Chapter
6

NP, PCP and Non-approximability Results

IN THIS chapter we first present a formal treatment of the complexity concepts introduced in Chap. 1. This treatment will provide the reader with a precise machine-based characterization that will be used in Sect. 6.3 to develop the notion of *probabilistically checkable proof* (in short, PCP). In Sect. 6.4 we will see how probabilistically checkable proofs can be used in a rather surprising way to show non-approximability results for NP-hard optimization problems.

6.1 Formal complexity theory

Historically, all basic concepts of the theory of computational complexity have been stated in terms of *Turing machines*. Let us introduce this machine model in the form of an *acceptor*, a simple form needed for discussing the complexity of recognition or decision problems.

6.1.1 Turing machines

A Turing machine M can be seen as a computing device (see Fig. 6.1) provided with:

1. A set Q of internal states, including a *start* state q_S and an *accepting* state q_A.

2. An infinite memory, represented by an infinite *tape*[1] consisting of *cells*, each of which contains either a symbol in a work alphabet Γ or the special *blank* symbol $\square$.

3. A *tape head* that spans over the tape cells and at any moment identifies the current cell.

4. A finite control (program) Δ whose elements are called *transition rules*: any such rule $((q_i, a_k), (q_j, a_l, r))$ specifies that if q_i is the current state and a_k is the symbol in the cell currently under the tape head, then a computing step can be performed that makes q_j the new current state, writes a_l in the cell, and either moves the tape head to the cell immediately to the right (if $r = \text{R}$) or to the left (if $r = \text{L}$) or leaves the tape head on the same cell (if $r = \text{N}$).

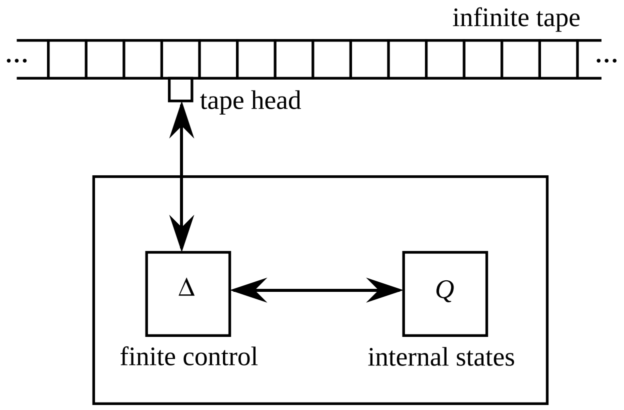

Figure 6.1
A Turing machine

We can view M as a computer with a fixed single program. The software is thus represented by the set of transition rules and the hardware is given by the tape, the tape head, and the mechanism that controls the head and changes states. More formally, a Turing machine can be defined as follows.

Definition 6.1 ► *Turing machine*

A Turing machine M *is a 6-tuple* $M = (Q, \Sigma, \Gamma, \Delta, q_S, q_A)$ *where:*

1. *Q is a finite set of internal states.*

2. *The input alphabet Σ is a finite set of symbols (not including the special symbol $\square$).*

3. *The work alphabet Γ is a finite set of symbols that includes all symbols in Σ and does not include $\square$.*

[1]All the following considerations can be easily extended to the case when M has more than one tape.

4. *The set of transition rules* Δ *is a subset of* $(Q \times (\Gamma \cup \{\square\})) \times (Q \times (\Gamma \cup \{\square\}) \times \{\mathrm{L},\mathrm{R},\mathrm{N}\})$.

5. $q_S, q_A \in Q$ *are the starting and the accepting states, respectively.*

As a particular case, a Turing machine $\mathcal{M}$ is said to be *deterministic* (see Sect. 6.1.2) if Δ is a (partial) function $\delta : Q \times (\Gamma \cup \{\square\}) \mapsto Q \times (\Gamma \cup \{\square\}) \times \{\mathrm{L},\mathrm{R},\mathrm{N}\}$. That is, $\mathcal{M}$ is deterministic if and only if, for any pair (q_i, a_k), there exists at most one transition that can fire if q_i is the current state and a_k is the currently read tape symbol. In this case, δ is said to be the *transition function*.

In the general case, a Turing machine is said to be *nondeterministic* (see Sect. 6.1.3). In this case, for the same pair (q_i, a_k) there might be several possible transitions.

A *configuration* of $\mathcal{M}$ is a description of the current overall status of the computation, including:

1. The current state.

2. The current content of the tape.

3. The current position of the tape head.

Formally, a configuration is an element of

$$Q \times (\Gamma \cup \{\square\})^* \{\#\} (\Gamma \cup \{\square\})^+$$

where $\# \notin \Gamma$. A configuration $(q_i, x_1x_2 \cdots x_{k-1}\#x_k \cdots x_n)$ means that q_i is the current state, $x_1x_2 \ldots x_{k-1}x_k \ldots x_n$ is the current tape content,[2] and the tape head is positioned on the cell containing x_k.

Given any string $\sigma = a_1a_2 \cdots a_n \in \Sigma^*$ as input to $\mathcal{M}$, the machine starts in the initial configuration $C_0 = (q_S, \#a_1a_2 \cdots a_n)$: that is, $\mathcal{M}$ is in the initial state, with the tape containing only σ, and the tape head on the first symbol.

Any computing step is represented by the application of one transition rule in Δ to the current configuration C, and brings $\mathcal{M}$ into a different configuration C'. We denote the occurrence of such a computing step as $C \vdash C'$. Thus, in general, we may define a (finite) *computation path* as a sequence $C_0, C_1, \ldots, C_m$ such that C_0 is an initial configuration and $C_i \vdash C_{i+1}$, for $i = 0, \ldots, m-1$ (notice that it is immediately possible to define also infinite computation paths).

[2]It is assumed that all cells to the left of x_1 and to the right of x_n contain the blank symbol $\square$. Observe that if the input is a finite string then, at any step of the computation, only a finite portion of the tape contains symbols different from $\square$.

We assume that a computation halts when either the current state is q_A or no transition rule is applicable to the current configuration. More formally, for any configuration $C = (q_i, x_1 x_2 \cdots x_{k-1} \# x_k \cdots x_n)$, we say that:

1. C is *accepting* if $q_i = q_A$.
2. C is *rejecting* if $q_i \neq q_A$ and there is no pair in Δ whose first element is (q_i, x_k).

Similarly, we say that a finite computation path $C_0, C_1, \ldots, C_m$ is accepting (respectively, rejecting) if C_m is an accepting (respectively, rejecting) configuration. In an accepting path, C_m must be the first configuration that contains q_A.

If an accepting (respectively, rejecting) configuration corresponds to the value TRUE (respectively, FALSE), then we have that Turing machines can be used to compute Boolean functions. It is not hard to extend the definition of Turing machines so that they can express functions returning outputs of arbitrary types. For example, when a Turing machine accepts, the result of the computation may be defined as the content of the tape between two occurrences of a special output symbol in Σ not used for anything else (for example, the symbol \$).

6.1.2 Deterministic Turing machines

As stated above, we say that a Turing machine $\mathcal{M}$ is deterministic if and only if for any possible configuration there is at most one applicable computing step.

We say that a string $\sigma \in \Sigma^*$ is *accepted* by $\mathcal{M}$ if it leads the Turing machine to halt in state q_A. That is, σ is accepted by $\mathcal{M}$ if there exists $m \geq 0$ such that the unique computation path $C_0, C_1, \ldots, C_m$ starting from the initial configuration $C_0 = (q_S, \#\sigma)$ is an accepting path.

Similarly, σ is *rejected* by $\mathcal{M}$ if there exists $m \geq 0$ such that the unique computation path $C_0, C_1, \ldots, C_m$ starting from the initial configuration $C_0 = (q_S, \#\sigma)$ is a rejecting path.

Definition 6.2 ► *Recognized language*

A language $L \subseteq \Sigma^*$ is recognized *by a deterministic Turing machine $\mathcal{M}$ if and only if:*

1. *Each string $\sigma \in L$ is accepted by $\mathcal{M}$.*
2. *Each string $\sigma \in \overline{L}$ is rejected by $\mathcal{M}$.*

Notice that if L is recognized by M then M halts for all input strings.[3] We will refer to the language recognized by a deterministic Turing machine M as $L(M)$.

◄ Example 6.1

Consider the (deterministic) Turing machine with $Q = \{q_S, q_A\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b\}$, and the following transition function δ:

	q_S	q_A
a	(q_A, a, N)	-
b	(q_S, b, R)	-
$\square$	-	-

It is possible to see that such a machine recognizes the language $L \subseteq \{a, b\}^*$ consisting of all strings containing at least one a (see Exercise 6.2). The behavior of this machine with input *bba* is shown in Fig. 6.2.

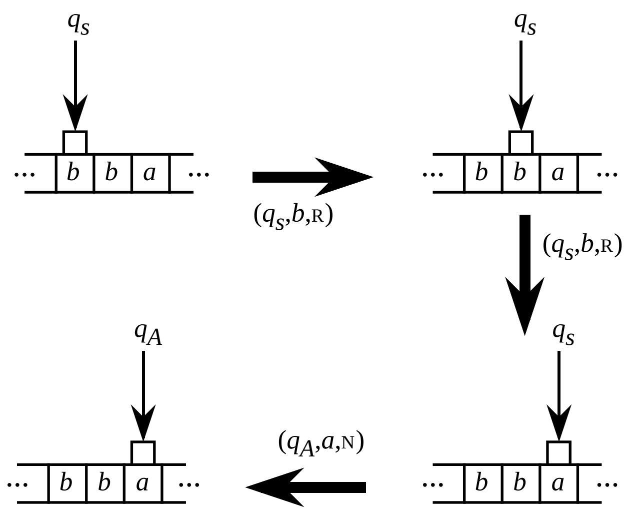

Figure 6.2
An accepting computation path of the machine of Example 6.1

◄ Example 6.2

Let us consider the (deterministic) Turing machine with $Q = \{q_S, q_1, q_A\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b\}$, and the following transition function δ:

	q_S	q_1	q_A
a	(q_1, a, R)	(q_S, a, R)	-
b	(q_S, b, R)	(q_1, b, R)	-
$\square$	$(q_A, \square, \text{N})$	-	-

It is possible to see that such a machine recognizes the language $L \subseteq \{a, b\}^*$ consisting of all strings with zero or an even number of a's (see Exercise 6.1). The behavior of this machine with input *abaab* is shown in Fig. 6.3.

[3]That is, all languages we will deal with are decidable: for the purpose of studying the complexity of decision problems, we do not have to take into consideration semi-decidable languages.

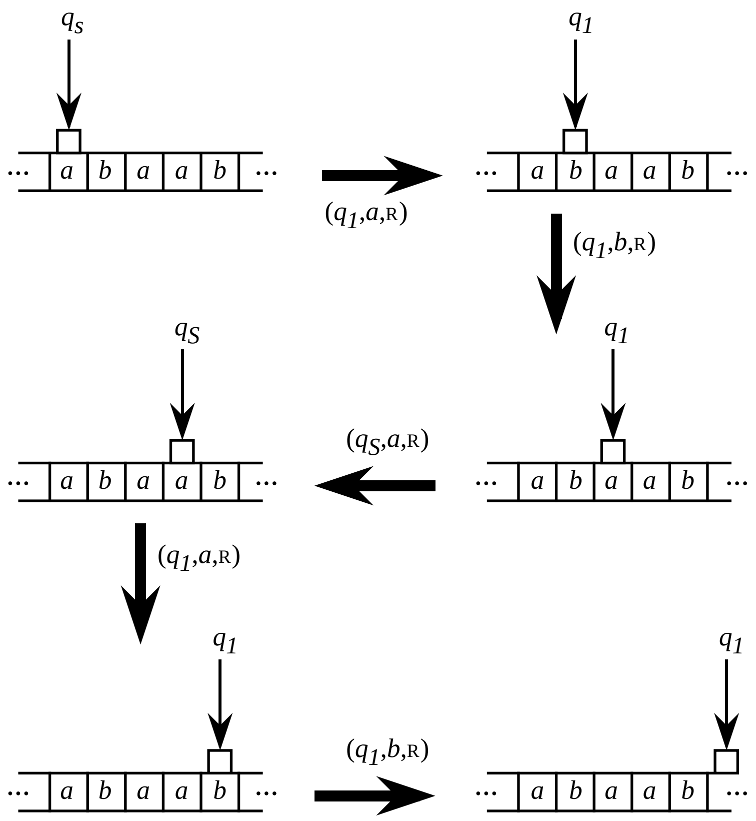

Figure 6.3
A rejecting computation path of the machine of Example 6.2

6.1.3 Nondeterministic Turing machines

Nondeterministic Turing machines correspond to the general definition of such computing devices. In general, for a nondeterministic Turing machine M, an arbitrary finite number of computing steps can be applicable to a given configuration C, i.e., there may exist a set $\{C^1, C^2, \ldots, C^r\}$ of configurations such that $C \vdash C^i$, for $i = 1, \ldots, r$. This implies that, given a nondeterministic Turing machine, there exists a tree of computation paths starting from the same initial configuration.

We say that a string $\sigma \in \Sigma^*$ is *accepted* by M if at least one such path leads the Turing machine to halt in state q_A. That is, σ is accepted by M if there exists an accepting computation path $C_0, C_1, \ldots, C_m$ starting from the initial configuration $C_0 = (q_S, \#\sigma)$.

On the other hand, σ is *rejected* by M if no computation path leads the Turing machine to halt in state q_A. That is, σ is rejected by M if all computation paths starting from the initial configuration are rejecting (notice the asymmetry between the definitions of acceptance and rejection).

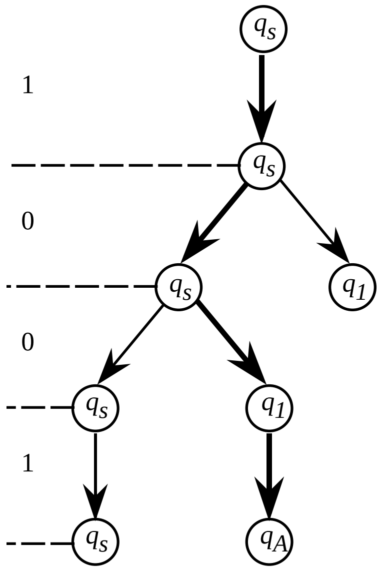

Figure 6.4
A tree of computation paths of the machine of Example 6.3

◀ Example 6.3

Let us consider the nondeterministic Turing machine with $Q = \{q_S, q_1, q_A\}$, $\Sigma = \{0,1\}$, $\Gamma = \{0,1\}$, and the following transition rules:

	q_S	q_1	q_A
0	$\{(q_S, 0, \text{R}), (q_1, 0, \text{R})\}$	-	-
1	$(q_S, 1, \text{R})$	$(q_A, 1, \text{R})$	-
□	-	-	-

The tree of computation paths relative to the input string $\sigma = 1001$ is shown in Fig. 6.4 where an accepting computation path is also highlighted.

The definition of a language recognized by a nondeterministic Turing machine is then similar to the one given in the deterministic case. As before, we will refer to the language recognized by a nondeterministic Turing machine M as $L(M)$.

6.1.4 Time and space complexity

In order to determine the execution cost of a (deterministic or nondeterministic) Turing machine on some input we may refer to two possible measures:

1. The number of computing steps performed by the machine (time complexity).

2. The amount of different tape cells visited during the computation (space complexity).

Definitions similar to those in Chap. 1 for decision problems can then be stated.

Definition 6.3 ► *Complexity bounds of Turing machines*

A language L is recognized by a deterministic Turing machine $\mathcal{M}$ in time $O(t(n))$ (respectively, space $O(s(n))$) if $L = L(\mathcal{M})$ and, for each input string $\sigma \in \Sigma^*$ with $|\sigma| = n$, $\mathcal{M}$ performs $O(t(n))$ computing steps (respectively, accesses $O(s(n))$ tape cells).[4]

The Turing machine given in Example 6.2 recognizes language L in time $O(n)$ and space $O(n)$. Actually, it is worth noting that the tape is accessed by the machine only for reading the input string: the machine never writes anything, since it does not need to store any information. This is due to the simplicity of language L, that in fact could be accepted also by simpler types of acceptors. The acceptance or recognition of more complex languages would indeed require the Turing machine to use the full power of reading and writing on its tape.

Example 6.4 ► Consider the Turing machine with $Q = \{q_S, q_1, q_2, q_3, q_A\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, \alpha, \beta\}$, and the following transition function:

	q_S	q_1	q_2	q_3	q_A
a	(q_1, α, R)	(q_1, a, R)	(q_2, a, L)	-	-
b	-	(q_2, β, L)	-	-	-
α	-	-	(q_S, α, R)	-	-
β	(q_3, β, R)	(q_1, β, R)	(q_2, β, L)	(q_3, β, R)	-
$\square$	-	-	-	$(q_A, \square, \text{N})$	-

Such a machine recognizes the language $L = \{a^n b^n \mid n \geq 1\}$ in time $O(n^2)$ and space $O(n)$. Indeed, the machine starts by marking an a (transforming it into an α) at the beginning of the input string and then goes to mark the first b (transforming it into a β); it then returns back to the first a and proceeds in the same way until all a's are marked, and the string is accepted if an equal number of b's has been marked and there are no more bs in the tape. Any other string on the alphabet $\{a, b\}$ would be rejected. The first five steps of the machine with input *aabbb* are shown in Fig. 6.5.

The space required by the machine is $O(n)$ for storing the string and the time is $O(n^2)$ because, after marking any of the a's, the machine scans $O(n)$ symbols until it finds the first unmarked b.

[4]In general, we want to consider only the amount of tape used for the computation, excluding cells necessary to hold the input string. If we want to consider the case in which the space complexity is smaller than the length of the input string we then have to use a slightly different model of Turing machine with two tapes: the first one is a read-only *input tape* that contains the input string and can be scanned only from left to right, the other one is a (read/write) *work tape* used for the computation. However, we will not need this model of computation in this book.

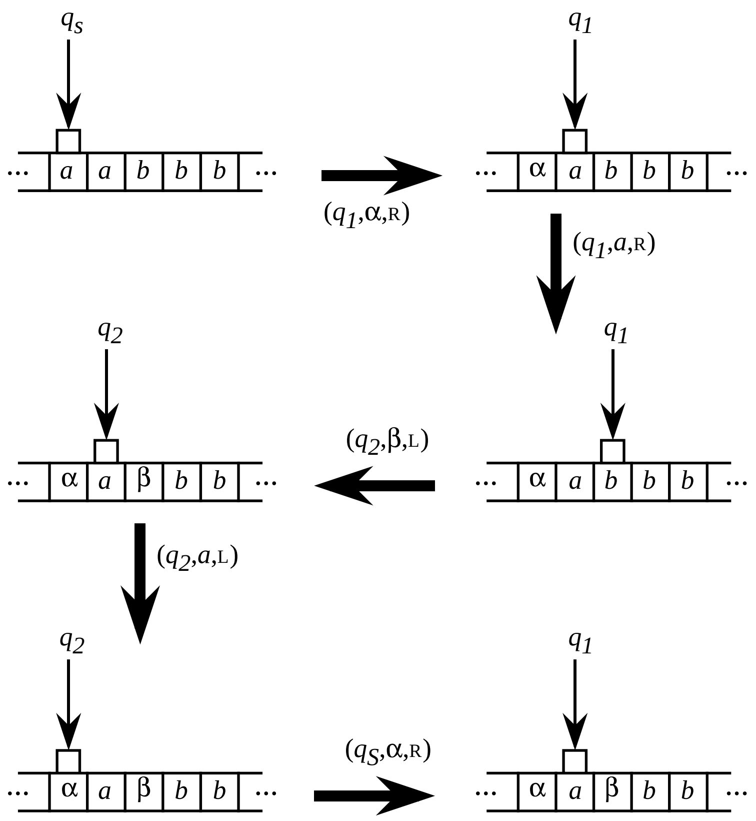

Figure 6.5
The first five steps of the machine of Example 6.4 with input *aabbb*

We will now relate the complexity of a Turing machine computation to the complexity of a Pascal program, the computation model informally described in Chap. 1. Indeed, one can show that each Pascal program can be simulated by a Turing machine. Moreover, simulating one Pascal statement does not take more steps than a polynomial in the size of the input. This means that if the Pascal program runs in polynomial time, then one can construct a Turing machine that executes the same algorithm in polynomial time. Since this is true also in the opposite situation of simulating a Turing machine by a Pascal program, we say that the Pascal program and the Turing machine models are *polynomially related*.

Thus, we can define complexity classes as P, PSPACE, and NP using Turing machines instead of Pascal programs, and using analogous definitions as in Chap. 1 (recall that with any decision problem P we can associate a language L_P).

◂ Definition 6.4
Time and space complexity classes

For any function $f : \mathbb{N} \mapsto \mathbb{R}$, *the complexity classes* $\text{TIMETM}(f(n))$ *and* $\text{SPACETM}(f(n))$ *are the sets of decision problems* P *whose corresponding language* L_P *can be recognized by a deterministic Turing machine*