Introduction to the Theory of Computation

Mahesh Viswanathan

Fall 2018

We will now review the basic definitions and theorems in the area of *computational complexity*, which tries to study various models of computation with the goal of understanding their relative computational power, and classify computational problems in terms of computational resources they need. Here, we will primarily consider time and space as the principal resources we will measure for an algorithm.

Recall that the computational problems one studies in the context of theoretical computer science are usually *decision problems*. Decision problems are those where given an input, one expects a boolean answer. Typically, input instances are encoded as strings over some alphabet of symbols. A decision problem partitions inputs into those for which the expected answer is "yes"/"true" and those for which the answer is "no"/"false". Therefore, a decision problem is often identified with a *language*, or a collection of strings, namely, those for which the problem demands a "yes" answer. Similarly, the machines we will define, will answer "yes"/"accept" or "no"/"reject" on input strings, and we associate a language $\mathbf{L}(M)$ with machine M, which is the collection of all strings it accepts. Given this interpretation of problems and machines, we will typically say that a machine M solves a problem L (or rather *accepts/recognizes*) if $L = \mathbf{L}(M)$, i.e., M answers "yes" on exactly the inputs that the problem demands the answer to be "yes".

The main model of computation that we will consider is that of a Turing machine. However before introducing this model, let us recall some of the notation on strings and languages that we will use.

Alphabet, Strings, and Languages. An *alphabet* Σ is a finite set of elements. A (finite) *string* over Σ is a (finite) sequence $w = a_0 a_1 \cdots a_k$ over Σ (i.e., $a + i \in \Sigma$, for all i). The *length* of a string $w = a_0 a_1 \cdots a_k$, denoted $|w|$, is the number of elements in it, which in this case is $k + 1$. The unique string of length 0, called the *empty string*, will be denoted by ε. For a string $w = a_0 a_1 \cdots a_k$, the ith symbol of the string a_i will be denoted as $w[i]$ [1]. For strings $u = a_0 a_1 \cdots a_k$ and $v = b_0 b_1 \cdots b_m$, their *concatenation* is the string $uv = a_0 a_1 \cdots a_k b_0 b_1 \cdots b_m$. The set of all (finite) strings over Σ is denoted by Σ^*; we will sometimes use Σ^i to denote the set of strings of length i. A *language* A is a set of strings, i.e., $A \subseteq \Sigma^*$. Given languages A, B, their concatenation $AB = \{uv \mid u \in A,\ v \in B\}$. For a language A, $A^0 = \{\varepsilon\}$, and A^i denotes the i-fold concatenation of A with itself, i.e., $A^i = \{u_1 u_2 \cdots u_i \mid \forall j.\ u_j \in A\}$. Finally, the *Kleene closure* of a language A, is $A^* = \bigcup_{i \geq 0} A^i$.

1 Turing Machines

We will now recall the definition of a Turing machine. Since we will use this model to define the time and space bounds during a computation, as well as define computable functions, the most convenient model to consider is that of a multi-tape Turing machine shown in Figure 1. Such a model has a read-only *input* tape, a write-only *output* tape, and finitely many read/write *work* tapes. Intuitively, the machine works as follows. Initially, the input string is written out on the input tape, and all the remaining tapes are *blank*. The tape head are scanning the leftmost cell of each tape, which we will refer to as cell 0. This cell contains a special symbol $\triangleright$ in every tape (except the output tape). This is the *left end marker*, which helps the machine realize which cell is the leftmost cell. We will assume these cells are never overwritten by any other

[1] Here we are assuming the the 0th symbol is the "first".

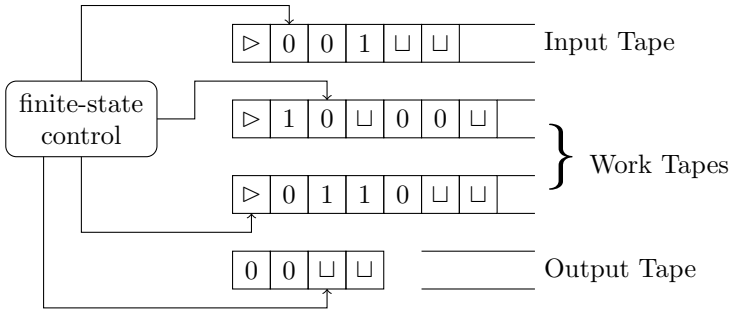


Figure 1: Multi-tape Turing machine, with a read-only input tape, finitely many read/write worktapes, and a write-only output tape.

symbol, and whenever $\triangleright$ is read on a particular tape, the tape head of the Turing machine will move right. At any given step of the Turing machine does the following. Based on the current state of its finite control, and symbols scanned by each tape head, the machine will change the state of its finite control, write new symbols on each of the worktapes, and move it's heads on the input and worktapes either one cell to the left or one cell to the right. During the step, the machine may also choose to write some symbol on its output tape. If it writes something on the output tape, then the output tape head moves one cell to the right. If it does not write anything, then the output tape head does not move. We will assume that the machine has two special *halting* states — q_{acc} and q_{rej} — with the property that the machine cannot take any further steps from these states. These are captured in the formal definition of deterministic Turing machines below.

Definition 1. A *deterministic Turing machine with k-worktapes* is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \triangleright)$ where

- Q is a finite set of control states
- Σ is a finite set of input symbols
- $\Gamma \supseteq \Sigma$ is a finite set of tape symbols. We assume that $\{\sqcup, \triangleright\} \subseteq \Gamma \setminus \Sigma$.
- $q_0 \in Q$ is the initial state
- $q_{\mathsf{acc}} \in Q$ is the accept state
- $q_{\mathsf{rej}} \in Q$ is the reject state, with $q_{\mathsf{rej}} \neq q_{\mathsf{acc}}$, and
- $\delta : (Q \setminus \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}) \times \Gamma^{k+1} \to Q \times \{-1, +1\} \times (\Gamma \times \{-1, +1\})^k \times (\Gamma \cup \{\varepsilon\})$ is the transition function; here lft indicates moving the head one position to the left and $+1$ indicates moving the head one position to the right. We will assume that if $\delta(p, \gamma_0, \gamma_1, \ldots \gamma_k) = (q, d_0, \gamma'_1, d_1, \gamma'_2, d_2, \ldots \gamma'_k, d_k, o)$, for any $i \in \{0, 1, \ldots k\}$, if $\gamma_i = \triangleright$ then $\gamma'_i = \triangleright$ and $d_i = +1$.

We will now formally describe how the Turing machine computes. For this we begin by first indentifying information about the Turing machine that is necessary to determine it's future evolution. This is captued by the notion of a *configuration*. A single step of a Turing machine depends on all the factors that determine which transition is taken. This clearly includes the control state, and the symbols being read on the input tape and each of the worktapes. However this is not enough. The contents of the worktape change, and what is stored influences what will be read in a future step. Thus we need to know what is stored in each cell of such tapes. Since the input tape is read-only, its contents remain static and so we don't need to carry around its contents. We also need to know the position of each tape head, because that determines what is read in this step, how the contents of a tape will change based on the current step, and what will be read in the future as the heads move. Because of all of these observations, a configuration of a Turing machine is

taken to be the control state, the position of the input head, the contents of each worktape, and the position of each worktape head. The worktape contents and head position is often represented as a single string where a special marker indicates the head position. These are captured formally by the definition below.

Definition 2 (Configurations)**.** A *configuration* C of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \triangleright)$ is a member of the set $Q \times \mathbb{N} \times (\Gamma^*\{*\}\Gamma\Gamma^* \sqcup^\omega \times \mathbb{N})^k$ [2], where we assume that $* \notin \Gamma$ indicates the position of the head. For example, a configuration $\mathrm{C} = (q, i, u_1 * a_1 v_1 \sqcup^\omega, u_2 * a_2 v_2 \sqcup^\omega, h_2)$ is the configuration of a 2-worktape Turing machine, whose control state is currently q, the input head is scanning cell i, worktape i ($i \in \{1, 2\}$) contains u_i to left of the head, head is scanning symbol a_i and $v_i \sqcup^\omega$ are the contents of cells to the right of the head.

The *initial configuration* (the configuration of the Turing machine when it starts) is $(q_0, 0, * \triangleright \sqcup^\omega, \ldots, * \triangleright \sqcup^\omega)$. An *accepting configuration* is a member of the set $\{q_{\mathsf{acc}}\} \times \mathbb{N} \times (\Gamma^*\{*\}\Gamma\Gamma^*\sqcup^\omega)^k$. In other words, it is a configuration whose control state is q_{acc}. A *halting configuration* is a configuration whose control state is either q_{acc} or q_{rej}, i.e., it is a member of the set $\{q_{\mathsf{acc}}, q_{\mathsf{rej}}\} \times \mathbb{N} \times (\Gamma^*\{*\}\Gamma\Gamma^*\sqcup^\omega)^k$.

Having defined configurations, we can formally define how configurations change in a single step of the Turing machine. We begin by defining a function that updates the worktape. For a worktape $u * av\sqcup^\omega$, $\mathsf{upd}(u * av\sqcup^\omega, b, d)$ is the resulting worktape when b is written and the head is moved in direction d. This can be formmaly defined as

$$\mathsf{upd}(u * av\sqcup^\omega, b, d) = \begin{cases} ub * \sqcup\sqcup^\omega & \text{if } d = +1 \text{ and } v = \varepsilon \\ ub * cv'\sqcup^\omega & \text{if } d = +1 \text{ and } v = cv' \\ u' * cbv\sqcup^\omega & \text{if } d = -1 \text{ and } u = u'c \end{cases}$$

Recall also that for a finite string $w \in \Gamma^*$, $w[i]$ denotes the ith symbol in the string. We can extend this notion to tape contents that are sequences of the form $w\sqcup^\omega$ as follows.

$$w\sqcup^\omega[i] = \begin{cases} w[i] & \text{if } i < |w| \\ \sqcup & \text{otherwise} \end{cases}$$

Definition 3 (Computation Step)**.** Consider configurations $\mathrm{C}_1 = (q_1, i_1, u_1 * a_1 v_1, \ldots u_k * a_k v_k)$ and $\mathrm{C}_2 = (q_2, i_2, t_1, \ldots t_k)$ of Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \triangleright)$. Let the input string be w. We say $\mathrm{C}_1 \overset{o}{\longmapsto} \mathrm{C}_2$ (machine M moves from configuration C_1 to C_2 in one step and writes o on the output tape) if the following conditions hold. Let $\delta(q_1, w\sqcup^\omega[i_1], a_1, \ldots a_k) = (p, d_0, b_1, d_1, \ldots b_k, d_k)$. Then,

- $q_2 = p$, and $i_2 = i_1 + d_1$,
- for each i, $t_i = \mathsf{upd}(u_i * a_i v_i, b_i, d_i)$

When the output symbol written during a step is not important, we will write $\mathrm{C}^{(1)} \longmapsto \mathrm{C}^{(2)}$ to indicate a step from $\mathrm{C}^{(1)}$ to $\mathrm{C}^{(2)}$.

Having defined how the configuration of a Turing machine changes in each step, we can define the result of a computation on an input.

Definition 4 (Computation)**.** A *computation* of Turing machine M on input w, is a sequence of configurations $\mathrm{C}_1, \mathrm{C}_2, \ldots \mathrm{C}_m$ such that C_1 is the initial configuration of M, and for each i, $\mathrm{C}_i \longmapsto \mathrm{C}_{i+1}$.

Definition 5 (Acceptance)**.** An input w is *accepted* by Turing machine M if there is a computation $\mathrm{C}_1, \mathrm{C}_2, \ldots \mathrm{C}_m$ such that C_m is an accepting configuration.

The *language recognized/accepted* by M is $\mathbf{L}(M) = \{w \mid w \text{ is accepted by } M\}$. We say that a language $A \subseteq \Sigma^*$ is *accepted/recognized* by M if $\mathbf{L}(M) = A$.

[2] $\sqcup^\omega$ is an *infinite* sequence of blank symbols. Recall that almost all cells contain $\sqcup$, and so the tape contents is a string of the form $u\sqcup^\omega$, where u is initial portion of the tape containing some non-blank symbols.

Definition 6 (Halting)**.** A Turing machine M is said *halt* on input w if there is a computation $\mathrm{C}_1, \mathrm{C}_2, \ldots \mathrm{C}_m$ such that C_m is a halting configuration.

The Turing machine model we introduced with an output tape can be used to compute (partial) functions as follows.

Definition 7 (Function Computation)**.** The *partial function* computed by a Turing machine M, denoted $\mathbf{f}_M$, is as follows. If on input w, M has a halting computation $\mathrm{C}_1 \overset{o_1}{\longmapsto} \mathrm{C}_2 \overset{o_2}{\longmapsto} \cdots \overset{o_{m-1}}{\longmapsto} \mathrm{C}_m$ then $\mathbf{f}_M(w)$ is defined and equal to $o_1 o_2 \cdots o_{m-1}$. On inputs w such that M does not halt, $\mathbf{f}_M(w)$ is undefined.

We say that a (partial) function g is *computable* if there is a Turing machine M such that for every w, $g(w)$ is defined if and only if $\mathbf{f}_M(w)$ is defined, and whenever $g(w)$ is defined, $g(w) = \mathbf{f}_M(w)$.

Most of the time we will be considering Turing machines that accept or recognize languages, rather than those that compute functions. In this context, the symbols written on the output tape don't matter, and so we will often ignore the output tape when describing transitions and computations of such machines.

2 Church-Turing Thesis

The Turing machine model introduced in the previous section, is a cannonical model to capture mechanical computation. The Church-Turing thesis embodies this statement by saying that anything solvable using a mechanical procedure can be solved using a Turing machine. Our belief in the Church-Turing thesis is based on decades of research in alternate models of computation, which all have turned out to be computationally equivalent to Turing machines. Some of these models include the following.

- Non-Turing machine models: Random Access Machines, λ-calculus, type 0 grammars, first-order reasoning, π calculus, ...
- Enhanced Turing machine models: Turing machines with multiple 2-way infinite tapes, nondeterministic Turing machines, probabilistic Turing machines, quantum Turing machines, ...
- Restricted Turing machine models: Single tape Turing machines, Queue machines, 2-stack machines, 2-counter machines, ...

We will choose to highlight two of these results, that will play a role in our future discussions. The first is the observation that a one worktape Turing machine is computationally as powerful as the multi-worktape model introduced in Definition 1.

Theorem 8. *For any k worktape Turing machine M, there is a Turing machine with a single worktape $\mathit{single}(M)$ such that $\mathbf{L}(M) = \mathbf{L}(\mathit{single}(M))$ and $\mathbf{f}_M = \mathbf{f}_{\mathit{single}(M)}$* [3]*.*

Proof of Theorem 8 can be found in any standard textbook and its precise details are skipped. The idea behind the proof is as follows. The single worktape machine single(M) will simulate the steps of the k-worktape machine M on any input. But in order to simulate M, single(M) needs to keep track of M's configuration at each step. That means keeping track of M's state, its worktape contents, and its tape head. This single(M) accomplishes by storing M's state in its own state, and the contents of all k worktapes of M (including the head positions) on the single worktape of single(M). In general, cell i of the single worktape, stores cell $(i \div k) + 1$ of tape $i \bmod k$; here $i \div m$ denotes the quotient when i is divided by m and $i \bmod m$ denotes the remainder. Then to simulate a single step of M, single(M) will make multiple passes over its single worktape, to first identify the symbols on each tape read by M to determine the transition to take, and then update the contents of the tape according to the transition.

The second result relates to the nondeterministic Turing machines. The Turing machine model introduced in Definition 1 is *deterministic*, in the sense that at any given time during the computation of the machine,

[3] For partial functions f and g, we write $f = g$ to indicate that f and g have the same domains (i.e., they are defined for exactly the same elements), and further when $f(x)$ is defined, $f(x) = g(x)$.

there is at most on possible transition the machine can take. *Nondeterminism*, on the other hand, is the computational paradigm where the computing device, at each step, may have multiple possible transitions to *choose from*. As a consequence, on a given input the machine may have multiple computations, and the machine is said to accept an input, if any one of these computations leads to an accepting configuration. Formally, we can define a nondeterministic Turing machine as follows.

Definition 9. A *nondeterministic Turing machine with k-worktapes* (and one input tape [4]) is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \triangleright)$, where $Q, \Sigma, \Gamma, q_0, q_{\mathsf{acc}}, q_{\mathsf{rej}}, \sqcup, \triangleright$ are just like that for deterministic Turing machine, and

$$\delta : (Q \setminus \{q_{\mathsf{acc}}, q_{\mathsf{rej}}\}) \times \Gamma^{k+1} \to 2^{Q \times \{-1,+1\} \times (\Gamma \times \{-1,+1\})^k}$$

is the transition function. The transition function, given current state and symbols read on the input and worktapes, returns a set of possible next states, direction to move the input head, and symbols to be written and direction to move the head in for each worktape.

The definition of configurations, initial configuration, accepting and halting configurations is the same as in Definition 2. The definitions of computation step (Definition 3), computation (Definition 4), and acceptance and language recognized (Definition 5) are also the same. Hence we skip defining these formally.

Every deterministic Turing machine is a special kind of nondeterministic machine, namely, one which has the property that at each time step there is at most one transition enable. One of the important results concerning nondeterministic Turing machines is that the converse is also true, i.e., nondeterministic Turing machines are not more powerful than deterministic Turing machines.

Theorem 10. *For every nondeterministic Turing machine N, there is a deterministic Turing machine $det(N)$ such that $\mathbf{L}(N) = \mathbf{L}(det(N))$.*

A detailed proof of Theorem 10 is skipped. It can be found in any standard textbook in theory of computation. The broad idea behind the result is the observation that once the length of computation, and the nondeterministic choices at each step are fixed, a deterministic machine can simulate N for that length, on those choices. Thus, the deterministic Turing machine $\det(N)$ simulates N for increasingly longer computations, and for each length, $\det(N)$ will cycle through all possible nondeterministic choices at each step. If any of these computations is accepting for N, then $\det(N)$ will halt and accept.

3 Recursive and Recursively Enumerable Languages

The Church-Turing thesis establishes the cannonicity of the Turing machine as a model of mechanical computation. The collection of problems solvable on Turing machines is, therefore, worthy of study. It worth noting that when a Turing machine M is run on an input string w there are 3 possible outcomes — M may (halt and) accept w, M may (halt and) reject w, or M may not halt on w (and therefore not accept). Depending on how a Turing machine behaves we can define two different classes of problems solvable on a Turing machine.

Definition 11. A language A is *recursively enumerable/semi-decidable* if there is a Turing machine M such that $A = \mathbf{L}(M)$.

A language A is *recursive/decidable* if there is a Turing machine M that halts on *all* inputs and $A = \mathbf{L}(M)$.

Observe that when a problem A is recursive/decidable, it has a special algorithm that solves it and in addition always halts, i.e., on inputs not in A, this specialized algorithm explicitly rejects. Thus, by definition, every recusive language is also recursively enumerable.

Proposition 12. *If A recursive then A is recursively enumerable.*

[4] We assume there is no output tape for a nondeterministic Turing machine since such machines are used for function computation.

We will denote the collection of recursive languages as REC and the collection of all recursively enumerable languages as RE; thus, Proposition 12 can be seen as saying that REC $\subseteq$ RE. The collection of recursive and recursively enumerable languages enjoy some closure properties that are worth recalling.

Theorem 13. *REC is closed under all Boolean operations while RE is closed under monotone Boolean operations. That is,*

- *If $A, B \in$ RE, then $A \cup B$ and $A \cap B$ are also in RE.*
- *If $A, B \in$ REC, then $\overline{A}$, $A \cup B$, and $A \cap B$ are all in REC.*

Proof. We will focus on the two most interesting observations in Theorem 13; the rest we leave as an exercise to the reader. The first observation we will prove is the closure of RE under union. Let us assume M_A and M_B are Turing machines recognizing A and B, respectively. The computational problem $A \cup B$ asks one to determine if a given input string w belongs to either A or B. We could determine membership in A and B by running M_A and M_B, respectively, but we need to be careful about *how* we run M_A and M_B. Suppose we choose to first run M_A on w and *then* run M_B on w, then we could run into problems. For example, consider the situation where M_A does not halt on w, but $w \in B$. Then, running M_A followed by M_B will never run M_B and therefore never accept, eventhough $w \in A \cup B$. Switching the order of running M_A and M_B also does not help. What one needs to instead do is, to run M_A and M_B *simultaneously* on w. How does one M_A and M_B at the same time? There are many ways to achieve this. One way is to initially run one step of M_A and then one step of M_B on w from the initial configuration. If either them accept, the algorithm for $A \cup B$ accepts. If not, it will run M_A for two steps, and M_B for two steps, again starting from the respective initial configurations. Again, the algorithm for $A \cup B$ accepts if either simulation accepts. If not the computations of M_A and M_B are increased by one more step, and this process continue, until at some point one of them accepts.

The second result we would like to focus on is the observation that REC is closed under complementation. Let $A \in$ REC and let M be a Turing machine that halts on all inputs and $\mathbf{L}(M) = A$. The algorthm $\overline{M}$ for $\overline{A}$, runs M on input w, and if M accepts it rejects and if M rejects then it accepts. Notice that $\mathbf{L}(\overline{M}) = \overline{A}$ only because M halts on all inputs — if M does on halt on (say) w, then $w \in \overline{A}$ but $\overline{M}$ would never acceot w! □

The following theorem is a useful way to prove that a problem is decidable.

Theorem 14. *A is recursive if and only if A and $\overline{A}$ are recurisvely enumerable.*

Proof. If $A \in$ REC then $\overline{A} \in$ REC by Theorem 13. Then both A and $\overline{A}$ are recursively enumerable by Proposition 12.

Conversely, suppose A and $\overline{A}$ are recognized by M_A and $M_{\overline{A}}$ respectively. The recursively algorithm M for A, on a given input w, will run both M_A and $M_{\overline{A}}$ simultaneously (as in the proof of Theorem 13), and accept if either M_A accepts or $M_{\overline{A}}$ rejects. Notice, that any given input w belongs to either A or $\overline{A}$, and therefore at least one out of M_A and $M_{\overline{A}}$ is guaranteed to halt on each input. Therefore M will always halt. □

Encodings. Every object (graphs, programs, Turing machines, etc.) can be encoded as a binary string. The details of the encoding scheme itself are not important, but it should be simple enough that the data associated with the object should be easily recoverable by reading the binary encoding. For example, one should be able to reconstruct the vertices and edges of a graph from its encoding, or one should be able to reconstruct the states, transitions, etc. of a Turing machine from its encoding. For a list of objects $O_1, O_2, \ldots O_n$, we will use $\langle O_1, O_2, \ldots O_n \rangle$ to denote their binary encoding. In particular, for a Turing machine M, $\langle M \rangle$ is its encoding as binary string. Conversely, for a binary string x, M_x denotes the Turing machine whose encoding is the string x.

Once we establish an encoding scheme, we can construct a *Universal Turing machine*, which is an *interpreter* that given an encoding of a Turing machine M and an input w, can simulate the execution of M on

the input string w. This is an extremely important observation that establishes the recursive enumerability of the membership problem for Turing machines.

Theorem 15. *There is a Turing machine U (called the universal Turing machine) that recognizes the language $\mathsf{MP} = \{\langle M, w\rangle \mid w \in \mathbf{L}(M)\}$. In other words, $\mathsf{MP} \in RE$.*

All decision problems/languages are not recursively enumerable. Using Cantor's diagonalization technique, one can establish the following result.

Theorem 16. *The language $\overline{\mathsf{K}} = \{x \mid x \notin \mathbf{L}(M_x)\}$ is not recursively enumerable.*

Proof. The proof of Theorem 16 relies on a diagonalization argument to show that the language of every Turing machine differs from $\overline{\mathsf{K}}$, and therefore $\overline{\mathsf{K}}$ is not recusrively enumerable.

Consider an arbitrary Turing machine M_x whose encoding as a binary string is x. We will show that $\mathbf{L}(M_x) \neq \overline{\mathsf{K}}$, thereby proving the theorem. Observe that if $x \in \mathbf{L}(M_x)$ then by definition $x \notin \overline{\mathsf{K}}$ and if $x \notin \mathbf{L}(M_x)$ then again by definition $x \in \overline{\mathsf{K}}$. Therefore $x \in (\overline{\mathsf{K}} \setminus \mathbf{L}(M_x)) \cup (\mathbf{L}(M_x) \setminus \overline{\mathsf{K}}) \neq \emptyset$. □

4 Reductions

Theorem 16 is the first result that establishes that there are problems that are computationally difficult. Further results on the computational hardness of problems are usually established using the notion of *reductions*. Reductions demonstrate how one problem can be converted into another in such a way that a solution to the second problem can be used to solve the first. Formally, it is defined as follows.

Definition 17. A *(many-one/mapping) reduction* from A to B is a computable (total) function $f : \Sigma^* \to \Sigma^*$ sush that for any input string w,

$$w \in A \text{ if and only if } f(w) \in B$$

In this case, we say A is *(many-one/mapping) reducible* to B and we denote it by $A \leq_m B$.

Since many-one/mapping reductions are the only form of reduction we will stufy, we will drop the adjective "many-one" and "mapping" and simply call these reductions. Let us look at a couple of examples of reductions.

Example 18. Let us consider the complement of MP, i.e., $\overline{\mathsf{MP}} = \{\langle M, w\rangle \mid w \notin \mathbf{L}(M)\}$. One can show that $\overline{\mathsf{K}} \leq_m \overline{\mathsf{MP}}$ as follows. The reduction f is the following function: $f(x) = \langle x, x\rangle$.

To prove that f is a reduction, we need to argue two things. First that f is computable, i.e., we need to come up with a Turing machine M_f that always halts and produces the string $f(x)$ on input x. In this example, to construct $f(x)$, we simply need to "copy" the string x which clearly is a computable function. Second we need to argue that $x \in \overline{\mathsf{K}}$ iff $f(x) \in \overline{\mathsf{MP}}$. This can be argues as follows: $x \in \overline{\mathsf{K}}$ iff $x \notin \mathbf{L}(M_x)$ (definition of $\overline{\mathsf{K}}$) iff $\langle x, x\rangle \in \overline{\mathsf{MP}}$ (definition of $\overline{\mathsf{MP}}$) iff $f(x) \in \overline{\mathsf{MP}}$ (definition of f).

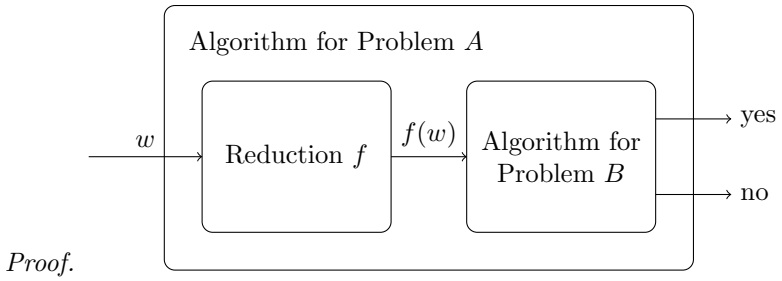
Example 19. Consider the problem

$$\overline{\mathsf{HP}} = \{\langle M, w\rangle \mid M \text{ does not halt on } w\}.$$

We will prove that $\overline{\mathsf{K}} \leq_m \overline{\mathsf{HP}}$.

Given a binary string x, let us consider the following program H_x.

```
H_x(w)
    result = M_x(x)
    if (result = accept)
        return accept (* on input w *)
    else
        while true do
```



Proof.

Figure 2: Schematic argument for Theorem 20.

In other words, the program H_x on input w, ignores its input and runs the program M_x on x. If M_x halts and accepts x then H_x halts and accepts w. Otherwise, H_x does not halt. Thus, the program H_x halts on some (all) inputs if and only if $x \in \mathbf{L}(M_x)$.

Let us now describe the reduction from $\overline{\mathsf{K}}$ to $\overline{\mathsf{HP}}$: $f(x) = \langle H_x, x \rangle$. Observe first that f satisfies the properties of a reduction because $x \in \overline{\mathsf{K}}$ iff $x \notin \mathbf{L}(M_x)$ iff H_x does not halt on x (and all input strings) iff $\langle H_x, x \rangle \in \overline{\mathsf{HP}}$. To establish that f is a reduction, we also need to argue that f is computable. On input string x, we need a program that produces the source code for H_x (given above) and copies the string x after the source code. This is clearly computable.

Reductions are a way for one to compare the computational difficulty of problems — if A reduces to B then A is at most as difficult as B, or B is at least as difficult as A. This is formally captured in the following proposition.

Theorem 20. *If $A \leq_m B$ and B is recursively enumerable (recursive) then A is recursively enumerable (recursive).*

Let f be a reduction from A to B that is computed by Turing machine M_f, and let M_B be a Turing machine that recognizes B. The algorithm for A is schematically shown in Figure 2 — on input w, compute $f(w)$ using M_f and run M_B on $f(w)$. Notice that this algorithm always halts if M_B always halts. Thus, if B is recursive then A is also recursive. □

Theorem 20 can be seen to informally say "if A reduces to B and B is computationally easy then A is computationally easy". It is often used in the contrapositive sense, it is useful to explicitly state this observation.

Corollary 21. *If $A \leq_m B$ and A is not recursively enumerable (undecidable) then B is not recursively enumerable (undecidable).*

We can use the above corollary to argue the computational hardness of some problems.

Theorem 22. $\overline{\mathsf{MP}}$ *is not recursively enumerable. Therefore,* MP *is undecidable.*

Proof. Example 18 establishes that $\overline{\mathsf{K}} \leq_m \overline{\mathsf{MP}}$. Together with Theorem **??** and Corollary 21, we can conclude that $\overline{\mathsf{MP}}$ is not recursively enumerable. Finally, since $\overline{\mathsf{MP}}$ is not recursively enumerable, Theorem 14 establishes that MP is not decidable/recursive. □

Since $\mathsf{MP} \in$ RE (Theorem 15) and $\overline{\mathsf{MP}} \notin$ RE (Theorem 22), we have a witness to the fact that RE is not closed under complementation. Just like *Theorem* 22, we could establish similar properties for the *halting problem*.

Theorem 23. $\overline{\mathsf{HP}}$ *is not recursively enumerable. Therefore,* $\mathsf{HP} = \{\langle M, w \rangle \mid M \text{ halts on } w\}$ *is undecidable.*

Proof. Follows from Example 19 and the argument in the proof of Theorem 22. □

Reductions are transitive and hence a *pre-order*; thus, the use of $\leq$ to denote them is justified.

Theorem 24. *The following properties hold for reductions.*

- *If* $A \leq_m B$ *then* $\overline{A} \leq_m \overline{B}$.
- *If* $A \leq_m B$ *and* $B \leq_m C$ *then* $A \leq_m C$.

Proof. If f is a reduction from A to B, then one can argue that f is also a reduction from $\overline{A}$ to $\overline{B}$. And, if f is a reduction from A to B and g a reduction from B to C then $g \circ f$ is a reductino from A to C. Establishing these observations to prove the theorem is left as an exercise. □

Having found a lens to compare the computational difficulty of two problems (namely, reductions), one can use them to argue that a problem is at least as difficult as a whole collection of problems, or something is the "hardest" problem in a collection. This leads us to notions of hardness and completeness.

Definition 25. A language A is RE-hard if for every $B \in \text{RE}$, $B \leq_m A$.
A language A is RE-complete if A is RE-hard and $A \in \text{RE}$.

Thus, an RE-complete problem is the hardest problem that is recursively enumerable, while an RE-hard problem is something that is at least as hard as any other RE problem. Are there examples of such problems? It turns out that MP, HP, and K are all RE-complete. We establish this for MP in the following theorem.

Theorem 26. MP *is RE-complete.*

Proof. Membership in RE has been established in Theorem 15. So all we need to prove is the hardness. Let B be any recursively enumerable language, and let M be a Turing machine recognizing B. The reduction from B to MP is as follows: $f(w) = \langle M, w \rangle$. It is easy to see that $w \in B$ iff $w \in \mathbf{L}(M)$ (since M recognizes B) iff $\langle M, w \rangle \in \mathsf{MP}$ (definition of MP) iff $f(w) \in \mathsf{MP}$ (definition of f). It is also easy to see that f is computable — in order to compute $f(w)$, all we need to do is prepend the source code of M. □

Establishing RE-hardness of a problem is sufficient to guarantee it's undecidability.

Theorem 27. *If* A *is RE-hard then* A *is undecidable.*

Proof. If A is RE-hard then since $\mathsf{MP} \in \text{RE}$, we have $\mathsf{MP} \leq_m A$. Since MP is undecidable (Theorem 22), by properties of a reduction (Corollary 21) A is undecidable. □