UMB CS420

Turing Machines (TMIs)

Monday, March 27, 2023
CS 420: Where We’ve Been, Where We’re Going

- **Turing Machines (TMs)**
  - **Memory**: states + infinite **tape**, (arbitrary read/write)
  - Expresses any “computation”

- **PDAs**: recognize context-free languages
  - **Memory**: states + infinite **stack** (push/pop only)
  - Can’t express: arbitrary dependency,
    - e.g., \( \{ww \mid w \in \{0,1\}^*\} \)

- **DFAs / NFAs**: recognize regular langs
  - **Memory**: finite states
  - Can’t express: dependency
    - e.g., \( \{0^n1^n \mid n \geq 0\} \)

A special subset of TMs
Alan Turing

• First to formalize a model of computation
  • i.e., he invented many of the ideas in this course

• And worked as a codebreaker during WW2

• Also studied Artificial Intelligence
  • The Turing Test

ChatGPT passes the Turing test

In 1950, Alan Turing proposed the Turing test as a way to measure a machine's intelligence. The test pits a human against a machine in a conversation. If the machine can fool the human into thinking it is also human, then it is said to have passed the test. In December 2022, ChatGPT, an artificial intelligence chatbot, became the second chatbot to pass the Turing Test, according to Max Woott, a data scientist at BuzzFeed.

Google's LaMDA AI passed the Turing test in the summer of 2022, demonstrating that it is invalid. For many years, the Turing test has been used as a standard for sophisticated artificial intelligence models.

congrats to OpenAI on winning the Turing Test
Finite Automata vs Turing Machines

- **Turing Machines** can **read and write** to arbitrary “tape” cells
  - Tape initially contains input string

- **Tape is infinite**
  - To the right

- **Each step**: “head” can move left or right

- **Turing Machine** can **accept / reject** at any time

Call a language *Turing-recognizable* if some Turing machine recognizes it.
Turing Machine Example

Define: $M_1$ accepts inputs in language $B = \{ w\#w | w \in \{0,1\}^* \}$

$M_1 = \text{"On input string } w:\n\begin{enumerate}
\item Zig-zag across the tape to corresponding positions on either side of the } \# \text{ symbol to check whether these positions contain the same symbol. If they do not, or if no } \# \text{ is found, reject.}
\item Cross off symbols as they are checked to keep track of which symbols correspond.
\end{enumerate}$

This is a high-level TM description

It is equivalent to (but more concise than) our typical (low-level) tuple descriptions, i.e., one step = maybe multiple $\delta$ transitions

Analogy
\begin{itemize}
\item \text{"High-level"}: Python
\item \text{"Low-level"}: assembly language
\end{itemize}
Turing Machine Example

\[ M_1 \text{ accepts inputs in language } B = \{ w\#w \mid w \in \{0,1\}^* \} \]

\[ M_1 = \text{“On input string } w:\]

1. Zig-zag across the tape to corresponding positions on either side of the \# symbol to check whether these positions contain the same symbol. If they do not, or if no \# is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

“Cross off” = write “\(x\)” char

\[ \begin{array}{c}
0 & 1 & 1 & 0 & 0 & 0 & \# & 0 & 1 & 1 & 0 & 0 & 0 & \ldots \\
\hline
x & 1 & 1 & 0 & 0 & 0 & \# & 0 & 1 & 1 & 0 & 0 & 0 & \ldots
\end{array} \]
Turing Machine Example

$M_1$ accepts inputs in language $B = \{w\#w \mid w \in \{0,1\}^*\}$

$M_1 = \text{"On input string } w:"

1. Zig-zag across the tape to corresponding positions on either side of the $\#$ symbol to check whether these positions contain the same symbol. If they do not, or if no $\#$ is found, reject.

Cross off symbols as they are checked to keep track of which symbols correspond.

“Cross off” = write “$\times$" char
Turing Machine Example

\[ M_1 \text{ accepts inputs in language } B = \{ w\#w | w \in \{0,1\}^* \} \]

\[ M_1 = \text{“On input string } w: \]

1. Zig-zag across the tape to corresponding positions on either side of the \# symbol to check whether these positions contain the same symbol. If they do not, or if no \# is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.
Turing Machine Example

\( M_1 \) accepts inputs in language \( B = \{ w#w \mid w \in \{0,1\}^* \} \)

\( M_1 \) = “On input string \( w \):

1. Zig-zag across the tape to corresponding positions on either side of the \( # \) symbol to check whether these positions contain the same symbol. If they do not, or if no \( # \) is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.”
$M_1$ accepts inputs in language $B = \{w\#w \mid w \in \{0,1\}^*\}$

$M_1 =$ “On input string $w$:

1. Zig-zag across the tape to corresponding positions on either side of the $\#$ symbol to check whether these positions contain the same symbol. If they do not, or if no $\#$ is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

2. When all symbols to the left of the $\#$ have been crossed off, check for any remaining symbols to the right of the $\#$. If any symbols remain, reject; otherwise, accept.”
Turing Machine Example

\( M_1 \) accepts inputs in language \( B = \{ w\#w \mid w \in \{0,1\}^* \} \)

\( M_1 = \) “On input string \( w \):

1. Zig-zag across the tape to corresponding positions on either side of the \# symbol to check whether these positions contain the same symbol. If they do not, or if no \# is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

2. When all symbols to the left of the \# have been crossed off, check for any remaining symbols to the right of the \#. If any symbols remain, reject; otherwise, accept.”
A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where \(Q\), \(\Sigma\), \(\Gamma\) are all finite sets and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\sqcup\),
3. \(\Gamma\) is the tape alphabet, where \(\sqcup \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).

This is a “low-level” TM description.

Is this machine deterministic? Or non-deterministic?
Formal Turing Machine Example

\[ B = \{ w\#w \mid w \in \{0,1\}^* \} \]

Transitions on this side: Crossed off a 0

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

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the *blank symbol* \(\_\),
3. \(\Gamma\) is the tape alphabet, where \(\_ \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the initial state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where
\(Q\), \(\Sigma\), \(\Gamma\) are all finite sets and
1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\_\)
3. \(\Gamma\) is the tape alphabet, where \(\_ \in \Gamma \) and \(\Sigma \subseteq \Gamma\)
4. \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
Formal Turing Machine Example

\[
B = \{ w\#w \mid w \in \{0,1\}^* \}
\]

A Turing machine is a 7-tuple, \( (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}) \), where \( Q, \Sigma, \Gamma \) are all finite sets and

1. \( Q \) is the set of states,
2. \( \Sigma \) is the input alphabet not containing the blank symbol \( \sqcup \),
3. \( \Gamma \) is the tape alphabet, where \( \sqcup \in \Gamma \) and \( \Sigma \subseteq \Gamma \),
4. \( \delta \) is the transition function,
5. \( q_0 \in Q \) is the initial state,
6. \( q_{\text{accept}} \in Q \) is the accept state, and
7. \( q_{\text{reject}} \in Q \) is the reject state, where \( q_{\text{reject}} \neq q_{\text{accept}} \).
Formal Turing Machine Example

\[ B = \{ w\#w \mid w \in \{0,1\}^* \} \]

- Read char (0 or 1), cross it off, move head R(right)
- This side: Crossed off a 1

A **Turing machine** is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})\), where 
\begin{itemize}
  \item \(Q\) is the set of states,
  \item \(\Sigma\) is the input alphabet not containing the **blank symbol** \(\sqcup\),
  \item \(\Gamma\) is the tape alphabet, where \(\sqcup \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
  \item \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
  \item \(q_0 \in Q\) is the start state,
  \item \(q_{accept} \in Q\) is the accept state, and
  \item \(q_{reject} \in Q\) is the reject state, where \(q_{reject} \neq q_{accept}\).
\end{itemize}
A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\sqcup\),
3. \(\Gamma\) is the tape alphabet, where \(\sqcup \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).

Formal Turing Machine Example

\[ B = \{w\#w | w \in \{0,1\}^*\} \]
$M_1 = "\text{On input string } w:\"

1. Zig-zag across the tape side of the # symbol to the same symbol. If the Cross off symbols as the symbols correspond.

2. When all symbols to the left of the # have been crossed off, check for any remaining symbols remain, reject; otherwise, accept."
Turing Machine: High-level Description

- $M_1$ accepts if input is in language $B = \{w#w \mid w \in \{0,1\}^*\}$

$M_1$ = “On input string $w$:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are encountered in track of which symbols correspond.

2. When all symbols to the left and right of # have been crossed off, check for any remaining symbols left in part of the #. If any symbols remain, reject; otherwise, accept.”
TM High-level Description Tips

Analogy:
- **High-level** TM description ~ function definition in “high level” language, e.g. Python
- **Low-level** TM tuple ~ function definition in bytecode or assembly

TM high-level descriptions are **not** a “do whatever” card, some rules:
1. All TMs must have a name, e.g., $M_1$
2. Input strings must also be named (like a function parameter), e.g., $w$
3. TMs can “call” or “simulate” other TMs (if they pass appropriate arguments)
   - e.g., a step for a TM $M$ can say: “call TM $M_2$ with argument string $w$, if $M_2$ accepts $w$ then ..., else ...”
4. Follow typical programming “scoping” rules
   - can assume functions we’ve already defined are in “global” scope, RE2NFA ...
5. Other variables must also be defined (named) before they are used
   - e.g., can define a TM inside another TM
6. must be equivalent to a low-level formal tuple
   - high-level “step” represents a finite # of low-level $\delta$ transitions
   - So one step cannot run forever
   - E.g., can’t say “try all numbers” as a “step”
Non-halting Turing Machines (TMs)

- A Turing Machine can run forever
  - E.g., the head can move back and forth in a loop

- We will work with two classes of Turing Machines:
  - A recognizer is a Turing Machine that may run forever (all possible TMs)
  - A decider is a Turing Machine that always halts.

So a TM computation has 3 possible results:
- Accept
- Reject
- Loop forever

Call a language **Turing-recognizable** if some Turing machine recognizes it. (3 possible computation results)

Call a language **Turing-decidable** or simply **decidable** if some Turing machine decides it. (2 possible computation results)
Formal Definition of an “Algorithm”

- An **algorithm** is equivalent to a **Turing-decidable Language** (always halts)

Many functions we have defined this semester are **algorithms**! e.g., all our conversion functions are **deciders**!
- $\text{d2n}$
- $\text{RE2NFA}$
- $\text{n2p}$
More Turing Machine Variations
1. Multi-tape TMs

2. Non-deterministic TMs

We will prove that these TM variations are equivalent to deterministic, single-tape machines.
Reminder: Equivalence of Machines

- Two machines are **equivalent** when ...

- ... they recognize the same language
**Theorem:** Single-tape TM $\iff$ Multi-tape TM

$\Rightarrow$ If a **single**-tape TM recognizes a language, then a **multi**-tape TM recognizes the language
  - Single-tape TM is equivalent to ...
  - ... multi-tape TM that only uses one of its tapes
  - (could you write out the formal conversion?)

$\Leftarrow$ If a **multi**-tape TM recognizes a language, then a **single**-tape TM recognizes the language
  - **Convert:** multi-tape TM $\rightarrow$ single-tape TM
Multi-tape TM $\Rightarrow$ Single-tape TM

Idea: Use delimiter ($\#$) on single-tape to simulate multiple tapes
• Add “dotted” version of every char to simulate multiple heads
Theorem: Single-tape TM $\iff$ Multi-tape TM

$
\Rightarrow$ If a single-tape TM recognizes a language, then a multi-tape TM recognizes the language
  • Single-tape TM is equivalent to ...
  • ... multi-tape TM that only uses one of its tapes

$
\Leftarrow$ If a multi-tape TM recognizes a language, then a single-tape TM recognizes the language
  • Convert: multi-tape TM $\Rightarrow$ single-tape TM
Check-in Quiz 3/27

On gradescope