4. Turing machines and computability

1. Turing machines
2. Turing decidability
3. Undecidability of the Halting Problem
4. Undecidable problems and reducibility

Inquiry

- What do Turing machines tell us about computing?
- How do TMs help a software engineer?
- Can transition systems compute functions on strings and natural numbers?
**Topic objective**

Define computability, showing the expressiveness and limitations of the Turing machine

---

**Subtopic objectives**

4.1 Describe the Turing-machine**

4.2a Show that a problem is decidable*

4.2b Show expressiveness of the TM model*

4.3a Describe the Halting Problem and related proof*

4.3b Show that a problem is undecidable

4.4 Explain the notion of reducibility of problems
1. Turing machines

- Can a state-transition system with infinite two-way tape solve problems a PDA cannot?
- How can arithmetic operators be computed with transition systems?
- Is there a way to enhance the TM model?
- Can a simple TM emulate a complex one?

Subtopic objective

4.1 Describe the Turing machine**
Not all decidable sets are CF

- What class of device accepts $L = \{xx | x \in \Sigma^*\}$?
  (strings that consist of the same substring repeated twice)
- $L$ is not context free, is not recognized by any PDA (provable by use of Pumping Lemma for CFLs)
- Clearly $L$ is decidable by some algorithm
- What change to a PDA would allow solving this problem?

Turing’s model of computation

- Equivalent to the notion of an algorithm
- Developed (1936) to solve a famous unsolved problem in logic
- Based on idea of a human “computer” with paper and pencil
- Helps us capture essence and limits of algorithmic computing
- Enabled invention of general-purpose computers
The Turing machine

- Augments DFA with a two-way read/write tape with infinite capacity
- *Operations*: tape head reads a symbol at current location on paper, moves left or right, writes symbol at current location
- Next action is looked up in *transition table* based on current input and “state of mind” of computer
- Machine halts when a it enters a “halting” (accept or reject) state

Example: negater TM

- The TM above reads one bit, writes its logical negation to the tape, and halts
- Alphabet $\Sigma = \{0, 1\}$
- State set $Q = \{ q_0, q_1 \}$
- Transition function $\delta =$ \{(q\(_0\), 0), (q\(_1\), 1)\}, \{(q\(_0\), 1), (q\(_1\), 0)\}\}
**Definition of TM**

- **Turing machine:**
  \[ M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}} \rangle \]
  where \( \Sigma \) and \( \Gamma \) are *input* and *tape* alphabets

- **Transition function** \( \delta \):
  \[ Q \times (\Sigma \cup \{\#\} \cup \Gamma) \rightarrow Q \times (\Sigma \cup \Gamma \cup \{\#', L, R\}) \]
  where \( L, R \) denote left or right moves

- Tape is infinite in both directions
- Tape head starts at leftmost nonblank cell
- First blank to right of a symbol in \( \Sigma \) has infinitely many blanks to its right

**Accepter and transducer TMs**

- **Accepter TMs** are like DFAs and PDAs in that according to the final state, an accepter *accepts* or *rejects* input
- Unlike DFAs and PDAs, TMs read tape rather than consuming input; hence may loop infinitely
- **Transducer TMs** compute functions \( \phi : \Sigma^* \rightarrow \Sigma^* \), where the return value (output) is the value left on the tape at the end of the computation
Example: unary incrementer

- Given input of a series of \( n \) 1’s on tape, this TM will leave \( (n + 1) \) 1’s as output
- \( \Sigma = \{ 1 \} \)
- \( Q = \{ q_0, q_{\text{accept}} \} \)
- \( \delta = \{ \langle (q_0, 1), (q_0, \text{R}) \rangle, \langle (q_0, \text{#}), (q_{\text{accept}}, 1) \rangle \} \)

Unary adder

- Reads two numbers with “+” between
- Replaces last “1” with a blank
- Example: \( 11 + 111 \rightarrow 11111 \)
4. Turing machines and computability

**Binary incrementer**

0, 1 → R
0, ‘#’ → 1

q₀

‘#’ → L

q₁

1 → 0

q₂

0 → L

q₃

1. Scan to right, then step left once
2. While input is 1
   write 0 and move left
3. Read 0 or ‘\0’
4. Write 1 and accept

---

**Decision and transduction**

- A *decision problem* requires *yes/no* responses to input
- Hence it is also known as a *language-recognition* problem
- A *transduction* computation produces a *string* as output
- Hence its solution is the computation of a function \( \Sigma^* \rightarrow \Sigma^* \) (equivalent to \( \mathbb{N} \rightarrow \mathbb{N} \))
- We speak of *decidable languages* (problems) or *computable functions*
TM computations and configurations

- A **TM configuration** is a snapshot of the TM’s components at an instant in time
- A **TM computation** is a sequence of configurations observable under $\delta$
- A computation that enters $q_{\text{acc}}$ or $q_{\text{rej}}$ is said to halt

TM configurations

- **Configuration** $(q, t_{\text{left}}, t_{\text{head}}, t_{\text{right}})$: state; tape contents to left of head; symbol at head; contents to right of head
- The assertion that configuration $C = (q, x, a, y)$ of TM $M$ yields configuration $C' = (q', w, b, z)$ in one step is written $C \Rightarrow_M C'$
- Intuitively, $C \Rightarrow_M C'$ means that from configuration $C$, application of $\delta(q, t_{\text{head}})$ yields configuration $C'$
Transitions between configurations

- When \( C = (q, x, a, y) \) and \( C' = (q', w, b, z) \),
  \( C \Rightarrow_M C' \), iff \( \delta(q, t_{head}) = \)
  - \((q', b)\) and \( w = x, z = y \), or
  - \((q', L)\) and \( w = \text{left}(x, \text{length}(x) - 1) \),
    \( b = \text{right}(x, 1), z = ay \), or
  - \((q', R)\) and \( w = xa, b = \text{left}(y, 1) \),
    \( z = \text{right}(y, \text{length}(y) - 1) \)

- Intuitively, under \( \delta \) either \( b \) replaces \( a \), or else the tape head moves left or right.

TM computations

- A *computation* is a sequence of configurations \( C_0, C_1, \ldots, C_n \), where \( (\forall i < n) \ C_i \Rightarrow_M C_{i+1} \)
- By convention that input is initial contents of tape and output is final contents, a TM computes a function \( \phi : \Sigma^* \rightarrow \Sigma^* \)
- Tape is erased between computations; hence \( C_0 = (q_0, \lambda, x[1], \text{right}(x, \text{length}(x) - 1)) \) where \( x \) is input
- A TM that eventually halts on all inputs computes a *total* function; otherwise \( \phi \) is *partial*.
TMs compute partial functions

- Let $\phi_M(x)$ be the string left on the tape, after $M$ halts, on input $x$
- Consider TM $M$ at right:
  $$\phi_M(x) = \begin{cases} x & \text{if } x[1] = 1 \\ \uparrow & \text{otherwise} \end{cases}$$
- $M$ computes a partial but not total function; i.e., $\phi_M(x)$ is undefined for some $x$
- A TM that always halts computes a total function

TM simulation with JFLAP

- JFLAP is Java software that simulates automata
- It has a graphical user interface
- You may test your designs with it
- http://www.jflap.org/
2. Turing decidability

- What languages (problems) are decidable (solvable) with some TM?
- What functions are TM computable?
- What are the limits of the TM’s expressiveness?

Subtopic objectives

4.2a Show that a problem is decidable*

4.2b Show expressiveness of the TM model*
Computable functions

- Given TM $M$, $x \in \Sigma^*$, let $\phi_M(x)$ denote the tape contents of $M$ after a halting computation with input $x$.
- Given function $f : \Sigma^* \to \Sigma^*$, $M$ is said to compute $f$ iff for all $x$ in $\text{Dom}(f)$, $M$ eventually halts on $x$ and outputs $f(x)$, while for all other $x$, $M$ hangs.
- Function $f$ is called Turing computable (recursive) if there is a TM $M$ that computes $f$.

Decidable languages

- Language or problem $L$ is said to be decidable (recursive) if some TM exists that
  - accepts all strings in $L$ (i.e., halts in state $q_{\text{acc}}$) and
  - rejects all other strings (i.e., halts in state $q_{\text{rej}}$).
Some TM-computable functions

- **Theorem**: constant, successor, predecessor, projection functions are TM computable
- **Constant** function \( f(x) = k \) is computed by a TM that ignores its input and writes \( k \) 1s
- **Successor (predecessor)**: TM steps to right-most 1, writes an additional 1 (deletes last 1)
- **Projection**: \( f_k(x_1, \ldots, x_n) = x_k \) is computed by a TM that erases all values before and after the \( k^{th} \)
Regular languages are decidable

*Theorem:* If $L$ is regular, then $L$ is recursive

*Proof* (by construction, defining TM $M$):
1. Let $A$ be a DFA that accepts $L$
2. $M$ has $A$’s states and transition function, modified to change labels for every $a \in \Sigma$ to $(a, \text{R})$, scanning the tape to the right
3. Add states, $q_{\text{accept}}$ and $q_{\text{reject}}$
4. For each accepting state of $A$, add to $M$ an edge (‘#’, ‘#’) that goes from that state to $q_{\text{accept}}$; for each other add edge (‘#’, ‘#’) to $q_{\text{reject}}$
5. Thus $M$ accepts exactly the strings in $L$

**Theorem:** CFLs are decidable

*Proof*:
1. Let PDA $A$ be an acceptor of $L$.
2. Construct 2-tape TM $M$ from $A$ as follows:
   - $Q_M = Q_A$, $\delta_M = \delta_A$ except as follows
   - For every stack-popping operation of $A$, let $M$ travel to rightmost cell of tape 2, reading it and replacing it with ‘#’;
   - simulate push by appending symbol at rightmost cell
3. Then $M$ simulates $A$ and accepts $L$
Algorithmically decidable properties of PDAs, CFGs

- Given PDA $A$, whether $A$ will accept a string (i.e., $x \in L(A)$) is decidable
- Given CFG $G$, there exist algorithms to decide the following about its language:
  - $(x \in L(G))$
  - $L(G) = \emptyset$
  - $L(G)$ is finite

A binary encoding for any TM

- For TM $M = (Q, \{0, 1\}, \Gamma, \delta, q_0, F)$, assign integers to states $q$, symbols $X$, directions $D$
- Encode transition rule $\delta(q_0, X_j) = (q_k, X_l, D_m)$ as $c_n = 0_i10_j10_k10_l10_m$
- Zeroes express integers in unary; ones are delimiters
- Encode the set of all transition rules as $c_11 \ c_211 \ c_311\ldots$, using “11” as delimiter
- Now each TM has a binary representation; hence can be denoted by a natural number
Universal Turing machines

• Suppose TM $U$ takes as inputs a pair:
  - $(M),$ encoding of $M$;
  - $x \in \Sigma^*$, an input to TM $M$
• Then for all inputs $(M, x)$, $U$ outputs what $M$ would output on input $x$
• Then $U$ is called a universal Turing machine
• Example: Any general-purpose stored-program computer is equivalent to a universal TM
• Theorem: A universal TM exists

Multitape TMs

• For $k$-tape TM, use
  $$\delta : Q \times (\Sigma \cup \Gamma)^k \rightarrow Q \times (\Sigma \cup \Gamma \cup \{L, R\})^k$$
• Theorem: For any $k$-tape TM $M_k$, an equivalent single-tape TM $M$ exists
• Proof sketch: construct $M$ to simulate $M_k$
  – $M$’s tape contains contents of all $M_k$’s tapes, delimited by blanks (‘#’)
  – $k$ head locations on $M_k$ are denoted on $M$ by special symbols using an alphabet $k$ times as large as $\Gamma$ (e.g., $a, b, \ldots, a, b, \ldots$)
Linear bounded automata

- LBA: TM restricted to tape space occupied by input or proportional to size of input
- LBAs accept a strict superset of CFLs and a strict subset of decidable language accepted by TMs
- LBA-accepted languages are generated by unrestricted grammars

Randomized TMs

- *Tapes*: input, random, scratch
- *Random tape* is pre-written with infinitely many random bits; alternatively a random bit is written whenever a new cell is accessed
- *Example use*: to execute Quicksort, choose pivot location randomly
**Is an infinite-tape model realistic?**

- **Arguments:**
  - No computer system can simulate a TM because all actual systems are finite
  - Likewise, no system can operate on unrestricted data values in $\Sigma^*$ or $\mathbb{N}$, because storage is finite in the real world
- **Counter arguments:**
  - Systems are arbitrarily expandable in storage, may store and access data remotely
  - No TM computation ever accesses infinite data

---

**TM subroutines**

- TMs may implement *subroutines* by grouping states and transitions with a destination state to “return” to at the end of the subroutine execution
- Calls from different states may be implemented by copying, having a different destination/return state
Nondeterministic TMs

- An NTM is a TM, except that for each state-symbol pair \((q, x)\), \(\delta(q, x)\) is a set of values \((q', y, \text{direction})\)
- **Theorem:** \(\{L(M) \mid M \text{ is an NTM}\} = \{L(M) \mid M \text{ is a DTM}\}\)
- **Proof:** Simulate an NTM deterministically on a multi-tape DTM by copying all configurations as \(\delta\) generates them, to right end of one tape; process all those configurations with \(\delta\), halting if a configuration is in halt state

Restricted TMs

**Theorem:** The following are equally expressive as DTMs with infinite tape:

- **TMs with semi-infinite tape**, i.e., tape that is infinite in one direction, and that don’t write blanks
- **2-stack machines** that read input from tape left to right and perform stack operations
- **2-counter machines** that replace stack or tape with two numeric registers
3. Undecidability of the Halting Problem

- What can we know about a TM’s behavior by looking at its structure?
- Can software be designed that tells what a program does in all cases?
- Are there problems that have no algorithmic solutions?

Subtopic outcome

4.3a  Describe the Halting Problem and related proof*
4.3b  Show that a problem is undecidable
Limits of TMs

- Some problems are not decidable, e.g., whether a TM halts on a given input
- **Goal**: to find the limits of algorithmic computation, i.e., the computable functions
- **Highest result**: Nothing interesting about the language accepted by an arbitrary TM or program can be determined by looking at it

The Halting Problem

- Consider the decision problem or language $HALT$, consisting of the set of pairs $(M,x)$ s.t. the TM with description $M$ halts on input $x$
- Is there a TM that decides $HALT$?
**Theorem:** HALT is undecidable

**Proof:**
1. Suppose HALT is decidable.
2. Then construct TM $S$ from HALT-decider $H$, where $S$ makes a copy of its input $M$, feeds $M$ and $M$ to $H$, loops forever if $\phi_H(M,M)\downarrow$, halts if $\phi_H(M,M)\uparrow$.
3. Consider whether $S$ halts on input $S$. If $\phi_S(S)\downarrow$ then $\phi_S(S)\uparrow$ and conversely -- a contradiction.
4. Since $S$ is clearly constructible except for component $H$, therefore $H$ cannot be constructed.
5. Hence HALT is undecidable.

---

**Cantor’s and Turing’s proofs**

- Consider an infinite table listing inputs $x$ across the top and TMs $M$ down the side.
- The entries tell whether $M$ halts on $x$.
- Consider TM $S$ that is designed to halt if its input is a TM that hangs on its own description and to hang if it halts.
- The behavior of $S$ is the bitwise negation of the diagonal of our table; but by definition no entry in the table can have this behavior (see Cantor proof).
- Hence no TM $M$ can detect halting behavior.
4. Undecidable problems and reducibility

- Is the problem of whether a TM ever halts on any input decidable?
- Is the output of a TM on no input computable from its representation?
- Are there any behaviors of TMs that can be determined with certainty by examining the TM’s representation?

Subtopic objective

4.4 Explain the notion of reducibility of problems
Reducibility

• Problem $B$ is reducible to problem $A$ iff a solution to $A$ enables a solution to $B$
• Intuitively, $B$ is at least as hard as $A$
• Example: multiplication is reducible to addition
• Hence we can build a multiplying machine out of an adding machine

Uses for reductions

• Suppose that, with a machine that solves problem $A$, plus a machine or module $C$, we can build a machine that solves problem $B$
• What if we know that $B$, reducible to $A$, is unsolvable? Then we know that $A$ is unsolvable
• Example: By showing that the Halting Problem is reducible to problem $P$, we can show that $P$ is undecidable too
Hardwiring a TM

- Any TM $M$ can be hardwired to a given input, $x$, i.e., converted to $M_x$ that ignores its input and outputs $\phi_M(x)$

- Let $HW$ be a TM that performs this conversion, by generating a description of a TM $M_x$ that discards its input and replaces it with $x$

Halt-on-blank

- Problem $HB$: Decide whether input is in $\{ M \mid M$ is a TM that halts on input $\lambda$ (blank) $\}$
- Using $HW$, $HALT(M, x)$ is reducible to $HB$:

- **Theorem:** Problem $HB$ is undecidable
- **Proof:** $HB$ is reducible to $HALT$ because $H$ can be constructed from $Halt-on-Blank$; but $H$ cannot exist, so $HB$ solver cannot exist
Halting on all inputs

- Let $\text{HALT-ALWAYS}(M)$ be the assertion that TM $M$ halts on all inputs
- **Theorem:** $\text{HALT-ALWAYS}$ is undecidable
- **Proof:** The following TM decides the undecidable halting problem for blank input ($HB$)

![Diagram]

- …where the preprocessor produces description of TM $M_\lambda$ that simulates $M$ running with blank input

Undecidability of TM behavior

- Not only is the problem of deciding whether a given program halts uncomputable…
- …but also no programs exist to decide any useful (nontrivial) property of programs
- By “nontrivial” we mean any behavioral property that holds for some but not all programs
- **Examples:**
  - Is $L(M)$ infinite? Regular? CF?
  - Is $x$ in $L(M)$?
  - Is $L(M_1)$ a subset of $L(M_2)$
Rice’s Theorem

Thm: For any nontrivial class \( C \) of r.e. languages (e.g., decidable sets), and any TM \( M \), \((L(M) \in C)\) is undecidable (all interesting behavioral properties of TMs are undecidable).

Proof:
1. Suppose \((L \in C)\) were decidable.
2. Then some program \( L\text{-in-}C \) decides this problem.

Rice theorem proof, ii

3. Note that TM generator \( SG_L \) is constructible, generating from TM \( Q \) and \( L \)-acceptor \( M_L \) a TM \( M_{Q,L} \), that loops forever if \( Q \) hangs, otherwise tells whether its input is in \( L \).
4. Hence \( L(M_{Q,L}) = L \) iff \( Q \) halts.
Rice theorem proof, iii

5. Then construct from $S_{GL}$ and $L$-$in$-$C$ the following TM:

6. But this TM decides the *Halt-always* problem, which is undecidable.

7. Hence $L$-$in$-$C$ is undecidable.

Undecidable properties of PDAs and CFGs

• Given PDAs $A_1$ and $A_2$, $(L(A_1) = L(A_2))$ is algorithmically undecidable
• Given CFG $G$, these are undecidable:
  - the complement of $L(G)$ is CF
  - $G$ is ambiguous
  - $L(G)$ is regular; $L(G) \supseteq L$ given RL $L$
• Given CFGs $G_1$ and $G_2$, it is undecidable whether $(L(G_1) = \subseteq L(G_2))$;
  - $L(G_1) \cap L(G_2)$ is CF) is undecidable
Undecidable problems about context-free grammars

Theorem: The following are undecidable for CFGs $G, G_1, G_2$, regular expression $R$:

- $L(G) = \Sigma^*$
- $L(G_1) \cap L(G_2) = \emptyset$
- $L(G_1) = L(G_2)$, $L(G_1) \subseteq L(G_2)$
- $L(G) = L(R)$, $L(R) \subseteq L(G)$

Proofs: Post’s correspondence problems is reducible to these

Post’s correspondence problem

- It is a puzzle about pairs of lists of strings:
  $(w_1, w_2, \ldots, w_k)$ and $(x_1, x_2, \ldots, x_k)$
- Problem: Does an index sequence $(i_1, i_2, \ldots, i_m)$ exist, s.t. $(w_{i_1}, w_{i_2}, \ldots, w_{i_m})$ and $(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$?
- Example: $w = (1,0,10), x = (01,1,0)$.
  Solution: $i = (2, 1, 3)$
- Example: $w = (1,0), x = (10,1)$. No solution
- Theorem: PCP is undecidable
- Proof: HALT is reducible to PCP
How the decidability question arose

- Mathematicians around 1900 sought to place all mathematics on a firm logical basis
- David Hilbert posed the *Entscheidungs-problem* (decision problem): Algorithmically determine the truth or falsehood of *any* logical assertion about numbers
- In 1936, Alan Turing proposed the TM as a formal model of computation to address this problem

References


