Finite Automata and Formal Languages

TMV026/DIT321– LP4 2012

Lecture 15
Ana Bove

May 14th 2012

Overview of today's lecture:

- Turing Machines
- Push-down Automata
- Overview of the Course

Undecidable and Intractable Problems

The theory of undecidable problems provides a guidance about what we may or may not be able to perform with a computer.

One should though distinguish between undecidable problems and *intractable problems*, that is, problems that are decidable but require a large amount of time to solve them.

In daily life, intractable problems are more common than undecidable ones.

To reason about both kind of problems we need to have a basic notion of *computation*.

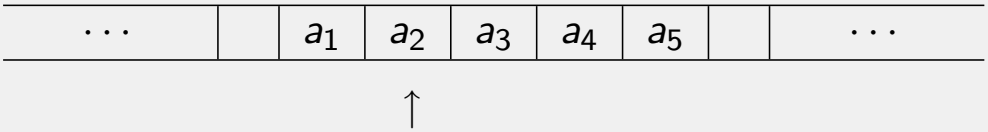
Alan Turing (1912 – 1954)

- *Alan Turing* was a mathematician, logician, cryptanalyst, and computer scientist. In the 50' he also became interested in chemistry;
- He took his Ph.D. in 1938 at Princeton with Alonzo Church;
- During his time at Princeton, he invented the concept of a computer, called a *Turing Machine* (TM);
- Turing showed that TM could perform any kind of *computation*. This result is known as *Turing completeness*;
- He also showed that his notion of *computable* was equivalent to Church's notion (that was published a bit earlier). Therefore, Church's thesis is sometimes known as *Church-Turing's thesis*;
- During the WWII he helped Britain to break the German Enigma machines.

Turing Machines (1936)

- Theoretically, a TM is just as *powerful* as any other computer! Powerful here refers only to which computations a TM is capable of doing, not to how *fast* or *efficiently* it does its job.;
- Conceptually, a TM has a finite set of states, a finite alphabet (containing a blank symbol), and a finite set of instructions;
- Physically, it has a *head* that can read, write, and move along an *infinitely long tape* (on both sides) that is divided into *cells*.

  Each cell contains a symbol of the alphabet (possibly the blank symbol):



Turing Machines: More Concretely

Let $\Box$ represents the *blank* symbol and let Σ be a non-empty alphabet of symbols such that $\{\Box, \mathrm{L}, \mathrm{R}\} \cap \Sigma = \emptyset$.
Now, we define $\Sigma' = \Sigma \cup \{\Box\}$.

The read/write head of the TM is always placed over one of the cells. We said that that particular cell is being *read*, *examined* or *scanned*.

At every moment, the TM is in a certain state $q \in Q$, where Q is a non-empty and finite set of states.

In some cases, we consider a set F of final states.

Turing Machines: Transition Functions

In one *move*, the TM will:

1. Change to a (possibly) new state;
2. Replace the symbol below the head by a (possibly) new symbol;
3. Move the head to the left (denoted by L) or to the right (denoted by R).

The behaviour of a TM is described by a (possibly partial) *transition function*

$$\delta \in Q \times \Sigma' \rightarrow Q \times \Sigma' \times \{\mathrm{L}, \mathrm{R}\}$$

δ is such that for every $q \in Q$, $a \in \Sigma'$ there is *at most* one instruction.

We have a *deterministic* TM here.

Turing Machine: Formal Definition

Definition: A *TM* is a 6-tuple $(Q, \Sigma, \delta, q_0, \Box, F)$ where:

- Q is a non-empty, finite set of states;
- Σ is a non-empty alphabet such that $\{\Box, \mathrm{L}, \mathrm{R}\} \cap \Sigma = \emptyset$;
- $\delta \in Q \times \Sigma' \to Q \times \Sigma' \times \{\mathrm{L}, \mathrm{R}\}$ is a transition function, where $\Sigma' = \Sigma \cup \{\Box\}$;
- $q_0 \in Q$ is the initial state;
- $\Box$ is the blank symbol, $\Box \notin \Sigma$;
- F is a non-empty, finite set of final or accepting states, $F \subseteq Q$.

Note: In some cases, the set F is not relevant. Then the formal definition of a TM is a 5-tuple.

How to Compute with a TM?

Before the execution starts, the tape of a TM looks as follows:

$\cdots$		a_1	a_2	$\cdots$	a_{n-1}	a_n		b_1	$\cdots$	b_m		$\cdots$
		$\uparrow$										

- The input data is placed on the tape, if necessary separated with blanks;
- There are infinitely many blank to the left and to the right of the input;
- The head is placed on the first symbol of the input;
- The TM is in a special *initial state* $q_0 \in Q$;
- The machine then proceeds according to the transition function δ.

Language Accepted by a Turing Machine

Definition: Let $M = (Q, \Sigma, \delta, q_0, \Box, F)$ be a TM. The *language accepted* by M is the set of $w \in \Sigma^*$ such that when we run M with w as input data we reach a final state.

Definition: The set of languages accepted by TM are called *recursively enumerable languages*.

Example

The following TM accepts the language $\mathcal{L} = \{ww^r \mid w \in \{0,1\}^*\}$.

Let $\Sigma = \{0, 1, X, Y\}$, $Q = \{q_0, \ldots, q_7\}$ and $F = \{q_7\}$,
Let $a \in \{0, 1\}$, $b \in \{X, Y, \Box\}$, and $c \in \{X, Y\}$.

$$\begin{array}{ll} \delta(q_0, 0) = (q_1, X, \mathrm{R}) & \delta(q_0, 1) = (q_3, Y, \mathrm{R}) \\ \delta(q_1, a) = (q_1, a, \mathrm{R}) & \delta(q_3, a) = (q_3, a, \mathrm{R}) \\ \delta(q_1, b) = (q_2, b, \mathrm{L}) & \delta(q_3, b) = (q_4, b, \mathrm{L}) \\ \delta(q_2, 0) = (q_5, X, \mathrm{L}) & \delta(q_4, 1) = (q_5, Y, \mathrm{L}) \\ \delta(q_5, a) = (q_6, a, \mathrm{L}) & \delta(q_5, c) = (q_7, c, \mathrm{R}) \\ \delta(q_6, a) = (q_6, a, \mathrm{L}) & \delta(q_6, c) = (q_0, c, \mathrm{R}) \end{array}$$

What happens with the input 0110?

And with the input 010?

Result of a Turing Machine

Definition: Let $M = (Q, \Sigma, \delta, q_0, \Box, F)$ be a TM. We say that M *halts* if for certain $q \in Q$ and $a \in \Sigma$, $\delta(q, a)$ is undefined.

Whatever is written in the tape when the TM *halts* can be considered as the *result* of the computation performed by the TM.

If we are only interested in the result of a computation, we can omit F from the formal definition of the TM.

Examples

Example: Let $\Sigma = \{0, 1\}$, $Q = \{q_0\}$ and let δ be as follows:

$$\delta(q_0, 0) = (q_0, 1, \mathrm{R})$$
$$\delta(q_0, 1) = (q_0, 0, \mathrm{R})$$

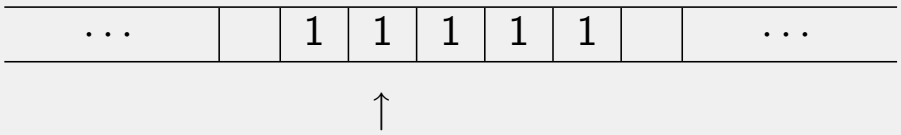
What does this TM do?

Example: The execution of a TM might never stop.
Consider the following set of instructions for Σ and Q as above.

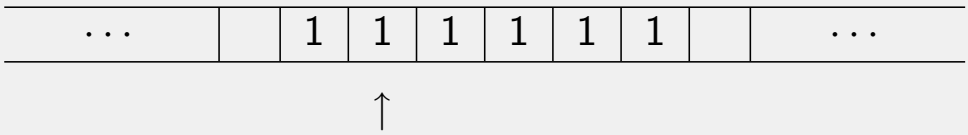
$$\delta(q_0, a) = (q_0, a, \mathrm{R}) \qquad \text{with } a \in \Sigma \cup \{\Box\}$$

Coding the Natural Numbers

Unary Coding The number 0 is represented by the empty symbol $\square$ and a number $n \neq 0$ is represented with n consecutive 1's.
The number 5 is then represented as



Kleene's Coding The natural number n is represented with $n + 1$ consecutive 1's.
The number 5 is then represented as



Turing Completeness

It is the equivalent to Church's thesis but talks about Turing machines.

Turing Completeness: A function is *mechanically computed* if and only if it can be defined as a Turing Machine.

This is not a theorem and it can never be one since there is no precise way to define what it means to be *mechanically computed*.
(The same applies to Church's thesis.)

However, it is strongly believed that both statements are true since they have not been refuted in the ca. 80 years which have passed since they were first formulated.

Variants of Turing Machines

What follows are some variants, extensions and restrictions to the notion of TM that we presented, none of them modifying the power of the TM.

- Storage in the state;
- Multiple tracks in one tape;
- Subroutines;
- Multiple tapes;
- Non-deterministic TM;
- Semi-infinite tapes.

Exercises

1. Write TM that compute the following functions over the Natural numbers:
   1. Successor and predecessor;
   2. Addition and subtraction;
   3. Multiplication and exponentiation.
2. Write TM that recognise the following languages:
   1. $\mathcal{L} = \{0^n 1^n \mid n > 0\}$;
   2. $\mathcal{L} = \{0^n 1^n 2^n \mid n > 0\}$.

Push-down Automata

Push-down automata (PDA) are essentially ϵ-NFA with the addition of a *stack* where to store information.
The stack is needed to give the automata extra "memory".

Example: To recognise the language 0^n1^n we proceed as follows:

- When reading the 0's, we push a symbol into the stack
- When reading the 1's, we pop the symbol on top of the stack
- We accept the language if when we finish reading the input the stack is empty.

The languages accepted by the PDA are exactly the CFL.
See the book, sections 6.1–6.3.

Variation of Push-down Automata

DFA + stack: This kind of PDA accepts a language that is between the RL and the CFL.
The languages accepted by these DPDA all have unambiguous grammars.
However, not all languages that have unambiguous grammars can be accepted by these DPDA.

Example: The language generated by the unambiguous grammar

$$S \to 0S0 \mid 1S1 \mid \epsilon$$

cannot be recognised by a DPDA.
See section 6.4 in the book.

2 or more stacks: A PDA with at least 2 stacks is as powerful as a TM.
Hence these PDA can recognise the recursively enumerable languages.
See section 8.5.2.

Overview of the Course

We have covered/you should know chapters 1–5 + 7 + (8):

Formal proofs: mainly proofs by induction

Regular languages: DFA, NFA, ϵ-NFA, RE
Algorithms to transform one formalism to the other
Properties of RL

Context-free languages: CFG
Properties of CFL

Turing machines: Just a bit

Formal Proofs

We have used formal proofs along the whole course to prove our results.

Mainly proofs by induction:

- By induction on the structure of the input argument;
- By induction on the length of the input string;
- By induction on the length of the derivation;
- By induction on the height of a parse tree.

Finite Automata and Regular Expressions

FA and RE can be used to model and understand a certain situation/problem.

Example: Consider the problem with the man, the wolf, the goat and the cabbage.
Also the Gilbreath's principle. There we went from NFA $\rightarrow$ DFA $\rightarrow$ RE.

They can also be used to describe (parts of) a certain language.

Example: RE are used to specify and document the lexical analyser (*lexer*) in languages (the part of the compiler reading the input and producing the different *tokens*).
The implementation performs the steps RE $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ min DFA.

Example: Using Regular Expression to Identify the Tokens

```
Tokens = Space (Token Space)*
Token  = TInt | TId | TKey | TSpec
TInt   = Digit Digit*
Digit  = '0' | '1' | '2' | '3' | '4' | '5' | '6' |
         '7' | '8' | '9'
TId    = Letter IdChar*
Letter = 'A' | ... | 'Z' | 'a' | ... | 'z'
IdChar = Letter | Digit
TKey   = 'i''f' | 'e''l''s''e' | ...
TSpec  = '+''+' | '+' | ...
Space  = (' ' | '\n' | '\t')*
```

Regular Languages

Intuitively, a language is regular when a machine needs only limited amount of memory to recognise it.

We can use the Pumping lemma for RL to show that a certain language is not regular.

There are many decision properties we can answer for RL.
Some of them are:

$$\mathcal{L} \neq \emptyset? \qquad w \in \mathcal{L}? \qquad \mathcal{L} \subseteq \mathcal{L}'? \qquad \mathcal{L} = \mathcal{L}'?$$

Context-Free Grammars

CFG play an important role in the description and design of programming languages and compilers.

CFG are used to define the syntax of most programming languages.

Parse trees reflect the structure of the word.
In a compiler, the parser takes the input into its abstract syntax tree which also reflects the structure of the word but abstracts from some concrete features.

A grammar is ambiguous if a word in the language has more than one parse tree.

LL grammars are unambiguous.
There are algorithms to decide if a grammar is LL(1).

Context-Free Languages

These languages are generated by CFG.
It is enough to provide a stack to a NFA in order to recognise these languages.

We can use the Pumping lemma for CFL to show that a certain language is not context-free.

There are only a couple of decision properties we can answer for CFL. They are:

$$\mathcal{L} \neq \emptyset? \qquad w \in \mathcal{L}?$$

However there are no algorithms to determine whether $\mathcal{L} \subseteq \mathcal{L}'$ or $\mathcal{L} = \mathcal{L}'$.

There is no algorithm either to decide if a grammar is ambiguous or a language is inherently ambiguous.

Turing Machines

Simple but powerful devices.
They can be though of as a DFA plus a tape which we can read and write.

Define the recursively enumerated languages.

It allows the study of *decidability*: what can or cannot be done by a computer (halting problem).

Computability vs *complexity* theory: we should distinguish between what can or cannot be done by a computer, and the inherent difficulty of the problem (*tractable* (polynomial)/*intractable* (NP-hard) problems).

Church-Turing Thesis

In the 1930's there has been quite a lot of work about the nature of *effectively computable (calculable) functions*:

- Recursive functions by Stephen Kleene;
- λ-calculus by Alonzo Church;
- Turing machines by Alan Turing.

The three computational processes were shown to be equivalent by Church, Kleene, (John Barkley) Rosser (1934—6) and Alan Turing (1936—7).

The *Church-Turing thesis* states that if an algorithm (a procedure that terminates) exists then, there is an equivalent Turing machine, recursively-definable function, or a definable λ-function for that algorithm.