Next, we state the rigorous definition of $\text{NP}$–completeness:

**Definition 4** $\pi$ is $\text{NP}$–complete provided that:

1. $\pi \in \text{NP}$.

2. $\forall \pi' \in \text{NP}$, $\pi' \leq \pi$,

**Claim 3** If $\pi$ is $\text{NP}$–complete and $\pi \in \mathcal{P}$, then $\mathcal{P} = \text{NP}$.

**Proof:** Suppose we have an $\text{NP}$–complete $\pi$, such that $\pi \in \mathcal{P}$, which implies that $\exists A_\pi$, which runs in $n^d$. Using this $A_\pi$, we can construct the following algorithm for any $\pi' \in \text{NP}$:

Given input $x$ for $\pi'$,
- Compute $f(x)$
- Run $A_\pi$ on $f(x)$
- If $A_\pi(f(x)) = \text{YES}$, then output YES, if $A_\pi(f(x)) = \text{NO}$, then output NO.

Now, observe that from the definition of $\text{NP}$–completeness, the aforementioned algorithm works correctly; moreover, since $x \in \mathcal{P}$, we have $n^d$ as the running time of $A_\pi$ and $n^d$ for $f$ (recall the definition of reducibility). Hence, the algorithm’s running time is $n^{e+d}$, which is also polynomial. Therefore, we conclude that $\pi' \in \mathcal{P}$. If $\pi \in \mathcal{P}$ for any $\text{NP}$–complete $\pi$, then we have $\forall \pi' \in \text{NP}$, $\pi' \in \mathcal{P}$. Hence the claim follows. $\square$
6 Steps for $\mathcal{NP}$–Completeness Proofs

In this part, we will show that $SAT \leq 0 - 1 IP$, where $IP$ stands for ‘integer program’. We should construct a polynomial time algorithm with following properties:

**Input:** $\Phi$.
**Output:** A $0 - 1 IP$.

For each $x_i$ of $SAT$, we have $y_i \in \{0, 1\}$ for $0 - 1 IP$, where $y_i = 0$ means $x_i$ is false and $y_i = 1$ means $x_i$ is true. For any clause $c_j = \bigvee_{i \in P_j \subseteq [n]} x_i \lor \bigvee_{k \in N_j \subseteq [n]} \neg x_i$, we have $\sum_{i \in P_j} y_i + \sum_{k \in N_j} (1 - y_k)$ for $0 - 1 IP$.

Observe that the aforementioned algorithm is polynomial time. Therefore, we conclude $SAT \leq 0 - 1 IP$, which implies that $0 - 1 IP \in \mathcal{NP}$.

**Theorem 4** (Cook, Levin '71) SAT is $\mathcal{NP}$–complete.

Since $SAT \leq 0 - 1 IP$ (as we have argued above), $0 - 1 IP$ is also $\mathcal{NP}$–complete. Therefore, intuitively speaking it is also one of the ‘hardest problems in $\mathcal{NP}$'.