TURING MACHINES

Turing machines are one of the most commonly used models for general-purpose computers. The formal definition of a TM is given in Definition 3.3, p. 168 in the text and we will go through the definition in class. The precise details of TM definition vary from textbook to textbook, but it is easy to see that all the variants are equivalent.

Note: The TM of Definition 3.3 is deterministic, that is, each configuration allows only one next computation step. Later we will consider also the nondeterministic extention of TMs.

Some conventions for the TM model we are using:

1. At the start of the computation, the input string is placed at the left end of the input tape. The input string is followed by infinitely many blank symbols ⊔.

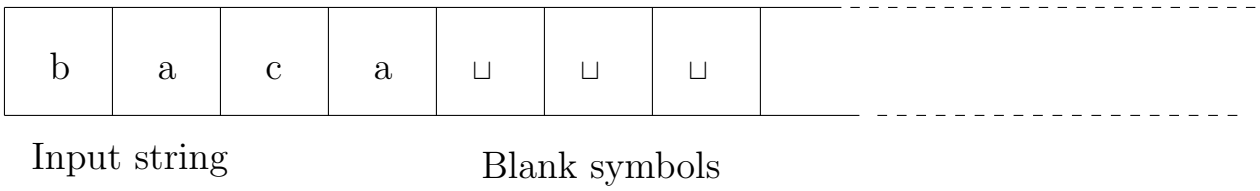

Figure 1: Tape of a Turing machine

2. There is a unique accepting state q_{accept} and a unique rejecting state q_{reject}. The TM halts when it enters the accepting or the rejecting state.

Note. When a TM is started on an input, three outcomes are possible:

1. the machine halts in accepting state;
2. the machine halts in rejecting state;
3. the computation goes on forever.

An unending computation may or may not enter a cycle (where the same configuations repeat). The latter possibility can be viewed as a cause for unsolvability of termination (and other related properties).

We use the following notation for TM state diagrams: a transition

$$\delta(q, a) = (p, b, R)$$

is denoted by an edge as in Figure 2.

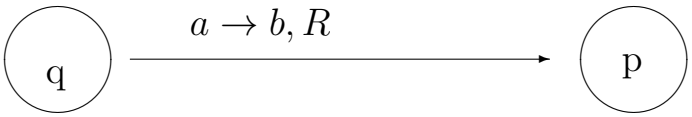

Figure 2: A transition in a TM state diagram

Example. Let $\Sigma = \{a, b\}$ and consider the following TM. What does this TM do if the

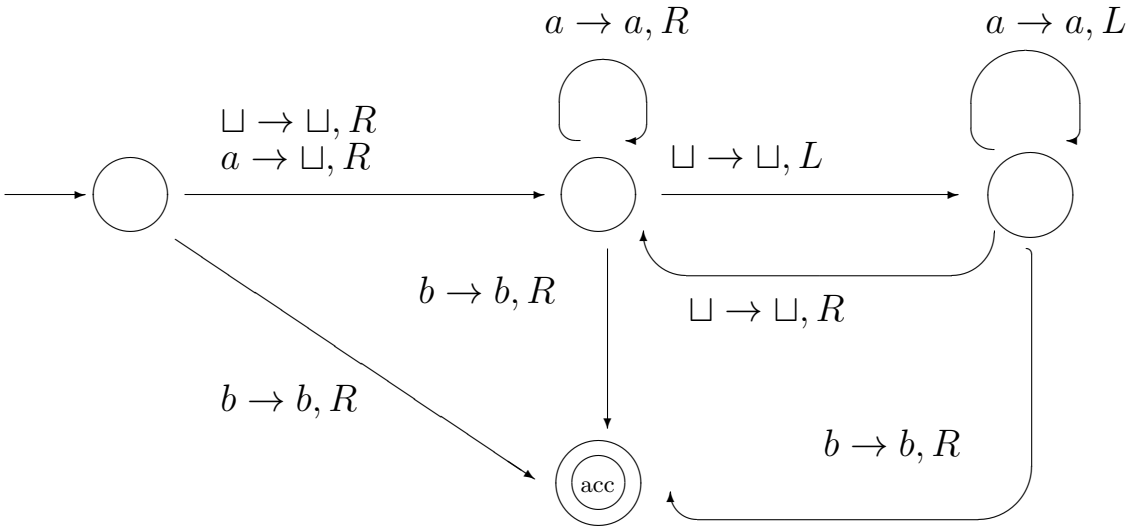

Figure 3: A Turing machine state diagram.

input string consists only of symbols a?

Example. Construct a TM that decides the language

$$L = \{a^i b^i \mid i \geq 0\}$$

Note: To simplify the state diagrams, we often do not include transitions leading to the reject state. The convention is that if $\delta(q, \gamma)$ is not defined (not included in the figure) for some state q and tape symbol γ , then $\delta(q, \gamma)$ should be interpreted as the reject state.

For a more detailed treatment of the following definitions see chapter 3 in the textbook.

A TM M **recognizes** a language L if

1. On all inputs belonging to L, the machine M enters the accepting state (after finitely many computation steps).

2. On inputs not in L, the machine either enters the rejecting state *or* does not halt.

A TM M **decides** a language L if

1. On inputs belonging to L, the machine M enters the accepting state.

2. On inputs not in L, the machine enters the rejecting state.

The latter type of machine is called a *decider.* The computation of a decider halts on all inputs. The requirement of deciding a language is stronger than recognizing the language and, as we will see, there are Turing recognizable languages that cannot be decided by any TM.

The textbook contains several more examples of TM's, see the end of section 3.1. We will go through some of the constructions in class. As we see already from these simple examples, "programming" a TM can be a quite cumbersome task. Often it is sufficient to give a more high-level description of a TM implementation.

Example. We consider a high-level description of a TM implementation for a *palindrome decider.*

A palindrome is a string that is the same as it's mirror-image. Define

$$L_{\text{pal}} = \{w \in \{a, b\}^* \mid w = w^R\}$$

(Is L_{pal} regular? Is it context-free?)

We describe how a TM can decide whether an input string over $\{a, b\}$ belongs to the language L_{pal}.

The tape alphabet Γ contains an additional symbol $*$.

The TM "remembers" (in the state) the first input symbol (which is a or b), changes it to $*$ and travels to the end of the input to compare it to the last symbol.

1. If the symbols coincide, the machine changes the last symbol to $*$, searches the first square from the left end of the tape that is not marked $*$ and repeats the above. This is continued until the remaining string of a's and b's is of length 0 or 1, and then the TM goes to state "accept".

2. If the compared symbols do not coincide, the TM goes to state "reject".

The above operations could be coded as a finite set of transitions. The implementation would be straight-forward but extremely tedious if we have to write all transitions in detail.

Since the TM's are a somewhat invonvenient model to encode even modest size algorithms, we may ask:

Why are TM's useful?

1. TM's have a very elegant and simple definition (when compared e.g. to some high-level programming language). This is very useful when we want to show that a given problem is non-computable (there is *no* algorithm for it) or when we want to establish

lower bounds for computational complexity (as will be discussed in the second half of the course).

2. In spite of their simplicity, TM's are surprisingly powerful and can, in principle, compute all algorithmic functions. This is called the Church-Turing thesis (it is discussed at the end of this section).

3. TM's allow a clear and precise definition of notions like

   - **time** used by the computation;
   - **space** (memory) used by the computation.

   We will come back to this when discussing complexity issues.

4. TM's are a very *robust model:* the power of the model remains unchanged under all reasonable extensions.

Extensions of TM's

A k-tape TM is defined similarly as an ordinary TM except

- there are k independent tapes;
- there are k read-write heads, one for each tape.

The transitions of a k-tape TM are defined as a function

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$$

Example. We construct a 2-tape TM (as a TM state-diagram) to decide the language

$$\{ba^i ba^j b \mid i, j \geq 1, i \neq j\}$$

Naturally this could be done also with a 1-tape TM, but we can make essential use of two tapes (and simplify the state-diagram) by first copying the starting b and first sequence of

a's onto the second tape, and then simply comparing the two sequences of a's one symbol at a time.

The state-diagram will be presented in class.

Theorem. Every language recognized (respectively, decided) by a multitape TM can be recognized (respectively, decided) by a single tape TM (see section 3.2).

Other TM variants include:

Multihead TMs:

A k-head TM is defined as an ordinary TM except that it has k independent read-write heads operating on the same tape.

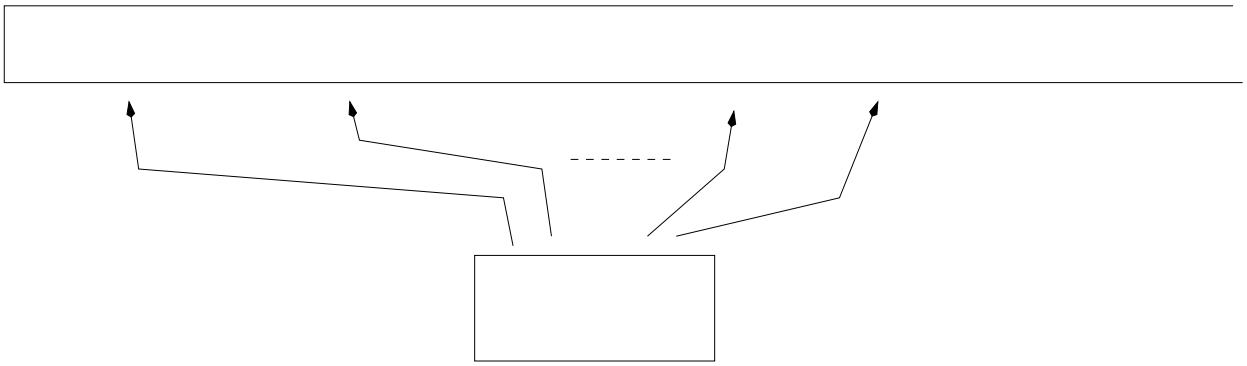

Figure 4: A multihead, single-tape TM

TMs with doubly infinite tape:

This is the same as the definition of an ordinary TM with the change that the tape is potentially infinite in both directions. The computation begins at the left end of the input string. The tape contains an infinite number of squares, initially containing the blank symbol, both towards the left and towards the right from the input, see Figure 5.

It is left as an exercise to show that these models are equivalent to the original TM definition, that is, the models recognize/decide the same family of languages.

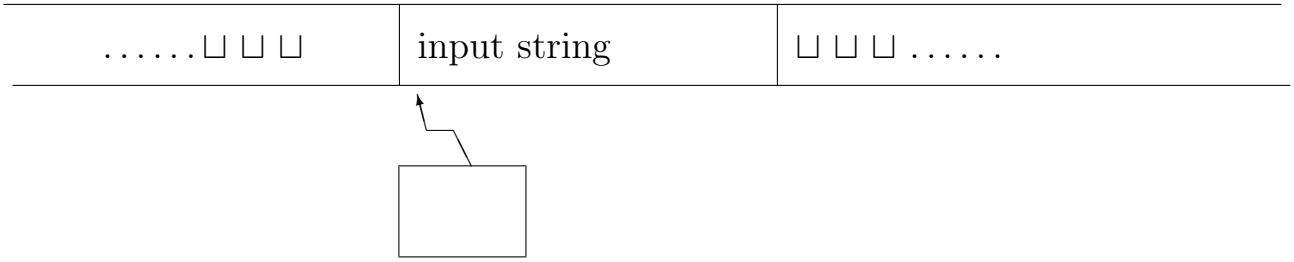

Figure 5: A TM with input tape that is infinite in both directions

Nondeterministic Turing machines

A nondeterministic computation can make "guesses" and has he magical ability to always make the correct guess. While a deterministic algorithm finds a solution to a computational problem, we can think that a nondeterministic algorithm is given the solution "for free" and the nondeterministic algorithm just needs to verify that the solution is correct.

The transition function of a nondeterministic Turing machine (NTM) associates to each pair

$$(q, \gamma) \in Q \times \Gamma$$

a subset of $Q \times \Gamma \times \{L, R\}$.

Elements belonging to $\delta(q, \gamma)$ represent the different transitions the NTM can make when its state is q and the read-write head is scanning tape symbol γ. The transitions are represented as a function

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

An NTM accepts input string w if when started with w on the tape (as the non-blank contents of the tape), it has *some choice of transitions* that lead to the accepting state q_{accept}.

Note: Even if an NTM accepts w it is possible that some computations on w may halt in a rejecting state or go on forever. It is only required that <u>at least one</u> computation halts and accepts. (This is exactly the same idea that was used when considering acceptance by nondeterministic finite automata.)

The computation of an NTM on given input w can be viewed as a tree:

- the root of the tree is labeled by the start configuration corresponding to w;
- if a node is labeled by configuration C, then it has children labeled by all the (finitely many) configurations that can be reached from C in a single computation step.

The NTM accepts input w if the above tree has at least one path (from the root to a leaf) ending in an accepting configuration.

Note: Even if the NTM accepts w, some paths of the computation tree may be infinite.

Example. Consider the language

$$L = \{ba^{i_1}ba^{i_2}b\cdots ba^{i_n}b \mid n \geq 2, \text{ and } i_j = i_{j+1} \text{ for some } j, 1 \leq j \leq n-1\}$$

To accept a string:

- a deterministic TM checks all the "neighboring pairs" of substrings $ba^{i_j}b$.
- an NTM can nondeterministically guess one pair and check this pair.
  What happens if the NTM chooses a wrong pair (where the substrings of a's are of different length)?

Example. Give a high-level description of a 2-tape nondeterministic TM that decides the language

$$L = \{0^{n^2} \mid n \geq 1\}$$

(Will be done in class or as an assignment.)

Theorem. Every language recognized by an NTM can be recognized also by a deterministic TM (see section 3.2 in the textbook)

Note: In the proof we use a deterministic multitape TM to simulate an arbitrary NTM. This makes the simulation easier and we do not lose generality because we have shown that an arbitrary multitape TM has an equivalent TM with only one tape.

The family of languages recognized by TM's is called the family of *recursively enumerable languages.* The reason for this terminology is explained at the end of section 3.2 in the text.

Church-Turing thesis and universality

Informally, an algorithm can be defined as a finite set of instructions for carrying out some task. Typically we require also that the instructions are *terminating,* that is, for each input the result is obtained after a finite number of applications of the instructions.

Up to now we have used TM's as recognizers: they give only an answer: "yes", "no", or the computation does not halt. A deterministic recognizer TM computes a function that associates to any input one of the values "yes", "no" or "undefined".

TM's can be used to carry out tasks where the algorithm has to produce more general type output: the output produced by the TM algorithm is the string given on the tape when the machine halts. In this way a TM can be viewed to compute a (partial) function from strings to strings.

- The *Church-Turing thesis* states that any function on strings that can be computed by an informal algorithm can also be computed by a Turing-machine.

In the above we assume that the inputs and outputs are encoded as finite strings over a finite alphabet. This can be easily done for all problems where the inputs/outputs are finitely specified (that is, have some kind of finite representation). All algorithmic problems have this property, at least in principle.

The Church-Turing thesis is often interpreted to mean that Turing-machines are a "universal" model of computation, that is, TM's could simulate any computation. In this context we have to be careful to remember that universality refers only to functions on strings, i.e.,

the algorithmic problem receives all inputs at the begin of the computation and there are no limitations on how the computation is performed and no need of feedback from the outside world during the computation.

When talking about universality we should be careful to separate the following three claims:

(U1) There exists a Turing machine M_1 that can simulate the computation of any other Turing machine M' (when M' is given as input for M_1).

(U2) There exists a Turing machine M_2 that can compute any algorithmically computable function on strings (that is, any partial function on strings that is computed by an informal algorithm).

(U3) There exists a Turing machine M_3 that can simulate any computation that is executable on a finitely specified computer.

The existence of a universal Turing machine M_1 that satisfies the claim (U1) will be proven in the next section.

The claim (U2) is the Church-Turing thesis. It is generally believed to be true but it cannot, even in principle, be proven to be correct because the notion of "informal algorithm" is not formally defined. The claim (U2) has been proven to hold if we replace "informal algorithm" by any of a large number of other formal models of computation, or by any existing programming language.

Many authors state the Church-Turing thesis in a form resembling (U3), examples are given in [2]. However, the claim (U3) is *incorrect.* Much more generally, it has been established in [1] that there can exist no finite model of computation that would satisfy (U3).

References

[1] N. Nagy and S.G. Akl, Time indeterminacy, non-universality in computation and the demise of the Church-Turing thesis. Queen's School of Computing Technical Report No. 2011–580.

[2] S.G. Akl, Some quotes of interest. Queen's School of Computing Technical Report No. 2006–511.