Reductions via computation histories

Teodor Rus

rus@cs.uiowa.edu

The University of Iowa, Department of Computer Science

Computation history

The computation history for a TM on an input is the sequence of configurations that the machine goes through as it processes the input

Note: the computation history of TM M is a complete record of the computation performed by M.

Accepting and rejecting histories

Let M be a TM and w an input string. We distinguish two kind of histories:

- Accepting computation history;
- Rejection computation history;

Accepting computation history

An *accepting computation history* for M on w is a sequence of configurations $C_1, C_2, \ldots, C_k$ where:

1. C_1 is the start configuration of M on w;
2. C_k is an accepting configuration of M;
3. C_{i+1} legally follows from C_i according to the transition function of M.

Rejecting computation history

A *rejecting computation history* for M on w is a sequence of configurations $C_1, C_2, \ldots, C_k$ where:

1. C_1 is the start configuration of M on w;
2. C_k is a rejecting configuration of M;
3. C_{i+1} legally follows from C_i according to the transition function of M.

Facts

1. Computation histories are finite sequences;
2. If M does not halt on w, no accepting or rejecting computation history exists for M on w;
3. Deterministic machines have at most one computation history on a given input;
4. Nondeterministic machines may have many computation histories on a single input, corresponding to various computation branches.

Linear Bounded Automata

A LBA is a restricted type of TM wherein the tape head isn't permitted to move off the portion of the tape containing the input.

Note: if the machine tries to move its head off either end of the input, the head stays where it is, in the same way that the head will not move off the left-hand end of the ordinary TM's tape.

Schematically, Figure 1

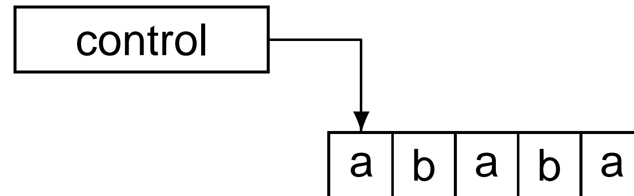

Figure 1: Schematic of a LBA

Facts

1. An LBA is a TM with a limited memory, as shown in Figure 1;
2. An LBA can only solve problems requiring memory that can fit within the tape used as input;
3. Using a tape alphabet larger than input alphabet allows the available memory to be increased up to a constant factor;
4. For an input of length n, the memory amount available is linear in n (hence, the name of the model).

Observations

- Despite their memory constraint, LBA are quite powerful.
- Deciders for A_{DFA}, A_{CFG}, E_{DFA}, E_{CFG} all are LBAs.
- Every CFL can be decided by an LBA.

Note: constructing a decidable language that cannot be decided by an LBA is an intractable problem. That is, such problems are solvable but their solutions require so much time or space that they can't be used in practice.

A solvable problem

Problem:
For a given LBA and string w does LBA accept w?

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

Theorem 5.9:

The language A_{LBA} is decidable.

Lemma 5.8

Let M be an LBA with q states and g symbols in its tape alphabet. There are exactly qng^n distinct configurations of M for a tape of length n.

Proof:

- A configuration of M is a tuple $(state, headPosition, tapeContents)$;
- M has q states;
- Length of input is n, so the head can be in n positions;
- Only g^n possible strings can appear on the tape.

Hence, qng^n is the number of different configurations of M

Proof of theorem 5.9

Proof idea: to decide whether an LBA M accepts w one can perform as follows:

- Simulate M on w;
- During the simulation, if M halts and accepts or reject, accepts or rejects accordingly;
- If simulation does not halt, detect the loop so that one can halt and reject.

Loop detection

- As it computes, M goes from configuration to configuration;
- If M ever repeats a configuration, it would go on to repeat this configuration over and over again, and thus looping;
- Since M is an LBA, it has only qng^n different configurations. Hence, if it performs a number of steps larger than qng^n and it did not halted, then M repeats a configuration and thus it loops.

Proof

The algorithm that decides A_{LBA} is:

L = "On input $\langle M, w \rangle$, M and LBA, w a string:

1. Simulate M on w for qng^n steps or until it halts;
2. If M halted, *accept* if M has accepted, and *reject* if M rejected; If M has not halted, *reject*."

Observations

- Theorem 5.9 shows that LBA and TM differ in one essential way: for LBAs acceptance problem is decidable while for TMs it is not.
- Certain other problems involving LBAs remain undecidable.
- **Example:** emptiness problem
  $E_{LBA} = \{\langle M \rangle | M \in LBA \wedge L(M) = \emptyset\}$
  is undecidable.

Theorem 5.10

E_{LBA} is undecidable.

Proof idea:

- By reduction from A_{TM}, show that:
  if E_{LBA} is decidable then so is A_{TM}.

Using E_{LBA} decidability

- For a TM M and input w determine whether M accept w by constructing an LBA B and testing if $L(B)$ is empty.
- Language recognized by B consists of all accepting computation histories of M on w. If M accepts, $L(B)$ contains one string and so $L(B) \neq \emptyset$; if M does not accept, $L(B) = \emptyset$.

Hence, if we can detect whether $L(B)$ is empty we can determine whether M accept w!

Constructing B form M and w

- We need to show how a TM can obtain a description of B from M and w;
- Construct B to accept its input x, if x is an accepting computation history for M on w.

Note: the accepting configurations of B are represented as single strings with configurations separated by $\#$, i.e.:

$$x = \#C_1\#C_2\#\ldots\#C_k\#$$

LBA B works as follows

- Breaks up x determining configurations $C_1, C_2, \ldots, C_k$;
- Check the conditions:
  1. C_1 is the start configuration for M on w, i.e. $C_1 = q_0 w_1 w_2 \ldots w_n$, q_0 the start state of M;
  2. Each C_{i+1} legally follows from C_i, i.e., verify that C_i and C_{i+1} are identical except for the positions under and adjacent to the head in C_i. These positions must be updated according to the transition function of M;
  3. C_k is an accepting configuration for M on w, i.e., C_k contains q_{accept} of M.

Facts

1. Item (2) above can be verified directly from the transition function δ. This is shown in Figure 2 for $\delta(q_3, a) = (q_5, x, R)$.
2. Item (2) above can be performed by zig-zagging between corresponding positions of C_i, C_{i+1}.

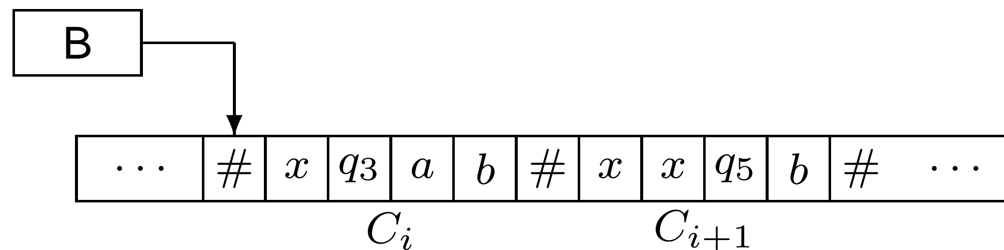

Figure 2: Checking computation history

Proof

Suppose that TM R decides E_{LBA}. Construct TM S that decides A_{TM} as follows:

S = "On input $\langle M, w\rangle$, M a TM and w a string:

1. Construct LBA $B_{\langle M,w\rangle}$ from M and w as described above;
2. Run R on input $\langle B_{\langle M,w\rangle}\rangle$;
3. If R rejects, *accept*; if R accepts, *reject*."

Facts

1. If R accepts $\langle B_{\langle M,w\rangle}\rangle$ then $L(B_{\langle M,w\rangle}) = \emptyset$, thus M has no accepting computation on w, i.e., M does not accept w. Consequently S rejects $\langle M, w\rangle$.
2. If R rejects $\langle B_{\langle M,w\rangle}\rangle$, $L(B_{\langle M,w\rangle}) \neq \emptyset$. Since the only string $B_{\langle M,w\rangle}$ can accept is an accepting computation history for M on w it means that M accepts w. Consequently S accepts $\langle M, w\rangle$.

This is a contradiction, so R cannot exist.

Theorem 5.13

Problem:
Determine whether a CFG generates all possible strings over its terminal alphabet Σ.

Language: the language
$ALL_{CFG} = \{\langle G \rangle | G\ CFG \wedge L(G) = \Sigma^*\}$

Theorem: the language ALL_{CFG} is undecidable.

Proof idea: by the reduction of A_{TM} to ALL_{CFG}.
Assume that ALL_{CFG} is decidable and show that A_{TM} is then decidable too.

Proof idea, continuation

Using the decidability of ALL_{CFG}, one can devise the following decision procedure for A_{TM}:

- For a TM M and input w construct CFG $G_{\langle M,w\rangle}$ that generates all strings over the extended alphabet of M iff M does not accept w.
- If M does accept w, $G_{\langle M,w\rangle}$ does not generate a particular string which is the accepting computation history for M on w.

Note: Since accepting and non-accepting histories are finite strings and $G_{\langle M,w\rangle}$ can generate all strings over its alphabet, this is a reasonable itdea.

Deciding A_{TM}

Assume that R decides the ALL_{CFG}. The TM S that decides A_{TM} follows:

S = "On input $\langle M, w\rangle$:

1. Use $\langle M, w\rangle$ to construct $G_{\langle M,w\rangle}$ such that:

$$L(G_{\langle M,w\rangle}) = \begin{cases} \Sigma^* & \text{if } M \text{ does not accept } w \\ \Sigma^* \setminus \{x\} & x \text{ accepting history of } M \text{ on } w\text{, otherwise.} \end{cases} \quad (1)$$

2. Run R on $\langle G_{\langle M,w\rangle}$. IF R accepts, *reject*; if R rejects, *accept*."

Note: for technical reasons one can construct the $PDA_{\langle M,w\rangle}$ rather than $G_{\langle M,w\rangle}$ and then convert it to $G_{\langle M,w\rangle}$.

Strategy

- An accepting computation history for M on w has the form $\#C_1\#C_2\#\ldots\#C_k\#$.
- Hence, G generates all strings that:
  1. do not start with C_1;
  2. do not end with an accepting configuration;
  3. for some i, C_i does not properly yield C_{i+1} under the rules of M.
- Construct the equivalent $PDA_{\langle M,w\rangle}$ and then convert it to $G_{\langle M,w\rangle}$.

Actions performed by $PDA_{\langle M,w \rangle}$

$PDA_{\langle M,w \rangle}$ starts by nondeterministically branching to guess which of the preceding three conditions to check:

1. Checks on whether the beginning of the input string is C_1 and accepts if it isn't;
2. Checks on whether the input string ends with a configuration that contains q_{accept} and accepts if it isn't;
3. Checks on whether some configuration C_i does not properly yield C_{i+1}.

Checking configuration match

1. Scan the input until it nondeterministically decides that it has come to C_i;
2. Pushes elements of C_i on the stack until it finds its end specified by $\#$;
3. Pop the stack and compare C_i with C_{i+1} which suppose to match except around the head position;
4. Accept if a mismatch or an improper update is discovered.

Problem

When $PDA_{\langle M,w\rangle}$ pops C_i off the stack it is in reveres order and not suitable for comparison with C_{i+1}.

Remedy: write every other configuration of the input in reverse order. That is, configurations in the computation history are written as: $\#C_1\#C_2^{\mathcal{R}}\#\ldots\#C_{2k-1}\#C_{2k}^{\mathcal{R}}\#\ldots$

Constructing $G_{\langle M,w\rangle}$

- Construct the equivalent PDA $D_{\langle M,w\rangle}$ and then convert it to $G_{\langle M,w\rangle}$.
- To test that C_i yields properly C_{i+1} using the stack reverse C_{i+1}, i.e., the accepting configurations for $G_{\langle M,w\rangle}$ are:
  $\#C_1\#C_2^R\#C_3\#C_4^R\#\ldots\#C_k\#$;
- The PDA $D_{\langle M,w\rangle}$ is able to push a configuration so that when it is popped the order is suitable for comparison with the next configuration.

Application

Use the theorem 5.13 to show that the language
$EQ_{CFG} = \{\langle G, H \rangle |\ G$ and H are CFG and $L(G) = L(H)\}$
is undecidable

Solution: Assume EQ_{CFG} is decidable. Construct a decider M for $ALL_{CFG} = \{\langle G \rangle | G\ CFG, L(G) = \Sigma^*\}$ by:

M= "On input $\langle G \rangle$

1. Construct CFG H such that $L(H) = \Sigma^*$;
2. Run the decider for EQ_{CFG} on $\langle G, H \rangle$;
3. If it accepts, *accept*; if it rejects, *reject*."

Facts

1. M decides EQ_{CFG} assuming that a decider for ALL_{CFG} exists;
2. Since we know that ALL_{CFG} is undecidable, a contradiction result.