Reduction via Computation Histories

Another important technique for proving that something is undecidable.

- useful for testing for existence of something
- basis for undecidability of Hilbert's tenth.

Review: Configurations

Configuration:

$$1011q_7 0111$$

means:

- state is q_7
- LHS of tape is 1011
- RHS of tape is 0111
- head is on RHS 0

Configurations

- configuration uaq_ibv *yields* uq_jacv if $\delta(q_i, b) = (q_j, c, L)$
- Of course, uaq_ibv yields $uacq_jv$ if $\delta(q_i, b) = (q_j, c, R)$
- Special case: q_ibv yields q_jcv if $\delta(q_i, b) = (q_j, cL)$.
- Configuration wq_i is the same as $wq_i\sqcup$.

Computation Histories

Let M be a TM and w an input string.

An *accepting computation history* for M on w is a sequence $C_1, C_2, \ldots, C_\ell$, where

- C_1 is the starting configuration of M on w
- C_ℓ is an accepting configuration of M,
- each C_i legally yields C_{i+1}.

An *rejecting computation history* for M on w is the same, except

- C_ℓ is a rejecting configuration of M,

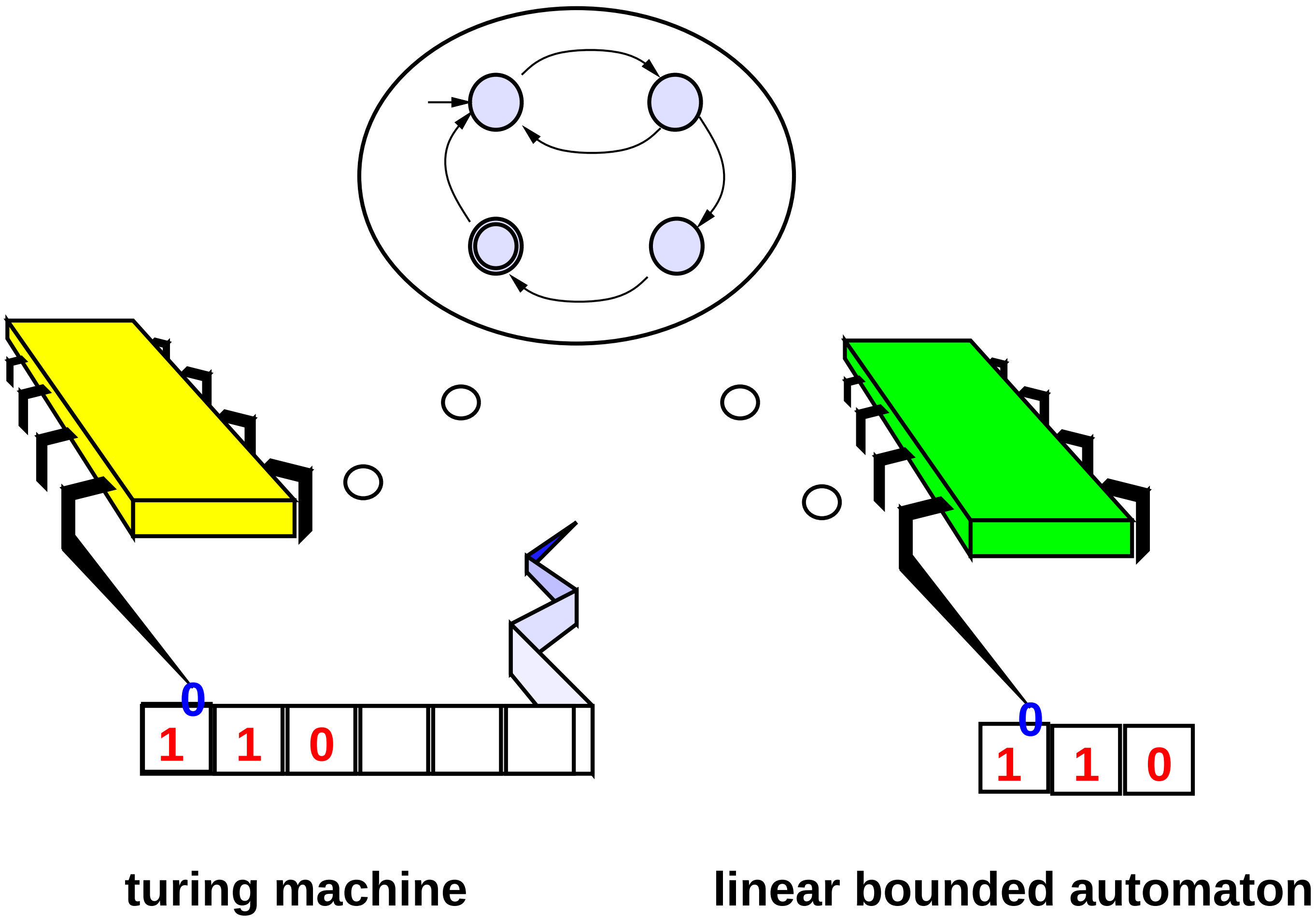
Remarks

- Computation sequences are finite.
- If M does not halt on w, no accepting or rejecting computation history exists.
- Useful for both deterministic (one history) and non-deterministic (many histories) TMs.

Linear Bounded Automata

A linear bounded automaton is

- a restricted form of TM.
- not allowed to move of portion of tape containing input
- ignores attempts to move head beyond input
- size of input determines size of memory



Linear Bounded Automata

Question: Whence the name?

Answer: Using a tape alphabet larger than the input alphabet increases memory by a constant factor.

Linear Bounded Automata

Believe it or not, LBAs are quite powerful.

The deciders for

- A_{DFA} (does DFA accept?)
- A_{CFG} (is string in CFG?)
- E_{DFA} (is DFA trivial?)
- E_{CFG} (is CFG empty?)

are all LBAs.

Every CFL can be decided by a LBA.

Not easy to find a decidable language that cannot be decided by an LBA.

A Language

Define

$$A_{\mathsf{LBA}} = \{\langle M, w\rangle | M \text{ is an LBA that accepts } w\}$$

Question: Is A_{LBA} decidable?

A Lemma

Lemma: Let M be a LBA with

- q states
- g symbols in tape alphabet

On an input of size n, there are exactly qng^n distinct configurations. Configuration involves:

- control state (q possibilities)
- head position (n possibilities)
- tape contents (g^n possibilities)

Theorem

Theorem: A_{LBA} is decidable.

Idea:

- Simulate M on w.
- But what do we do if M loops?
- Must detect looping and reject.
- M loops if and only if it repeats a configuration.
- Why? Is this also true of TMs?
- M must repeat a configuration if it runs long enough

Theorem

Theorem: A_{LBA} is decidable.

On input $\langle M, w\rangle$, where M is an LBA and w is a string

1. Simulate M on w for qng^n steps, or until it halts.

2. If M has halted, *accept* if it accepted, and *reject* if it rejected, and *reject* if M has not halted.

More LBAs

Here is a related problem.

$$E_{\text{LBA}} = \{\langle M\rangle | M \text{ is an LBA and } L(M) = \emptyset\}$$

Question: Is E_{LBA} decidable?

More LBAs

$$E_{\textsf{LBA}} = \{\langle M \rangle | M \text{ is an LBA and } L(M) = \emptyset\}$$

Theorem: $E_{\textsf{LBA}}$ is undecidable.

Proof by reduction using computation histories.

More LBAs

Surprisingly though, LBAs do have undecidable problems.

$$E_{\mathsf{LBA}} = \{\langle M\rangle | M \text{ is an LBA and } L(M) = \emptyset\}$$

Theorem: E_{LBA} is undecidable.

Proof by reduction from A_{TM}.

If E_{LBA} were decidable, then A_{TM} would also be.

Question: Suppose that E_{LBA} is decidable. How can we use this supposition to decide A_{TM}?

More LBAs

We will construct an LBA B.

- $L(B)$ is all accepting computation histories for M on b
- M accepts w iff $L(B) \neq \emptyset$.
- If we can decide whether $L(B) = \emptyset$ then we can decide whether M accepts w.

More LBAs

It is *not* enough to show that B exists.

We must show that a TM can construct $\langle B \rangle$ from $\langle M, w \rangle$.

Assume an accepting computation history is presented as a string:

$$\# \underbrace{\qquad\qquad}_{C_1} \# \underbrace{\qquad\qquad}_{C_2} \# \underbrace{\qquad\qquad}_{C_3} \# \cdots \# \underbrace{\qquad\qquad}_{C_\ell} \#$$

descriptions of configurations separated by # delimiters.

The LBA

The LBA B works as follows.

On input x, B:

- breaks x according to the # delimiters
- constructs strings $C_1, C_2, \ldots, C_\ell$.
- then checks:
  - C_1 is the start configuration
  - each C_{i+1} legally follows from C_i
  - C_ℓ is an accepting configuration

The LBA

Checking that C_1 is the start configuration

$$q_0 w_1 w_2 \cdots w_n$$

is easy, because the string is "built in" to B.

Checking that C_ℓ is an accepting configuration is easy, because C_ℓ must include the accepting state q_a.

The hard part is checking that each C_{i+1} legally follows from C_i.

The Hard Part

Verifying that each C_{i+1} legally follows from C_i.

- C_i and C_{i+1} identical except for positions under and adjacent to head.
- These positions updated according to transition function.

Do this by

- zig-zagging between corresponding positions of C_i and C_{i+1}.
- use "dots" on tape to mark current position

Important!

The LBA B is not constructed to run on some input!

(This is a common source of confusion.)

We construct B only to feed it to the decider of E_{LBA} which we assume to exist.

Once this decider returns its answer, we invert it to decide whether M accepts w.

The Proof

Suppose TM R decides E_{LBA}.

Define TM S that decides A_{TM}:

On input $\langle M, w\rangle$

1. Construct LBA B from M and w as described above.

2. Run R on $\langle B\rangle$.

3. if R rejects, *accept*; if R accepts, *reject*.

If R accepts $\langle B\rangle$

- M has not accepting computation history on w
- M does not accept w
- S rejects $\langle M, w\rangle$

If R rejects $\langle B\rangle$

- the language of B is non-empty
- the only string B can accept is an accepting computation of M on w
- thus M accepts w
- S accepts $\langle M, w\rangle$.

So S decides A_{TM}, a contradiction.

Another Use for Computation Histories

We have already seen an algorithm to check whether a context-free grammar is empty.

On input $\langle G \rangle$ where G is a CFG:

1. Mark all terminal symbols in G.

2. Repeat until no new variables become marked:

3. Mark any A where

$$A \to U_1 U_2 \ldots U_k$$

and each U_i has already been marked.

4. If start symbol marked, *accept*, otherwise *reject*.

Another Use for Computation Histories

We have already seen an algorithm to check whether a context-free grammar is empty.

Question: What about the complementary question: when does a CFG generate *all* strings?

Does a CFG Generate All Strings?

$$\text{All}_{\text{CFG}} = \{\langle G \rangle | G \text{ is a CFL and } L(G) = \Sigma^*\}$$

Theorem: All_{CFG} is undecidable.

Proof by contradition and reduction.

- Assume All_{CFG} is decidable.
- show that A_{TM} is then decidable.
- for TM M and input w construct CFG G
- G generates all strings that are not accepting computation histories for M on w
- if M does not accept w, G generates all strings
- if M does accept w, G does not generate the accepting computation history.

Does a CFG Generate All Strings?

An accepting computation appears as
$\#C_1\#C_2\#\ldots\#C_\ell\#$, where

- C_1 is the starting configuration of M on w
- C_ℓ is an accepting configuration of M,
- each C_i legally yields C_{i+1}.

A string fails if it fails any of these tests.

Does a CFG Generate All Strings?

Convert G into a PDA D.

D non-deterministically "guesses" which condition to check.

- Is C_1 the starting configuration of M on w?
- Is C_ℓ an accepting configuration of M?
- Does C_i legally yield C_{i+1}?

Last condition is the tricky one.

Does a CFG Generate All Strings?

- Does C_i legally yield C_{i+1}?

Idea:

- scan input until it nondeterministically decides it has reached C_i.
- Push C_i onto the stack.
- scan C_{i+1} and pop matching symbols of C_i
  - check that C_i and C_{i+1} match, except . . .
  - around the head position,
  - where difference dictated by transition function for M

Does a CFG Generate All Strings?

Problem: When D pops C_i from stack, C_i is in reverse order.

Solution: Write the accepting computation history so that every other configuation is in reverse order.

$$\# \underbrace{\quad\longrightarrow\quad}_{C_1} \# \underbrace{\quad\longleftarrow\quad}_{C_2} \# \underbrace{\quad\longrightarrow\quad}_{C_3} \# \underbrace{\quad\longleftarrow\quad}_{C_4} \# \cdots \# \underbrace{\qquad\qquad\qquad}_{C_\ell} \#$$

This completes the proof.