P and NP

Hsiang-Hsuan (Alison) Liu

1 Variants of Turing machines

There are different versions of Turing machines. In this chapter, we focus on two variants: multiple-tape Turing machines and non-deterministic Turing machines.

Multiple-tape Turing machines. A multiple Turing machine is like the ordinary Turing machine but with multiple tapes. Each tape has its own read-write head. At each step, the configuration of the multiple-tape Turing machine is defined by the state of control and the symbols reading by each read-write head. And the multiple-tape Turing machine can read, write, and move the head on some or all the tapes simultaneously.

Nondeterministic Turing machines. A non-deterministic Turing machine is similar to the ordinary Turing machine but with nondeterministic behavior. At any time, according to the current configuration, the nondeterministic Turing machine may proceed to several possibilities. That is, given an input, a nondeterministic Turing machine can have different sequences of configurations. Given an input string, as long as one of the sequences of configurations leads to accept configuration, the nondeterministic Turing machine accepts this input string. The configurations and the proceeding steps form a *computation tree* of the nondeterministic Turing machine.

1.1 Equivalence of Turing machines (optional)

Two Turing machines are *equivalent* if they recognize the same language. Surprisingly, adding multiple tapes or nondeterministic power does not make a Turing machine recognize more language. Actually, we can prove that the ordinary Turing machine, the multiple-tape Turing machine, and the nondeterministic Turing machine are equivalent.

Theorem 1. *Every multiple-tape Turing machine has an equivalent single-tape Turing machine.*

Proof. (Scketch.) We can show this by using a single-tape Turing machine S to simulate the multiple-tape Turing machine M.

Say that M has k tapes. The tape of S initially contains the initial information on the k tapes, separated by a special symbol #. Moreover, we put a dot over the symbol on the S tape (for example, $\dot{1}$) if the pointer of some tape of M is at the corresponding position.

For each step, S mimics what M does on the k tapes on its own tape. If M moves one of its tape heads to the left or to the right, S mimics it by changing the position of the dot. If M writes something on one of its tapes, S writes the same symbol on the corresponding position. Finally, if M changes its control state, S does the same change. □

Similarly, we can simulate the nondeterministic Turing machine by a multiple-tape (deterministic) Turing machine. Since there exists a single-tape Turing machine that is equivalent to the multiple-tape Turing machine, we can conclude that for any nondeterministic Turing machine, there is an equivalent (single-tape) deterministic Turing machine.

Theorem 2. *Every nondeterministic Turing machine has an equivalent multiple-tape Turing machine.*

Proof. (Scketch.) Given any nondeterministic Turing machine N, we can use a multiple-tape Turing machine M with three tapes to simulate it. The Turing machine M simulates the computation on one of the branches of the computation tree of N at a time. The first tape is for storing the initial input of the N and never changed. The second tape is the "working tape" for M to simulate the computation of N on one branch of its computation tree. The third tape stores the information of which branch is simulated now.

To avoid entering an infinite loop on some branch and losing the opportunity for entering the accepting configuration, the simulation should work in a breadth-first-search manner on the computation tree. □

2 Time complexity

Definition 1. *Let M be a deterministic Turing machine that halts on all inputs. The* time complexity *or* running time *of M is the function $f : \mathcal{N} \to \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses ton any input of length n.*

Although multiple-tape Turing machines and nondeterministic Turing machines all have the same power as (ordinary) Turing machines (that is, they all recognize the same set of languages), the time (or steps) they need to recognize a string can be different.

By analyzing the total number of steps needed for the worst case to simulate the variants, we get the following two theorems that concern the time complexity:

Theorem 3. *Let $f(n)$ be a function. Every $f(n)$ time multiple-tape Turing machine has an equivalent $O(f^2(n))$ time single-tape Turing machine.*

Theorem 4. *Let $f(n)$ be a function. Every $f(n)$ time nondeterministic Turing machine has an equivalent $2^{O(f(n))}$ time single-tape Turing machine.*

2.1 The class P

Definition 2. *The class* **P** *is the set of languages that are decidable in polynomial time on a deterministic single-tape Turing machine.*

To show that a language (problem) is in **P**, one should design a deterministic Turing machine (algorithm) that correctly decides the language and show that this Turing machine halts in polynomial time in the length of the input string on the tape.

2.2 The class NP

Definition 3. *The class* **NP** *is the set of languages that are decidable in polynomial time on a nondeterministic single-tape Turing machine.*

Similar to the class **P**, to show that a language (problem) is in **NP**, one should design a nondeterministic Turing machine (algorithm) that correctly decides the language and shows that this Turing machine halts in polynomial time in the length of the input string on the tape. It may be a bit difficult to think about the behavior of nondeterministic Turing machines. There is an alternative definition of the class **NP**:

Definition 4. *The class* **NP** *is the set of languages that can be verified in polynomial time.*

Here, we introduce two problems that are in **NP**, Clique, and Subset-Sum. The Clique problem is defined formally as follows:

$$\textsc{Clique} = \{\langle G, k\rangle \mid G \text{ is a undirected graph with a } k\text{-clique}\}$$

That is, given a (undirected) graph and a natural number k, a correct algorithm should return *yes* (or *accept*) if there is a k-clique in G.

The Subset-Sum problem is defined formally as follows:

$$\textsc{Subset-Sum} = \{\langle S, t\rangle \mid S = \{x_1, \cdots, x_n\} \text{ and for some } \{y_1, \cdots, y_k\} \subseteq \{x_1, \cdots, x_n\},$$

$$\text{we have } \sum_i y_i = t.$$

That is, given a set of numbers $S = \{x_1, \cdots x_n\}$ and a target number t, a correct algorithm should return *yes* (or *accept*) if there is a subset of S such that the sum of elements in the subset is equal to the target number t.

2.2.1 Verifier

The concept of *verify* uses an extra piece of information to check if a string is in the language. Solving some problems directly is difficult. But if someone somehow discovers the solution (perhaps using an exponential time algorithm), it is easy for us to be convinced that the answer is correct by being shown the solution.

Definition 5. *A* verifier *for a language A is an algorithm V, where*

$$A = \{w \mid V \textit{ accepts } \langle w, c\rangle \textit{ for some string } w.\}$$

The string c is called *certificate* or *proof* of the string $w \in A$. Notice that it is impossible for any string $w \notin A$ to have such a certificate.

For example, a subset of k vertices in G that form a clique is a good certificate for the Clique problem. For the Subset-Sum problem, a certificate can be a subset with sum that is equal to the target number.

Polynomial-time verifiable. Consider a language A and a string $w \in A$. If a verifier can be run in polynomial time in the length of w, A is *polynomial-time verifiable*. Notice that if the verifier can be run in polynomial time, the size of the certificate should also be polynomial in the size of w (otherwise, the verifier algorithm cannot run in polynomial time).

According to Definition 4, to show that a problem A is in **NP**, we should show that it is polynomial time verifiable. The proof consists of three parts:

1. Show that for every string $w \in A$, there is a certificate c.
2. Design a verifier (which is a Turing machine/algorithm) that accepts $\langle\langle w\rangle, c\rangle$ if $w \in A$, where c is the certificate of the string w.
3. Show that this verifier can be run in polynomial time in the length of w.

Now let's see an example for proving Clique is in **NP**.

Theorem 5. *Clique is in* **NP**.

Proof. Let string c that encodes a clique with size k in G be a certificate.

```
V = " On input ⟨⟨G, k⟩, c⟩ :
    1. Test whether $c$ is a set of $k$ nodes in $G$
    2. Test whether $G$ contains all edges connecting vertices in $c$
    3. If both 1 and 2 pass, accept; otherwise, reject."
```

Step 1 takes at most $|c| = k$ times of scanning through the input tape. Step 2 takes at most $|c|^2 = k^2$ times of scanning through the input tape. Hence, V runs in polynomial time in the input length. □

Next, we show that Subset-Sum is in **NP**.

Theorem 6. *Subset-Sum is in* **NP**.

Proof. Let string c that encodes a subset of S with sum t be a certificate.

```
V = " On input ⟨⟨S, t⟩, c⟩ :
    1. Test whether $|c| \le |S|$
    2. Test whether $c$ is a collection of numbers that sum to $t$
    3. Test whether $S$ contains all the numbers in $c$
    4. If all 1, 2, and e pass, accept; otherwise, reject."
```

□

Step 1 takes, at most, one time to scan through the input. Step 2 takes $|c| \le |S|$ summations. Step 3 takes at most $|c|$ times of scanning through the input. Hence, V runs in polynomial time in the input length.