TDDD14/TDDD85 Lecture 14: Turing Machines

Victor Lagerkvist (based on slides by Christer Bäckström and Gustav Nordh)

In the last theme of the course we will introduce a model of computation, which at a first glance may only seems like a minor upgrade to pushdown automata, but which turns out to be an incredible powerful model of universal computation, *Turing machines*.

Background and Intuition

At this stage we have seen several classes of languages which we have defined either on the "grammatical" side or on the "automaton" side. While both regular and context-free languages have their own, important applications, it is clear that both finite automata and pushdown automata have rather limited expressive power. For example, the language $\{0^n1^n2^n \mid n \geq 0\}$ is not context-free, and if we try to implement a pushdown automaton for this language then the problem is that the stack is not flexible enough to keep track of both the number of zeroes, the number of ones, and the number of twos.

Intuitively, we should then consider a less restrictive form of memory. Should we add yet another stack, a random-access memory, a queue, or something completely different? Amazingly, it turns that essentially every generalisation of a PDA which adds some form of (unbounded) memory results in a computational device of the same expressive power. Hence, although we in principle have several choices, we in this lecture settle for a (conceptually) simple machine which uses a *tape*. The resulting machine is called a *Turing machine* and the tape is used as a read/write memory where one is allowed to read a single "cell" (containing a symbol from the alphabet) and transition to the left or right on the tape, in a sequential manner. Importantly, the Turing machine can both read and write on this tape, and it can be stretched arbitrary long so that it does not run out of memory in the middle of a computation. Although much more powerful than finite automata and pushdown automata, many definitions and concepts carry over, and we define (1) acceptance and rejectance of strings, (2) the language recognised by a Turing machine, (3) the class of *Turing-recognisable* languages, and (4) the important subclass of *decidable* languages. Last, we introduce and discuss the *Church-Turing thesis* and see how deterministic Turing machines can simulate nondeterministic Turing machines.

Named after the British mathematician Alan Turing.

The Turing Machine

We now properly define the most powerful model of computation that we will investigate in this course. Intuitively, a Turing machine may be viewed as a DFA which can both read and write on the input string, and which in addition is allowed to increase the length of this string if it needs additional memory. In addition the Turing machine satisfies the following specification.

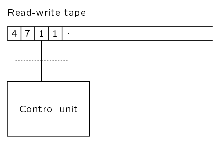

Figure 1: A visualisation of a Turing machine.
Those of you not familiar with magnetic tapes can simply view a magnetic tape as a doubly-linked list where each node, corresponding to a cell, knows the cell to the left and to the right, but nothing else. Hence, just as in a doubly linked list, we can move sequentially both to the left and to the right, but do not have "random access" to the memory.

1. The input string is visualized as a *tape* consisting of cells.
2. Cells on the tape either contains a symbol from the alphabet or a special *blank* symbol, which we denote by B.
3. The Turing machine has a "head" which can read, write, move left, and move right, on the tape, but does this in a sequential manner, just as a DFA.
4. The Turing machine cannot move any further left once it has reached the leftmost position of the tape.
5. However, the tape is effectively *unbounded* in size, meaning that the Turing machine cannot reach the (right) end of it. Conceptually, the tape at a single moment in time is always finite, but if the Turing machine reaches the rightmost cell then we "stretch" the tape by adding additional blank cells.

The formal definition of a Turing machine is then surprisingly undramatic and closely follows the various types of automata defined earlier.

Definition 1. *A* Turing machine *is a 7-tuple* $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ *where:*

- *Q is the finite set of states,*
- *Σ is the finite input alphabet not containing the blank symbol B,*
- *Γ is the finite tape alphabet where $B \in \Gamma$ and $\Sigma \subseteq \Gamma$,*
- *$\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function, where L and R stands for "left" and "right", respectively,*
- *$q_0 \in Q$ is the start state,*
- *$q_{\text{accept}} \in Q$ is the accept state, and*
- *$q_{\text{reject}} \in Q$ is the reject state.*

Intuitively, a Turing machine then operates as follows.

1. The “input string” is written to the “left” of the tape.
2. The Turing machine moves its head to the first cell of the tape and begins in its initial state q_0.
3. The Turing machine reads a single symbol at the current position of its head, writes a symbol at the cell, transitions to the state specified by the transition function, and moves its head left or right.
4. The machine accepts (immediately) if it reaches the accept state, and rejects (immediately) if it reaches the reject state, and in both cases the machine is said to *halt*.
5. If the machine does not reach an accept state or the reject state then it continues forever and is said to *loop*.

The operational semantics of a Turing machine can be formally described by *configurations*, very similar to how one can describe the behaviour of a PDA by a configuration consisting of a state, the current symbol from the input string, and the stack. Here, we only sketch the details. A configuration of a Turing machine then consists of the current state, the current position of the head, and the current content of the tape. This is typically written xqy where (1) x is the content of the tape to the left of the current position, q is the current state, and y is the content of the tape to the right, including the current cell as its first element. A Turing machine then *accepts* a string if the start configuration leads to a configuration containing the accept state, and *rejects* the string if it leads to a configuration containing the reject state. Note that if a Turing machine does *not* accept a string then it either (1) rejects the string or (2) loops indefinitely. We consider a detailed example of a simple Turing machine in the appendix, but for the moment continue to define the class of languages recognised by a Turing machine.

Definition 2. *The collection of strings that a Turing machine M accepts is the language recognized by M, denoted $L(M)$.*

This immediately leads to the following definition.

Definition 3. *A language is* Turing-recognizable *if some Turing machine recognizes it*[1].

[1] Sometimes called a recursively enumerable language.

However, note that in the above definition it could be the case that we have a language L which is recognised by a Turing machine M, but that there exists $x \notin L$ where the machine M loops. That is, instead of rejecting the string the machine simply fails to provide any answer and loops. This behaviour is not always desirable, so to circumvent this problem we also make the following stronger definition which in addition requires the Turing machine to always accept or reject a string (but is not allowed to loop indefinitely).

Definition 4. *A language is* decidable *if some Turing machine recognizes it and rejects all strings that are not in the language*[2].

[2] Sometimes called a recursive language.

We conclude this section with two examples.

Example 1. *Consider a Turing machine M with $\Sigma = \{0, 1\}$ that works as follows: M accept all strings of even length and loops on all strings of odd length. Then $L(M) = \{w \in \{0, 1\}^* \mid |w|$ is even$\}$, i.e., simply the set of strings accepted by the Turing machine M, and the fact that the machine M loops on every other string does not matter. But is this language also decidable?*

Yes! For example, by the Turing machine M' which accept all strings of even length and reject all strings of odd length (we will see later that this is a very simple task for a Turing machine).

Example 2. *Let us construct a Turing machine which recognises the language $\{0^n 1^n 2^n \mid n \geq 0\}$. This language is well-known not to be context-free, so if we succeed with this then Turing machines definitely exceed the power of finite automata and pushdown automata. Consider a Turing machine which operates according to the following description.*

1. *Scan the input from left to right and make sure it is of the form $0^*1^*2^*$ (if it is not, then reject).*

2. *Return the head to the left end of the tape.*

3. *If there is no 0 on the tape, then scan right and check that there are no 1's and 2's on the tape and accept (should a 1 or 2 be on the tape, then reject).*

4. *Otherwise, cross of the first 0 and continue to the right crossing of the first 1 and the first 2 that is found (should there be no 1 or no 2 on the tape, then reject).*

5. *Go to Step 2.*

When describing a Turing machine our main interest is typically *not* to describe the precise number of states or how the transition function should be defined. All that matters is that the high-level description is unambiguous and precise enough so that we in principle could describe the states and the resulting transition function. See Lecture 28 in Kozen [3] and Chapter 3.1 in Sipser [4] for concrete and formal examples.

[3] D. C. Kozen. *Automata and Computability*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1997

[4] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013

The Church-Turing Thesis

THE TURING machine seems to be stronger than the other computational devices we have encountered thus far. But how strong is it really?

Example 3. *Assume that we want to describe a Turing machine which, given a binary number on its tape, adds* 1 *to this number and accepts. The machine could then shift the input one step to the right and proceed to the last symbol of the string. If this symbol is* 0 *then it simply replaces it by* 1*. If this number is* 1 *then it replaces it by* 0*, remembers that it has* 1 *in "carry", goes to the left, and repeats this as long as necessary.*

We do not bother with describing the exact number of states or the transition function, since it is clear that we can accomplish the procedure with a finite number of states. Even better: if we can add 1 *to a binary number, then we could certainly also compute the addition of two arbitrary binary numbers simply by repeatedly adding* 1 *to the sum. However, if we can do addition, then we can also do multiplication as repeated addition. Similarly, we could describe a Turing machine which given two binary numbers writes* 1 *if the first number is strictly smaller than the second number, and* 0 *otherwise. But if we can implement addition, multiplication, all normal arithmetical operations and relations, then we could certainly implement e.g. "for loops", and so on. We can easily continue in this fashion and describe more sophisticated constructs from programming languages and describe their implementation in term of Turing machines.*

Once we have the idea of implementing more and more complicated data structures and concepts by shuffling strings around on the tape then it becomes more and more reasonable that a Turing machine can compute anything in this manner. Naturally, if we tried to build a Turing machine following Definition 1 then it would turn out to be awfully slow in practice, but since the Turing machine is a theoretical model of computation we do not care about this deficiency. The conjecture that Turing machines can compute everything that can be computed is known as the *Church-Turing thesis*.

The "Church" part of the conjecture is named after the American mathematician Alonzo Church who discovered an equivalent notion of computability called the *lambda calculus*.

Conjecture 1. *(The Church-Turing thesis) Everything that is "computable" can be computed by a Turing machine.*

There exist several variants of the Church-Turing thesis but the above claim is good enough for our purposes. Note that the Church-Turing thesis is not a proper mathematical conjecture since we have not provided a proper definition of "computable". We will not delve deeper into the problematic nature of giving a general definition of "computation" and take a very pragmatic view: every reasonable model of computation that has been discovered thus far can be simulated by a Turing machine. In particular, no matter how we try to generalise a Turing machine (e.g., by adding more memory, or more features), the resulting machine is still not more powerful than a Turing machine.

Nondeterministic Turing Machines

As a first sanity check of the Church-Turing thesis we will investigate nondeterminism in the context of Turing machines. This generalisation will be defined in exactly the same way as we have seen before, i.e., instead of having a transition function returning a single state we allow it to return a set of possible states. In the following definition we could also have defined δ to be a relation (as for the PDA case in lecture 9), rather than a function, but the two definitions yield the same result.

Definition 5. *A* nondeterministic Turing machine *is a 7-tuple*

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$$

where:

- *Q is the finite set of states,*
- *Σ is the finite input alphabet not containing the blank symbol B,*
- *Γ is the finite tape alphabet where $B \in \Gamma$ and $\Sigma \subseteq \Gamma$,*
- *$\delta \colon Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$ is the transition function, where L and R stands for "left" and "right", respectively,*
- *$q_0 \in Q$ is the start state,*
- *$q_{\text{accept}} \in Q$ is the accept state, and*
- *$q_{\text{reject}} \in Q$ is the reject state.*

Recall that we write $\mathcal{P}$ for the powerset operation.

The notion of a configuration immediately generalises to a nondeterministic Turing machine, and we say that a nondeterministic Turing machine accepts a string if and only if there *exists* some sequence of configurations leading to a configuration containing the accept state. This is entirely analogous to how we defined acceptance in the context of NFA/DFA, and we will not provide a more formal definition since it is not needed for our purposes. The important result is then that we can "simulate" a nondeterministic Turing machine using an ordinary Turing machine. The idea is very similar to the subset construction for converting an NFA to a DFA, where each state in the DFA represented a set of possible states in the NFA.

Theorem 1. *Every nondeterministic Turing machine can be simulated by a deterministic Turing machine.*

Proof. (Sketch) Let M be a nondeterministic Turing machine. We will construct a deterministic Turing machine which recognises the same language.

1. Given an input string w our machine explores *all* possible configurations of the non-deterministic Turing machine using a complete search strategy (e.g., breadth-first search).

2. If any of these branches leads to an accepting configuration, the machine accepts w.

3. If all branches have been fully explored without finding an accepting configuration, then the machine rejects w.

□

Naturally, writing out the full details is more complicated, but the above sketch represents the intuition behind the proof. Here it is also important that the deterministic Turing machine may need significantly more steps to reach an answer than a nondeterministic Turing machine since it has to try *all* possibilities. Whether a deterministic Turing machine can simulate a nondeterministic Turing machine with a reasonably small "overhead" is the most important open question in theoretical computer science, and is known as the $\mathrm{P} = \mathrm{NP}$ question.

Summary

We introduced and exemplified Turing machines, the most powerful model of computation that we will encounter during the course. Despite this tremendous computational power we in the next lecture will see that not all languages are Turing recognisable, and see how Turing machines can be used to relate languages via *mapping reductions*.

Food for Thought

1. Give a *high-level* description of a Turing machine which recognises the language of palindromes over $\Sigma = \{a, b\}$. To make this task simpler you may assume that you have access to a stack.

2. Give a *detailed* description of a Turing machine over the input alphabet $\{0\}$ which for any input string loops (i.e., neither accepts nor rejects).

Appendix

Example 4. *We will describe a Turing machine over $\Sigma = \{0,1\}$ which (1) writes a blank symbol on the first cell and (2) writes 0 on every other cell of the input tape and returns to the leftmost position of the tape. That is, the newly written blank symbol. Naturally, this is not a particularly exciting example, but will show how the various components of a Turing machine are defined.*

- $Q = \{q_0, q_1, q_2, q_r, q_f\}$ *is the finite set of states,*
- $\Sigma = \{0,1\}$ *is the finite input alphabet,*
- $\Gamma = \{0,1,B\}$ *is the finite tape alphabet where B is the blank symbol.*
- $q_0 \in Q$ *is the start state,*
- $q_f \in Q$ *is the accept state, and*
- $q_r \in Q$ *is the reject state.*

We proceed by defininig the transition function δ.

- $\delta(q_0, 0) = (q_1, B, R)$.
- $\delta(q_0, 1) = (q_1, B, R)$.
- $\delta(q_0, B) = (q_r, B, R)$.

In state q_0 the machine *writes* a blank symbol, proceeds to the right, and jumps to state q_1. If it *reads* a blank symbol then the input tape was empty and the machine simply rejects.

- $\delta(q_1, 0) = (q_1, 0, R)$.
- $\delta(q_1, 1) = (q_1, 0, R)$.
- $\delta(q_1, B) = (q_2, B, L)$.

In state q_1 the machine *writes* a 0 and proceeds to the right as long as possible. If it *reads* a blank symbol then it has reached the end of the tape and it jumps to state q_2.

- $\delta(q_2, 0) = (q_2, 0, L)$.
- $\delta(q_2, 1) = (q_r, B, L)$.
- $\delta(q_2, B) = (q_f, B, L)$.

In state q_2 the machine proceeds to the left as long as possible. Once it has reached the blank symbol which it originally wrote on the leftmost position of the tape it accepts.

- $\delta(q_f, a) = (q_f, a, L)$ *for each* $a \in \Gamma$.
- $\delta(q_r, a) = (q_r, a, L)$ *for each* $a \in \Gamma$.

Note that quite a few entries of δ are rather uninteresting, and cannot actully occur due to the behaviour of the machine (e.g., once the machine enters state q_2 and goes to the left it is impossible to read 1). However, since δ is a total function it still needs to be defined for all possible combinations. Now consider the input string $w = 011 \in \{0,1\}^*$. We obtain the following configurations.

- $q_0 011$ *(the start configuration when there is nothing to the left).*

- $Bq_1 11$ *(the machine replaces the first symbol* 0 *with a blank symbol).*
- $B0q_1 1$ *(the machine replaces* 1 *by* 0*).*
- $B00q_1$ *(the machine replaces* 1 *by* 0 *and has now reached the end of the input string).*
- $B0q_2 0$ *(the machine moves to the left).*
- $Bq_2 00$ *(the machine moves to the left).*
- $q_2 B00$ *(the machine moves to the left, and has reached the leftmost position).*
- $q_f B00$ *(the machine has reached the leftmost position and is finished).*

The machine then accepts since q_f *is the accepting state.*

After having seen the gritty details we thankfully do not have to worry about them again, and as we have seen it is preferable to give "high-level" descriptions of Turing machines.

References

D. C. Kozen. *Automata and Computability*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 1997.

M. Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.