Turing machines and descriptive complexity

Sam van Gool

Wednesday, May 1, 2019

Logic and Computation, Master's course, Utrecht University

Central question in theory of computation

What are the fundamental capabilities

and limitations of computers?

Three types of sub-questions, plus a bonus question

- Complexity theory
  - How *hard* is a computational problem?
  - How much *time*? How much *memory space*?
- Computability theory
  - Is the answer to a problem *computable*, or not?
- Automata theory
  - What is a *minimal* model for computation?
- For each of the above, what is the connection to *logic*?

Overview for today

Turing machines and Decidability

Undecidability

Complexity classes

Describing complexity via logic

Turing machines and Decidability

Turing machines vs. Automata

> What are the essential components of a computer?

A computer (Turing machine) has:

- Initial input
- Ability to read from memory
- Ability to write to memory
- Final (yes/no) output
- Internal states

An automaton has:

- Initial input
- Ability to read from memory:

  once and in one direction
- ~~Ability to write to memory~~
- Final (yes/no) output
- Internal states

Turing machines vs. Automata: four differences

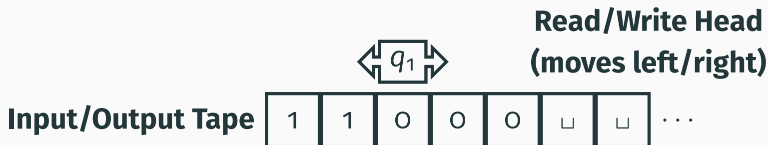
1. A Turing machine can read and write;
2. There is a read/write head that can move right and left;
3. The tape (memory) is infinite;
4. There are two special accept and reject states, which take effect as soon as they are reached.

Turing machine, schematically

Input/Output Tape	1	1	0	0	0	␣	␣	···

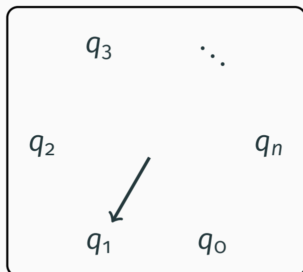
Reference: http://www.texample.net/tikz/examples/turing-machine-2/

Turing machine, schematically

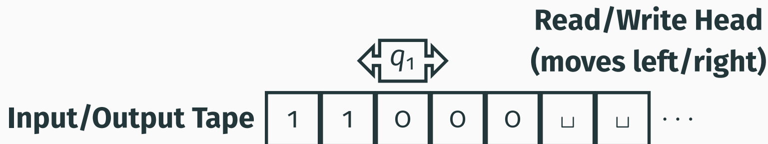


Reference: http://www.texample.net/tikz/examples/turing-machine-2/

Turing machine, schematically

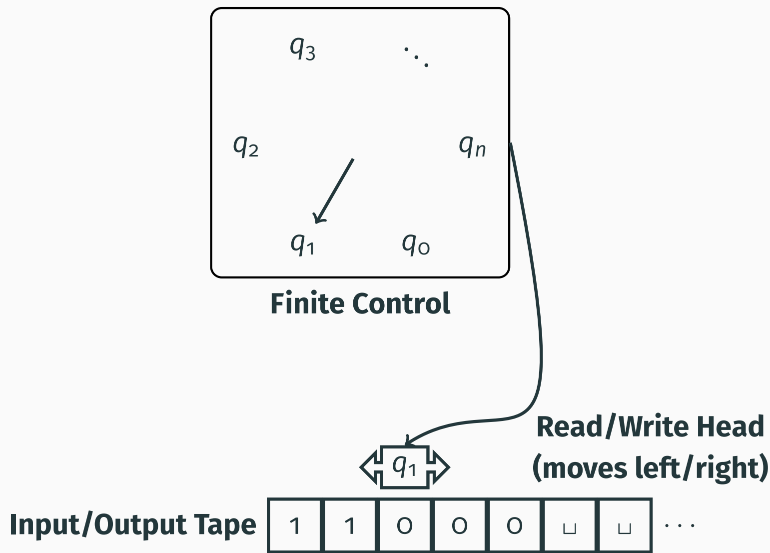


Finite Control



Reference: http://www.texample.net/tikz/examples/turing-machine-2/

Turing machine, schematically



Reference: http://www.texample.net/tikz/examples/turing-machine-2/

Turing machine, example

How to compute the language

$$L = \{w\#w \mid w \in \{0,1\}^*\}?$$

Informal description of the Turing machine

$$L = \{w\#w \mid w \in \{0,1\}^*\}$$

The Turing machine ‘checkIdenticalBinaryString’:

Informal description of the Turing machine

$$L = \{w\#w \mid w \in \{0, 1\}^*\}$$

The Turing machine 'checkIdenticalBinaryString':

1. Read an unmarked symbol: 0 or 1, and mark it with an '*x*';

Informal description of the Turing machine

$$L = \{w\#w \mid w \in \{0, 1\}^*\}$$

The Turing machine 'checkIdenticalBinaryString':

1. Read an unmarked symbol: 0 or 1, and mark it with an 'x';
2. Move to the right until you encounter '$\#$';

Informal description of the Turing machine

$$L = \{w\#w \mid w \in \{0, 1\}^*\}$$

The Turing machine ‘checkIdenticalBinaryString’:

1. Read an unmarked symbol: 0 or 1, and mark it with an ‘x’;

2. Move to the right until you encounter ‘$\#$’;
3. After that, move to the next unmarked symbol;
4. Check that it is 0 or 1 (according to step 1), and mark it with ‘x’;

Informal description of the Turing machine

$$L = \{w\#w \mid w \in \{0,1\}^*\}$$

The Turing machine 'checkIdenticalBinaryString':

1. Read an unmarked symbol: 0 or 1, and mark it with an 'x';

2. Move to the right until you encounter '$\#$';
3. After that, move to the next unmarked symbol;
4. Check that it is 0 or 1 (according to step 1), and mark it with 'x';
5. Move back left until you encounter '$\#$';
6. Move back left to the last 'x', then move one right and go to step 1.

Informal description of the Turing machine

$$L = \{w\#w \mid w \in \{0, 1\}^*\}$$

The Turing machine 'checkIdenticalBinaryString':

1. Read an unmarked symbol: 0 or 1, and mark it with an '*x*';
    - if it is '#' immediately, go to step 7.
2. Move to the right until you encounter '#';
3. After that, move to the next unmarked symbol;
4. Check that it is 0 or 1 (according to step 1), and mark it with '*x*';
5. Move back left until you encounter '#';
6. Move back left to the last '*x*', then move one right and go to step 1.

Informal description of the Turing machine

$$L = \{w\#w \mid w \in \{0, 1\}^*\}$$

The Turing machine ‘checkIdenticalBinaryString’:

1. Read an unmarked symbol: 0 or 1, and mark it with an ‘x’;
    - if it is ‘$\#$’ immediately, go to step 7.
2. Move to the right until you encounter ‘$\#$’;
3. After that, move to the next unmarked symbol;
4. Check that it is 0 or 1 (according to step 1), and mark it with ‘x’;
5. Move back left until you encounter ‘$\#$’;
6. Move back left to the last ‘x’, then move one right and go to step 1.
7. Move to the right to check that everything is marked until the first ‘$\sqcup$’.
8. Accept (whenever stuck before: reject).

Informal description of the Turing machine

$$L = \{w\#w \mid w \in \{0,1\}^*\}$$

The Turing machine ‘checkIdenticalBinaryString’:

1. Read an unmarked symbol: 0 or 1, and mark it with an ‘x’;
   - if it is ‘$\#$’ immediately, go to step 7.
2. Move to the right until you encounter ‘$\#$’;
3. After that, move to the next unmarked symbol;
4. Check that it is 0 or 1 (according to step 1), and mark it with ‘x’;
5. Move back left until you encounter ‘$\#$’;
6. Move back left to the last ‘x’, then move one right and go to step 1.
7. Move to the right to check that everything is marked until the first ‘␣’.
8. Accept (whenever stuck before: reject).

$\rightarrow$ Simulation

Definition

A *Turing machine* is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$, where

- Q is a finite set of *states*;
- Σ is a finite *input alphabet* not containing the symbol $\sqcup$;
- Γ is a finite *tape alphabet* containing $\Sigma \cup \{\sqcup\}$;
- $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a *transition function*;
- $q_0 \in Q$ is the start state;
- $q_a \in Q$ is the accept state;
- $q_r \in Q$ is the reject state; $q_r \neq q_a$.

Accept, reject, halt, loop

Definition

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ be a Turing machine.

- A *configuration* is a 3-tuple (u, q, v), where $u, v \in \Gamma^*$ and $q \in Q$.

Accept, reject, halt, loop

Definition

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ be a Turing machine.

- A *configuration* is a 3-tuple (u, q, v), where $u, v \in \Gamma^*$ and $q \in Q$.
- A configuration C *yields* a configuration C' if it agrees with δ.

Accept, reject, halt, loop

Definition

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ be a Turing machine.

- A *configuration* is a 3-tuple (u, q, v), where $u, v \in \Gamma^*$ and $q \in Q$.
- A configuration C *yields* a configuration C' if it agrees with δ.
- We say M *accepts* $w \in \Sigma^*$ if there is a sequence of configurations $C_1, \ldots, C_k$, with $C_1 = (\epsilon, q_0, w)$ and $C_k = (u_k, q_a, v_k)$, where C_i yields C_{i+1} for each $1 \leq i < k$.

Accept, reject, halt, loop

Definition

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ be a Turing machine.

- A *configuration* is a 3-tuple (u, q, v), where $u, v \in \Gamma^*$ and $q \in Q$.
- A configuration C *yields* a configuration C' if it agrees with δ.
- We say M *accepts* $w \in \Sigma^*$ if there is a sequence of configurations $C_1, \ldots, C_k$, with $C_1 = (\epsilon, q_0, w)$ and $C_k = (u_k, q_a, v_k)$, where C_i yields C_{i+1} for each $1 \leq i < k$.
- $L(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$.

Accept, reject, halt, loop

Definition

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ be a Turing machine.

- A *configuration* is a 3-tuple (u, q, v), where $u, v \in \Gamma^*$ and $q \in Q$.
- A configuration C *yields* a configuration C' if it agrees with δ.
- We say M *accepts* $w \in \Sigma^*$ if there is a sequence of configurations $C_1, \ldots, C_k$, with $C_1 = (\epsilon, q_0, w)$ and $C_k = (u_k, q_a, v_k)$, where C_i yields C_{i+1} for each $1 \leq i < k$.
- $L(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$.
- M *rejects* w: same definition but with q_r in place of q_a.

Accept, reject, halt, loop

Definition

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ be a Turing machine.

- A *configuration* is a 3-tuple (u, q, v), where $u, v \in \Gamma^*$ and $q \in Q$.
- A configuration C *yields* a configuration C' if it agrees with δ.
- We say M *accepts* $w \in \Sigma^*$ if there is a sequence of configurations $C_1, \ldots, C_k$, with $C_1 = (\epsilon, q_0, w)$ and $C_k = (u_k, q_a, v_k)$, where C_i yields C_{i+1} for each $1 \leq i < k$.
- $L(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$.
- M *rejects* w: same definition but with q_r in place of q_a.
- We say M *halts on* w if it either accepts or rejects w.

Accept, reject, halt, loop

Definition

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ be a Turing machine.

- A *configuration* is a 3-tuple (u, q, v), where $u, v \in \Gamma^*$ and $q \in Q$.
- A configuration C *yields* a configuration C' if it agrees with δ.
- We say M *accepts* $w \in \Sigma^*$ if there is a sequence of configurations $C_1, \ldots, C_k$, with $C_1 = (\epsilon, q_0, w)$ and $C_k = (u_k, q_a, v_k)$, where C_i yields C_{i+1} for each $1 \leq i < k$.
- $L(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$.
- M *rejects* w: same definition but with q_r in place of q_a.
- We say M *halts on* w if it either accepts or rejects w.
- If M neither accepts nor rejects w, we say it *loops* or *does not halt* on w.

> **Exercise.** Design a Turing machine that does not halt on any input. $\rightarrow$ Solution

Recognizability and decidability

Definition

A language L is called recursively enumerable or Turing-recognizable if $L = L(M)$ for some Turing machine M:

Recognizability and decidability

Definition

A language L is called recursively enumerable or Turing-recognizable if $L = L(M)$ for some Turing machine M:

- if $w \in L$ then M accepts w, and
- if $w \notin L$ then M does not accept w (it may reject or loop).

Recognizability and decidability

Definition

A language L is called recursively enumerable or Turing-recognizable if $L = L(M)$ for some Turing machine M:

- if $w \in L$ then M accepts w, and
- if $w \notin L$ then M does not accept w (it may reject or loop).

Definition

A language L is called recursive or (Turing-)decidable if $L = L(M)$ for some Turing machine M *which halts on all inputs*:

Recognizability and decidability

Definition

A language L is called recursively enumerable or Turing-recognizable if $L = L(M)$ for some Turing machine M:

- if $w \in L$ then M accepts w, and
- if $w \notin L$ then M does not accept w (it may reject or loop).

Definition

A language L is called recursive or (Turing-)decidable if $L = L(M)$ for some Turing machine M *which halts on all inputs*:

- if $w \in L$ then M accepts w, and
- if $w \notin L$ then M rejects w.

Variants of Turing machines

There are various more flexible, but equivalent Turing machine notions:

Variants of Turing machines

There are various more flexible, but equivalent Turing machine notions:

- Non-determinism: $\delta \colon Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$;

Variants of Turing machines

There are various more flexible, but equivalent Turing machine notions:

- Non-determinism: $\delta \colon Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$;
- Multiple tapes;

Variants of Turing machines

There are various more flexible, but equivalent Turing machine notions:

- Non-determinism: $\delta\colon Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$;
- Multiple tapes;
- Turing machines with a two-way infinite tape;

Variants of Turing machines

There are various more flexible, but equivalent Turing machine notions:

- Non-determinism: $\delta \colon Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$;
- Multiple tapes;
- Turing machines with a two-way infinite tape;
- Turing machines with an additional 'stay put' instruction;
- ...

Variants of Turing machines

There are various more flexible, but equivalent Turing machine notions:

- Non-determinism: $\delta\colon Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$;
- Multiple tapes;
- Turing machines with a two-way infinite tape;
- Turing machines with an additional ‘stay put’ instruction;
- ...

Church-Turing Thesis $\Rightarrow$ there is no *stronger* type of machine.

Variants of Turing machines

There are various more flexible, but equivalent Turing machine notions:

- Non-determinism: $\delta \colon Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$;
- Multiple tapes;
- Turing machines with a two-way infinite tape;
- Turing machines with an additional ‘stay put’ instruction;
- ...

Church-Turing Thesis $\Rightarrow$ there is no *stronger* type of machine.

But: the precise notion of Turing machine can make a (big!) difference for efficiency.

Algorithms

- An *algorithm* informally describes a Turing machine.

Algorithms

- An *algorithm* informally describes a Turing machine.

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

Algorithms

- An *algorithm* informally describes a Turing machine.

$>$ Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

- **Problem specification.**
  - Input: the encoding, $\langle G \rangle$, of a finite graph G.

Algorithms

- An *algorithm* informally describes a Turing machine.

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

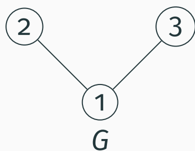
- **Problem specification.**
  - Input: the encoding, $\langle G \rangle$, of a finite graph G.
  - Output: accept $\langle G \rangle$ if it is the encoding of a connected graph G, reject otherwise.

Algorithms

- An *algorithm* informally describes a Turing machine.

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

- **Problem specification.**
    - Input: the encoding, $\langle G \rangle$, of a finite graph G.
    - Output: accept $\langle G \rangle$ if it is the encoding of a connected graph G, reject otherwise.
- Example of an encoding:



$(1, 2, 3)((1, 2), (1, 3))$

$\langle G \rangle$

$$CONN := \{\langle G \rangle \mid G \text{ is a finite connected graph.}\}$$

> Is the language CONN decidable?

Algorithms

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

- Given an input string w:

Algorithms

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

- Given an input string w:
    1. Check that w is the encoding of some finite graph G;

Algorithms

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

- Given an input string w:
  1. Check that w is the encoding of some finite graph G;
  2. Mark a node of G;

Algorithms

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

- Given an input string w:
  1. Check that w is the encoding of some finite graph G;
  2. Mark a node of G;
  3. Mark any node which has an edge to a node that is already marked;

Algorithms

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

- Given an input string w:
  1. Check that w is the encoding of some finite graph G;
  2. Mark a node of G;
  3. Mark any node which has an edge to a node that is already marked;
  4. If at least one new node was marked, repeat step 3.

Algorithms

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

- Given an input string w:
  1. Check that w is the encoding of some finite graph G;
  2. Mark a node of G;
  3. Mark any node which has an edge to a node that is already marked;
  4. If at least one new node was marked, repeat step 3.
  5. Scan and check if all nodes of G are marked; if so, accept ; otherwise, reject .

Algorithms

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

- Given an input string w:
  1. Check that w is the encoding of some finite graph G;
  2. Mark a node of G;
  3. Mark any node which has an edge to a node that is already marked;
  4. If at least one new node was marked, repeat step 3.
  5. Scan and check if all nodes of G are marked; if so, accept ; otherwise, reject .
- One can *implement* this algorithm in a Turing machine.

Algorithms

> Show that it is decidable whether a finite graph is *connected*, i.e., there is a path between any two nodes.

- Given an input string w:
  1. Check that w is the encoding of some finite graph G;
  2. Mark a node of G;
  3. Mark any node which has an edge to a node that is already marked;
  4. If at least one new node was marked, repeat step 3.
  5. Scan and check if all nodes of G are marked; if so, accept ; otherwise, reject .
- One can *implement* this algorithm in a Turing machine.
- One can *prove* that this algorithm always *terminates* and yields the *correct* output on any finite graph.

Algorithms for automata

> Show that it is decidable if a DFA $\mathcal{A}$ accepts a word w.

- Given as input an encoding of the pair $(\mathcal{A}, w)$:

Algorithms for automata

> Show that it is decidable if a DFA $\mathcal{A}$ accepts a word w.

- Given as input an encoding of the pair $(\mathcal{A}, w)$:
    1. Simulate $\mathcal{A}$ on input w;
    2. If the simulation ends up in an accepting state, accept ; otherwise, reject .

Algorithms for automata

> Show that it is decidable if a DFA $\mathcal{A}$ accepts a word w.

- Given as input an encoding of the pair $(\mathcal{A}, w)$:
  1. Simulate $\mathcal{A}$ on input w;
  2. If the simulation ends up in an accepting state, accept ; otherwise, reject .
- This is an informal description of a Turing machine for the language

$$A_{DFA} := \{\langle(\mathcal{A}, w)\rangle \mid \mathcal{A} \text{ is a DFA which accepts } w\}.$$

Algorithms for automata

> Show that it is decidable if a DFA $\mathcal{A}$ accepts a word w.

- Given as input an encoding of the pair $(\mathcal{A}, w)$:
    1. Simulate $\mathcal{A}$ on input w;
    2. If the simulation ends up in an accepting state, accept ; otherwise, reject .
- This is an informal description of a Turing machine for the language

$$A_{DFA} := \{\langle(\mathcal{A}, w)\rangle \mid \mathcal{A} \text{ is a DFA which accepts } w\}.$$

- Also decidable: acceptance by an NFA.

Algorithms for automata

> Show that it is decidable if a DFA $\mathcal{A}$ accepts a word w.

- Given as input an encoding of the pair $(\mathcal{A}, w)$:
    1. Simulate $\mathcal{A}$ on input w;
    2. If the simulation ends up in an accepting state, accept ; otherwise, reject .
- This is an informal description of a Turing machine for the language

$$A_{DFA} := \{\langle(\mathcal{A}, w)\rangle \mid \mathcal{A} \text{ is a DFA which accepts } w\}.$$

- Also decidable: acceptance by an NFA.
- Also decidable if an automaton accepts *any* word at all. (“Emptiness problem”)
- Also decidable if two automata accept *the same* words. (“Equivalence problem”)

A semi-algorithm for Turing machines

> How about $A_{TM} := \{\langle (M, w) \rangle \mid M \text{ is a TM which accepts } w\}$?

A semi-algorithm for Turing machines

> How about $A_{TM} := \{\langle (M, w) \rangle \mid M \text{ is a TM which accepts } w\}$?

- There exists a (single!) Turing machine, U, which takes as input an encoding $u = \langle (M, w) \rangle$, and outputs *accept* if M outputs *accept* on w, and outputs *reject* if M outputs *reject* on w.

A semi-algorithm for Turing machines

> How about $A_{TM} := \{\langle (M, w) \rangle \mid M \text{ is a TM which accepts } w\}$?

- There exists a (single!) Turing machine, U, which takes as input an encoding $u = \langle (M, w) \rangle$, and outputs *accept* if M outputs *accept* on w, and outputs *reject* if M outputs *reject* on w.
- U is called a universal Turing machine.

A semi-algorithm for Turing machines

> How about $A_{TM} := \{\langle(M, w)\rangle \mid M \text{ is a TM which accepts } w\}$?

- There exists a (single!) Turing machine, U, which takes as input an encoding $u = \langle(M, w)\rangle$, and outputs *accept* if M outputs *accept* on w, and outputs *reject* if M outputs *reject* on w.
- U is called a universal Turing machine.
- The machine U *simulates* what M would do with w.

A semi-algorithm for Turing machines

> How about $A_{TM} := \{\langle (M, w) \rangle \mid M \text{ is a TM which accepts } w\}$?

- There exists a (single!) Turing machine, U, which takes as input an encoding $u = \langle (M, w) \rangle$, and outputs *accept* if M outputs *accept* on w, and outputs *reject* if M outputs *reject* on w.
- U is called a universal Turing machine.
- The machine U *simulates* what M would do with w.
- If M loops on w, then in particular M does not accept w.

A semi-algorithm for Turing machines

> How about $A_{TM} := \{\langle (M, w) \rangle \mid M \text{ is a TM which accepts } w\}$?

- There exists a (single!) Turing machine, U, which takes as input an encoding $u = \langle (M, w) \rangle$, and outputs *accept* if M outputs *accept* on w, and outputs *reject* if M outputs *reject* on w.
- U is called a universal Turing machine.
- The machine U *simulates* what M would do with w.
- If M loops on w, then in particular M does not accept w.
- But the machine U does not know this!

A semi-algorithm for Turing machines

> How about $A_{TM} := \{\langle(M, w)\rangle \mid M \text{ is a TM which accepts } w\}$?

- There exists a (single!) Turing machine, U, which takes as input an encoding $u = \langle(M, w)\rangle$, and outputs *accept* if M outputs *accept* on w, and outputs *reject* if M outputs *reject* on w.
- U is called a universal Turing machine.
- The machine U *simulates* what M would do with w.
- If M loops on w, then in particular M does not accept w.
- But the machine U does not know this!
- U only shows that A_{TM} is recursively enumerable.

A semi-algorithm for Turing machines

> How about $A_{TM} := \{\langle(M, w)\rangle \mid M \text{ is a TM which accepts } w\}$?

- There exists a (single!) Turing machine, U, which takes as input an encoding $u = \langle(M, w)\rangle$, and outputs *accept* if M outputs *accept* on w, and outputs *reject* if M outputs *reject* on w.
- U is called a universal Turing machine.
- The machine U *simulates* what M would do with w.
- If M loops on w, then in particular M does not accept w.
- But the machine U does not know this!
- U only shows that A_{TM} is recursively enumerable.
- The language A_{TM} is not recursive.

Recognizability and decidability

Definition

A language L is called recursively enumerable or Turing-recognizable if $L = L(M)$ for some Turing machine M:

- if $w \in L$ then M accepts w, and
- if $w \notin L$ then M does not accept w (it may reject or loop).

Definition

A language L is called recursive or (Turing-)decidable if $L = L(M)$ for some Turing machine M *which halts on all inputs*:

- if $w \in L$ then M accepts w, and
- if $w \notin L$ then M rejects w.

Undecidability

The Halting Problem

Theorem (Turing, 1936)

It is undecidable whether a Turing machine M halts on an input w.

The Halting Problem

Theorem (Turing, 1936)
It is undecidable whether a Turing machine M halts on an input w.

- **Problem specification.** ("The Halting Problem")
  - Input: An encoding $\langle (M, w) \rangle$ of a pair (M, w), where M is a Turing machine and w is an input string.
  - Desired output: Accept the encoding $\langle (M, w) \rangle$ if M halts on input w, reject otherwise.

The Halting Problem

Theorem (Turing, 1936)

It is undecidable whether a Turing machine M halts on an input w.

- **Problem specification.** (“The Halting Problem”)
  - Input: An encoding $\langle (M, w) \rangle$ of a pair (M, w), where M is a Turing machine and w is an input string.
  - Desired output: Accept the encoding $\langle (M, w) \rangle$ if M halts on input w, reject otherwise.
- Turing’s Theorem: no Turing machine can do this!

Proof that the Halting problem is undecidable

- Suppose, towards a contradiction, that H were a Turing machine which decides the Halting problem: it accepts the encoding of (M, w) if M halts on w, and it rejects the encoding of (M, w) if M loops on w.

Proof that the Halting problem is undecidable

- Suppose, towards a contradiction, that H were a Turing machine which decides the Halting problem: it accepts the encoding of (M, w) if M halts on w, and it rejects the encoding of (M, w) if M loops on w.
- We use H to define a new Turing machine, N, which takes as input an encoding $\langle (M, \epsilon) \rangle$ of a Turing machine with empty input word, and does the following:

Proof that the Halting problem is undecidable

- Suppose, towards a contradiction, that H were a Turing machine which decides the Halting problem: it accepts the encoding of (M, w) if M halts on w, and it rejects the encoding of (M, w) if M loops on w.
- We use H to define a new Turing machine, N, which takes as input an encoding $\langle(M, \epsilon)\rangle$ of a Turing machine with empty input word, and does the following:
    1. Simulate the Turing machine H **with input** $\langle M, \langle(M, \epsilon)\rangle\rangle$;

Proof that the Halting problem is undecidable

- Suppose, towards a contradiction, that H were a Turing machine which decides the Halting problem: it accepts the encoding of (M, w) if M halts on w, and it rejects the encoding of (M, w) if M loops on w.
- We use H to define a new Turing machine, N, which takes as input an encoding $\langle (M, \epsilon) \rangle$ of a Turing machine with empty input word, and does the following:
    1. Simulate the Turing machine H **with input** $\langle M, \langle (M, \epsilon) \rangle \rangle$;
    2. If the simulation ends in reject , then reject ;

Proof that the Halting problem is undecidable

- Suppose, towards a contradiction, that H were a Turing machine which decides the Halting problem: it accepts the encoding of (M, w) if M halts on w, and it rejects the encoding of (M, w) if M loops on w.
- We use H to define a new Turing machine, N, which takes as input an encoding $\langle (M, \epsilon) \rangle$ of a Turing machine with empty input word, and does the following:
  1. Simulate the Turing machine H **with input** $\langle M, \langle (M, \epsilon) \rangle \rangle$;
  2. If the simulation ends in reject , then reject ;
  3. If the simulation ends in accept , then loop forever.

Proof that the Halting problem is undecidable

- Suppose, towards a contradiction, that H were a Turing machine which decides the Halting problem: it accepts the encoding of (M, w) if M halts on w, and it rejects the encoding of (M, w) if M loops on w.
- We use H to define a new Turing machine, N, which takes as input an encoding $\langle (M, \epsilon) \rangle$ of a Turing machine with empty input word, and does the following:
  1. Simulate the Turing machine H **with input** $\langle M, \langle (M, \epsilon) \rangle \rangle$;
  2. If the simulation ends in reject , then reject ;
  3. If the simulation ends in accept , then loop forever.
- **What does N do on input $n := \langle (N, \epsilon) \rangle$?**

Proof that the Halting problem is undecidable

- Suppose, towards a contradiction, that H were a Turing machine which decides the Halting problem: it accepts the encoding of (M, w) if M halts on w, and it rejects the encoding of (M, w) if M loops on w.
- We use H to define a new Turing machine, N, which takes as input an encoding $\langle (M, \epsilon) \rangle$ of a Turing machine with empty input word, and does the following:
  1. Simulate the Turing machine H **with input** $\langle M, \langle (M, \epsilon) \rangle \rangle$;
  2. If the simulation ends in reject , then reject ;
  3. If the simulation ends in accept , then loop forever.
- **What does N do on input $n := \langle (N, \epsilon) \rangle$?**
  - If H accepts $\langle (N, n) \rangle$, then N loops on input n;

Proof that the Halting problem is undecidable

- Suppose, towards a contradiction, that H were a Turing machine which decides the Halting problem: it accepts the encoding of (M, w) if M halts on w, and it rejects the encoding of (M, w) if M loops on w.
- We use H to define a new Turing machine, N, which takes as input an encoding $\langle(M, \epsilon)\rangle$ of a Turing machine with empty input word, and does the following:
  1. Simulate the Turing machine H **with input** $\langle M, \langle(M, \epsilon)\rangle\rangle$;
  2. If the simulation ends in reject , then reject ;
  3. If the simulation ends in accept , then loop forever.
- **What does N do on input $n := \langle(N, \epsilon)\rangle$?**
  - If H accepts $\langle(N, n)\rangle$, then N loops on input n;
  - If H rejects $\langle(N, n)\rangle$, then N rejects , so N halts on input n.

Proof that the Halting problem is undecidable

- Suppose, towards a contradiction, that H were a Turing machine which decides the Halting problem: it accepts the encoding of (M, w) if M halts on w, and it rejects the encoding of (M, w) if M loops on w.
- We use H to define a new Turing machine, N, which takes as input an encoding $\langle(M, \epsilon)\rangle$ of a Turing machine with empty input word, and does the following:
  1. Simulate the Turing machine H **with input** $\langle M, \langle(M, \epsilon)\rangle\rangle$;
  2. If the simulation ends in reject , then reject ;
  3. If the simulation ends in accept , then loop forever.
- **What does N do on input $n := \langle(N, \epsilon)\rangle$?**
  - If H accepts $\langle(N, n)\rangle$, then N loops on input n;
  - If H rejects $\langle(N, n)\rangle$, then N rejects , so N halts on input n.
- In both cases, H gives the wrong answer on input n. □

No general procedure for bug checks will do.

Now, I won't just assert that, I'll prove it to you.

I will prove that although you might work till you drop,

you cannot tell if computation will stop.

excerpt from: Geoffrey K. Pullum, "Scooping the Loop Snooper" (2000)

Undecidability

- Do undecidable problems occur “in nature”?

Undecidability

- Do undecidable problems occur "in nature"?
- **Matrix mortality problem.** (Paterson 1970, Neary 2013)
  Given as input five 3×3 integer matrices $M_1, \ldots, M_5$, decide whether or not the zero matrix, $\mathbf{0}$, can be obtained by multiplying them.

Undecidability

- Do undecidable problems occur "in nature"?
- **Matrix mortality problem.** (Paterson 1970, Neary 2013)
  Given as input five 3×3 integer matrices $M_1, \ldots, M_5$, decide whether or not the zero matrix, $\mathbf{0}$, can be obtained by multiplying them.
- **Post correspondence problem.** (Post 1946)
  Given as input a finite collection of 'word dominos', e.g.,
  $$\left\{ \frac{a}{ab}, \frac{b}{ca}, \frac{ca}{a}, \frac{abc}{c} \right\},$$
  decide whether or not there exists a match, i.e., a way to get the same word above and below,

Undecidability

- Do undecidable problems occur "in nature"?
- **Matrix mortality problem.** (Paterson 1970, Neary 2013)
  Given as input five 3×3 integer matrices $M_1, \ldots, M_5$, decide whether or not the zero matrix, $\mathbf{0}$, can be obtained by multiplying them.
- **Post correspondence problem.** (Post 1946)
  Given as input a finite collection of 'word dominos', e.g.,

$$\left\{ \frac{a}{ab}, \frac{b}{ca}, \frac{ca}{a}, \frac{abc}{c} \right\},$$

  decide whether or not there exists a match, i.e., a way to get the same word above and below, e.g.,

$$\frac{a|b|ca|a|abc}{ab|ca|a|ab|c}.$$

Complexity classes

Complexity Theory

- Suppose L is a decidable language.
- There are *many* Turing machines/algorithms which decide L.

- Suppose L is a decidable language.
- There are *many* Turing machines/algorithms which decide L.
- How can we distinguish between them?
- Are certain algorithms better than others?
- How can we be sure no better algorithm exists?

Complexity Theory

- Suppose L is a decidable language.
- There are *many* Turing machines/algorithms which decide L.
- How can we distinguish between them?
- Are certain algorithms better than others?
- How can we be sure no better algorithm exists?
- Complexity can be measured in terms of *time* and *space*.

Complexity Theory

- Suppose L is a decidable language.
- There are *many* Turing machines/algorithms which decide L.
- How can we distinguish between them?
- Are certain algorithms better than others?
- How can we be sure no better algorithm exists?
- Complexity can be measured in terms of *time* and *space*.
- We only discuss time complexity today.

Definition

Suppose M is a (deterministic or non-deterministic) Turing machine that halts on all inputs.

The *time complexity function* of M is defined as:

$$f_M(n) := \max_{|w| \leq n} \#\text{steps in any computation of } M \text{ on } w.$$

Linear and polynomial time

- We say M runs in *linear time* if there exists a number C such that $f_M(n) \leq Cn$ for n sufficiently large.

Linear and polynomial time

- We say M runs in *linear time* if there exists a number C such that $f_M(n) \leq Cn$ for n sufficiently large.
- We say M runs in *(deterministic) polynomial time* if there exist numbers k and C such that $f_M(n) \leq Cn^k$ for n sufficiently large.

Linear and polynomial time

- We say M runs in *linear time* if there exists a number C such that $f_M(n) \leq Cn$ for n sufficiently large.
- We say M runs in *(deterministic) polynomial time* if there exist numbers k and C such that $f_M(n) \leq Cn^k$ for n sufficiently large.
- We say a language L *can be decided in polynomial time*, or simply "L is in **P**" if there exists a polynomial time deterministic Turing machine which decides L.
- **Cobham's Thesis.** Problems in **P** are 'efficiently solvable' or 'tractable'.

Examples of problems in P

- Given a graph G and two nodes s and t in G, decide if G has a path from s to t.
- Given a graph G and a coloring of the nodes with three colors, decide if the coloring is *correct*, i.e., do adjacent nodes have distinct colors?
- Given an $n \times n$ filled Sudoku grid, decide if it is filled out correctly.
- Any linear programming problem.
- Given a number n, decide if it is prime.

Examples of problems in P

- Given a graph G and two nodes s and t in G, decide if G has a path from s to t.
- Given a graph G and a coloring of the nodes with three colors, decide if the coloring is *correct*, i.e., do adjacent nodes have distinct colors?
- Given an $n \times n$ filled Sudoku grid, decide if it is filled out correctly.
- Any linear programming problem.
- Given a number n, decide if it is prime.
  (Shown to be in **P** in 2002.)

Non-deterministic polynomial time

- We say a language L *can be decided in non-deterministic polynomial time*, or simply "L is in **NP**" if there exists a polynomial time non-deterministic Turing machine which decides L.
- Since deterministic Turing machines are a special case of non-deterministic Turing machines, obviously $\mathbf{P} \subseteq \mathbf{NP}$.

Non-deterministic polynomial time

- We say a language L *can be decided in non-deterministic polynomial time*, or simply "L is in **NP**" if there exists a polynomial time non-deterministic Turing machine which decides L.
- Since deterministic Turing machines are a special case of non-deterministic Turing machines, obviously $\mathbf{P} \subseteq \mathbf{NP}$.

Examples of problems in NP

- **Hamiltonian path problem.** Given a graph G and nodes s and t, is there a path from s to t *which visits each node exactly once*?

Examples of problems in NP

- **Hamiltonian path problem.** Given a graph G and nodes s and t, is there a path from s to t *which visits each node exactly once*?
- Given a graph G, is there a correct 3-coloring?

Examples of problems in NP

- **Hamiltonian path problem.** Given a graph G and nodes s and t, is there a path from s to t *which visits each node exactly once*?
- Given a graph G, is there a correct 3-coloring?
- Given a finite list of numbers $x_1, \ldots, x_k$ and a target number t, is there a sublist $x_{i_1}, \ldots, x_{i_k}$ which adds up to t?

Examples of problems in NP

- **Hamiltonian path problem.** Given a graph G and nodes s and t, is there a path from s to t *which visits each node exactly once*?
- Given a graph G, is there a correct 3-coloring?
- Given a finite list of numbers $x_1, \ldots, x_k$ and a target number t, is there a sublist $x_{i_1}, \ldots, x_{i_k}$ which adds up to t?
- Given an $n \times n$ Sudoku puzzle, does it have a solution?

Examples of problems in NP

- **Hamiltonian path problem.** Given a graph G and nodes s and t, is there a path from s to t *which visits each node exactly once*?
- Given a graph G, is there a correct 3-coloring?
- Given a finite list of numbers $x_1, \ldots, x_k$ and a target number t, is there a sublist $x_{i_1}, \ldots, x_{i_k}$ which adds up to t?
- Given an $n \times n$ Sudoku puzzle, does it have a solution?
- **SAT.** Given a propositional formula (in conjunctive normal form), is it *satisfiable*?

Examples of problems in NP

- **Hamiltonian path problem.** Given a graph G and nodes s and t, is there a path from s to t *which visits each node exactly once*?
- Given a graph G, is there a correct 3-coloring?
- Given a finite list of numbers $x_1, \ldots, x_k$ and a target number t, is there a sublist $x_{i_1}, \ldots, x_{i_k}$ which adds up to t?
- Given an $n \times n$ Sudoku puzzle, does it have a solution?
- **SAT.** Given a propositional formula (in conjunctive normal form), is it *satisfiable*?
- In fact, these problems are all **NP**-*complete* ("hardest").
- The fact that **SAT** is NP-complete is known as the *Cook-Levin Theorem* (1970's).

P vs. NP

- There is no problem in **NP** for which we know for sure that it is not in **P**.

P vs. NP

- There is no problem in **NP** for which we know for sure that it is not in **P**.

$$\mathbf{P} \neq \mathbf{NP}?$$

- There is no problem in **NP** for which we know for sure that it is not in **P**.

$$\mathbf{P} \neq \mathbf{NP}?$$

Describing complexity via logic

Recall Büchi's Theorem:

A language is regular if, and only if, it is definable in monadic second order logic:

$$\mathsf{REG} = \mathsf{MSO}.$$

Describing a language in existential second-order logic

- Consider the language

$$L := \{\langle G\rangle \mid G = (V, E) \text{ admits a correct 3-coloring}\}.$$

- This language can be *described* by an existential second-order sentence:

Describing a language in existential second-order logic

- Consider the language

$$L := \{\langle G \rangle \mid G = (V, E) \text{ admits a correct 3-coloring}\}.$$

- This language can be *described* by an existential second-order sentence:

$$\exists R \exists Y \exists B \big(\forall x [Rx \vee Yx \vee Bx \wedge \forall y [Exy \rightarrow \\ (\neg(Rx \wedge Ry) \wedge \neg(Bx \wedge By) \wedge \neg(Yx \wedge Yy))].$$

ESO vs. MSO

Two main differences when moving from *monadic* to *existential* second-order logic:

1. Second-order quantifiers may range over n-ary predicates, instead of only monadic;

ESO vs. MSO

Two main differences when moving from *monadic* to *existential* second-order logic:

1. Second-order quantifiers may range over n-ary predicates, instead of only monadic;
2. Any *type* of structure is allowed, instead of only words.

ESO vs. MSO

Two main differences when moving from *monadic* to *existential* second-order logic:

1. Second-order quantifiers may range over n-ary predicates, instead of only monadic;
2. Any *type* of structure is allowed, instead of only words.

Formalisms like ESO are used in database query languages like SQL.

Theorem (Fagin, 1973)

Existential second-order logic exactly captures the complexity class **NP***:*

$$ESO = \mathbf{NP}.$$

Proof that ESO $\subseteq$ NP

- Consider an ESO-sentence $\Phi = \exists R_1 \cdots \exists R_k \psi$, where the variable R_i is of arity r_i, and ψ is a first-order τ-sentence.

Proof that ESO $\subseteq$ NP

- Consider an ESO-sentence $\Phi = \exists R_1 \cdots \exists R_k \psi$, where the variable R_i is of arity r_i, and ψ is a first-order τ-sentence.
- Let A be a τ-structure and $n = |\langle A \rangle|$, the size of the encoding of A.

Proof that ESO $\subseteq$ NP

- Consider an ESO-sentence $\Phi = \exists R_1 \cdots \exists R_k \psi$, where the variable R_i is of arity r_i, and ψ is a first-order τ-sentence.
- Let A be a τ-structure and $n = |\langle A \rangle|$, the size of the encoding of A.
- On a structure $(A, R_1, \ldots, R_k)$, it is possible to check in polynomial time (actually: 'logspace') whether the *first-order* sentence ψ is true.

Proof that ESO $\subseteq$ NP

- Consider an ESO-sentence $\Phi = \exists R_1 \cdots \exists R_k \psi$, where the variable R_i is of arity r_i, and ψ is a first-order τ-sentence.
- Let A be a τ-structure and $n = |\langle A \rangle|$, the size of the encoding of A.
- On a structure $(A, R_1, \ldots, R_k)$, it is possible to check in polynomial time (actually: 'logspace') whether the *first-order* sentence ψ is true.
- Let M be the non-deterministic Turing machine which takes $\langle A \rangle$ as input and does the following:

Proof that ESO $\subseteq$ NP

- Consider an ESO-sentence $\Phi = \exists R_1 \cdots \exists R_k \psi$, where the variable R_i is of arity r_i, and ψ is a first-order τ-sentence.
- Let A be a τ-structure and $n = |\langle A \rangle|$, the size of the encoding of A.
- On a structure $(A, R_1, \ldots, R_k)$, it is possible to check in polynomial time (actually: 'logspace') whether the *first-order* sentence ψ is true.
- Let M be the non-deterministic Turing machine which takes $\langle A \rangle$ as input and does the following:
    1. Guess relations $R_1, \ldots, R_k$;

Proof that ESO $\subseteq$ NP

- Consider an ESO-sentence $\Phi = \exists R_1 \cdots \exists R_k \psi$, where the variable R_i is of arity r_i, and ψ is a first-order τ-sentence.
- Let A be a τ-structure and $n = |\langle A \rangle|$, the size of the encoding of A.
- On a structure $(A, R_1, \ldots, R_k)$, it is possible to check in polynomial time (actually: 'logspace') whether the *first-order* sentence ψ is true.
- Let M be the non-deterministic Turing machine which takes $\langle A \rangle$ as input and does the following:
  1. Guess relations $R_1, \ldots, R_k$;
  2. Check whether the guess was correct, if so, accept .

Proof that ESO $\subseteq$ NP

- Consider an ESO-sentence $\Phi = \exists R_1 \cdots \exists R_k \psi$, where the variable R_i is of arity r_i, and ψ is a first-order τ-sentence.
- Let A be a τ-structure and $n = |\langle A \rangle|$, the size of the encoding of A.
- On a structure $(A, R_1, \ldots, R_k)$, it is possible to check in polynomial time (actually: 'logspace') whether the *first-order* sentence ψ is true.
- Let M be the non-deterministic Turing machine which takes $\langle A \rangle$ as input and does the following:
  1. Guess relations $R_1, \ldots, R_k$;
  2. Check whether the guess was correct, if so, accept .
- Making a guess can be done in $n_1^r \cdot \cdots \cdot n_k^r$ steps (Why?)

Proof that ESO $\subseteq$ NP

- Consider an ESO-sentence $\Phi = \exists R_1 \cdots \exists R_k \psi$, where the variable R_i is of arity r_i, and ψ is a first-order τ-sentence.
- Let A be a τ-structure and $n = |\langle A \rangle|$, the size of the encoding of A.
- On a structure $(A, R_1, \ldots, R_k)$, it is possible to check in polynomial time (actually: 'logspace') whether the *first-order* sentence ψ is true.
- Let M be the non-deterministic Turing machine which takes $\langle A \rangle$ as input and does the following:
  1. Guess relations $R_1, \ldots, R_k$;
  2. Check whether the guess was correct, if so, accept .
- Making a guess can be done in $n_1^r \cdot \cdots \cdot n_k^r$ steps (Why?)
- Thus, M decides the ESO-sentence Φ in polynomial time.

Proof idea: NP $\subseteq$ ESO

- Let M be a non-deterministic Turing machine which halts within n^k steps on all computations on all inputs of size n.

Proof idea: NP $\subseteq$ ESO

- Let M be a non-deterministic Turing machine which halts within n^k steps on all computations on all inputs of size n.
- In particular, at most n^k tape cells are used by M.

Proof idea: NP $\subseteq$ ESO

- Let M be a non-deterministic Turing machine which halts within n^k steps on all computations on all inputs of size n.
- In particular, at most n^k tape cells are used by M.
- Introduce, for each tape symbol $\gamma \in \Gamma$, a $2k$-ary predicates $C_\gamma(\overline{x}, \overline{y})$, with intended meaning "tape cell number $\overline{x}$ has letter γ at moment $\overline{y}$".

Proof idea: NP $\subseteq$ ESO

- Let M be a non-deterministic Turing machine which halts within n^k steps on all computations on all inputs of size n.
- In particular, at most n^k tape cells are used by M.
- Introduce, for each tape symbol $\gamma \in \Gamma$, a $2k$-ary predicates $C_\gamma(\overline{x}, \overline{y})$, with intended meaning "tape cell number $\overline{x}$ has letter γ at moment $\overline{y}$".
- There is an ESO sentence of the form

$$\exists C_1 \cdots \exists C_n \psi$$

  which expresses that M reaches an accepting state within n^k steps.
- (Many details left to check.)

A logic for P?

- **Immerman-Vardi Theorem.** First order logic with induction (in the form of *least fixed points*) exactly captures **P** *on ordered structures*.

A logic for P?

- **Immerman-Vardi Theorem.** First order logic with induction (in the form of *least fixed points*) exactly captures **P** *on ordered structures*.
- The problem of finding a logic that captures **P** is open.
- $\rightarrow$ M. Grohe, “The quest for a logic capturing PTIME”, LICS (2008).

Summary

- Definition of Turing machine, first complete model of computation (if you believe the Church-Turing thesis).
- Undecidability of the Halting problem.
- Main time complexity classes: P and NP.
- ESO captures NP.

Summary

- Definition of Turing machine, first complete model of computation (if you believe the Church-Turing thesis).
- Undecidability of the Halting problem.
- Main time complexity classes: P and NP.
- ESO captures NP.

Homework

- **Reminder!** Submit your top-3 paper topic preferences **today** before 5pm.
- **Reminder!** On Friday before 5pm, submit:
  1. Your research question;
  2. Name of a preferred peer reviewer;
  3. Preference for presenting early/middle/late.
- **Self-study**: review this week's material, ask questions if needed.

Reference

M. Sipser, *Introduction to the Theory of Computation*, 3rd edition, Cengage Learning (2013).