CS420

Introduction to the Theory of Computation

Lecture 21: Acceptance, emptiness and equality tests

Tiago Cogumbreiro

Today we will learn...

- Acceptance tests
- Emptiness tests
- Equality tests

Section 4.1

Why do we need Turing Machines?

Why do we need Turing Machines?

- Turing Machines are Computers!
- Turing Machines are Programs!
- We will study mathematically the limits of what is possible

Turing Recognizable

Decision problems

Disclaimer: Henceforth, when we say a program, we are restricting ourselves to **decision problems**.

- Example: DFA accepting/rejecting a string
- Example: PDA accepting/rejecting a string
- Example: functions that return a boolean
- Example: programs run until they print yes/no
- Example: computers that run until they turn on a red/green light

Turing Recognizable

Recognized language of TM

Notation $L(M)$ is the set of strings that M **accepts**, its recognized language.

Definition 3.5: Turing-Recognizable language

A language is Turing-recognizable if some TM recognizes it.

What is a decidable Turing Machine?

What is a decidable Turing Machine?

For all inputs: REJECT $\vee$ ACCEPT

(No loops!)

Turing Decidable

A TM that for all inputs either accepts or rejects, and does not loop forever.

Definition 3.6: Decidable language

A language is decidable if some Turing-decidable machine recognizes it.

How do I know if a Turing Machine is decidable?

Turing Decidable

A TM that for all inputs either accepts or rejects, and does not loop forever.

Definition 3.6: Decidable language

A language is decidable if some Turing-decidable machine recognizes it.

How do I know if a Turing Machine is decidable?

We prove it!

Recap

- A decidable TM REJECT/ACCEPT any input
- A decidable ***language*** is one that is recognized by a decidable TM

Decidable algorithms

- Algorithms are equivalent to TMs
- An algorithm that returns REJECT/ACCEPT (eg, a boolean) for all inputs
- A decidable algorithm is always **total**†
  - A total function is defined (ie, returns a value) for all inputs. Looping implies that no value is being returned.
  - In some programming languages (eg, Coq, Agda, Idris) you can write total functions (mechanically proven by the language).

Decidable algorithms

- Algorithms are equivalent to TMs
- An algorithm that returns REJECT/ACCEPT (eg, a boolean) for all inputs
- A decidable algorithm is always **total**†
    - A total function is defined (ie, returns a value) for all inputs. Looping implies that no value is being returned.
    - In some programming languages (eg, Coq, Agda, Idris) you can write total functions (mechanically proven by the language).

Proving decidability

requires a proof that the function **terminates**!

(along with **correctness**)

Decidable algorithms

Example

Our algorithm that implements DFA acceptance

```
def dfa_accepts(dfa, inputs):
  st = dfa.start
  for i in inputs:
    st = dfa.state_transition(st, i)
  return st in dfa.accepted_states
```

Termination proof. The function loops over `len(inputs)`-steps (which is a natural number) and then returns a boolean.

A_X: Acceptance tests

Decidable algorithms on acceptance

(Will X accept this input?)

A_{DFA}: DFA Acceptance

The language of all DFAs that accept a given string w

$$A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

Theorem 4.1. A_{DFA} is a decidable language.

Proof.

A_{DFA}: DFA Acceptance

The language of all DFAs that accept a given string w

$$A_{\mathsf{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$$

Theorem 4.1. A_{DFA} is a decidable language.

Proof.

We already showed that function `dfa_accepts` is correct and that it terminates, thus there exists a TM that encodes it and that TM is decidable.

A_{NFA}: NFA Acceptance

The language of all NFAs that accept a given string w

$$A_{\mathsf{NFA}} = \{\langle N, w\rangle \mid N \text{ is an NFA that accepts input string } w\}$$

Theorem 4.2. A_{NFA} is a decidable language.

Proof.

A_{NFA}: NFA Acceptance

> The language of all NFAs that accept a given string w

$$A_{\mathsf{NFA}} = \{\langle N, w\rangle \mid N \text{ is an NFA that accepts input string } w\}$$

Theorem 4.2. A_{NFA} is a decidable language.

Proof.

If we assume that the function that converts a DFA into an NFA is total, then the following algorithm is total and correct:

```
def nfa_accepts(nfa, input):
  return dfa_accepts(nfa_to_dfa(nfa), input)
```

And therefore, the TM that implements it is decidable, and so is A_{NFA}.

$A_{\textsf{REX}}$: Regular Expression Acceptance

The language of all regex that accept a given string w

$$A_{\textsf{REX}} = \{\langle R, w\rangle \mid R \text{ is an regular expression that accepts input string } w\}$$

Theorem 4.3. $A_{\textsf{REX}}$ is a decidable language.

Proof.

$A_{\textsf{REX}}$: Regular Expression Acceptance

The language of all regex that accept a given string w

$$A_{\textsf{REX}} = \{\langle R, w\rangle \mid R \text{ is an regular expression that accepts input string } w\}$$

Theorem 4.3. $A_{\textsf{REX}}$ is a decidable language.

Proof.

Similarly, if we assume that the function that converts a regular expression into an NFA is total, then the following algorithm is total and correct:

```
def rex_accepts(rex, input):
  return nfa_accepts(rex_to_nfa(nfa), input)
```

And therefore, the TM that implements it is decidable, and so is $A_{\textsf{REX}}$.

A_{CFG}: Context-Free-Grammar Acceptance

The language of all context-free grammars that accept a given string w

$$A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid G \text{ is a context-free grammar that accepts input string } w\}$$

Theorem 4.7. A_{CFG} is a decidable language.

Proof.

A_{CFG}: Context-Free-Grammar Acceptance

The language of all context-free grammars that accept a given string w

$$A_{\mathsf{CFG}} = \{\langle G, w\rangle \mid G \text{ is a context-free grammar that accepts input string } w\}$$

Theorem 4.7. A_{CFG} is a decidable language.

Proof.

We studied the CYK algorithm that is decidable and given a CFG can test the acceptance of a CFG. Additionally, we also studied a decidable acceptance algorithm for PDAs, so we could convert the CFG to a PDA (which is a total function).

E_X: Emptiness tests

Decidable algorithms on emptiness

(Is the language of X empty?)

E_{DFA}: DFA Emptiness

The set of DFAs that recognize an empty language.

$$E_{\mathsf{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$$

Theorem 4.4. E_{DFA} is a decidable language.

Proof.

E_{DFA}: DFA Emptiness

The set of DFAs that recognize an empty language.

$$E_{\mathsf{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$$

Theorem 4.4. E_{DFA} is a decidable language.

Proof. (Note: we do not argue the correctness, although technically we should.)

1. Mark the initial state of DFA A as to-visit and visited.
2. While there are states to visit: Unmark one to-visit and mark all transitions that have not been visited as to-visit and as visited
3. Accept when zero visited states are accepted, otherwise reject.

Totality argument: The loop terminates because at each step the set of potential states to visit is bounded by the total number of states and that number decreases by at least one at each iteration step.

E_{DFA}: DFA Emptiness

Python implementation

```python
def is_empty(dfa):
  to_visit = [dfa.start_state]
  visited = set(to_visit)
  while len(to_visit) > 0:
    node = to_visit.pop()
    for i in dfa.alphabet:
      st = dfa.state_transition(st,i)
      if st not in visited:
        to_visit.append(st)
        visited.add(st)
  for st in visited:
    if st in dfa.accepted_states:
      return True
  return False
```

E_{CFG}: CFG Emptiness

The set of CFGs that recognize an empty language.

$$E_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$$

Theorem 4.8. E_{CFG} is a decidable language.

Proof.

E_{CFG}: CFG Emptiness

The set of CFGs that recognize an empty language.

$$E_{\mathsf{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$$

Theorem 4.8. E_{CFG} is a decidable language.

Proof.

1. Mark all terminal symbols of G
2. Until no new variables get marked
   - Mark any variable where G has a rule $G \to A_1 \ldots A_n$ and each A_i has been marked.
3. If the start variable is marked reject, otherwise accept.

Totality argument: The set of unmarked variables is bounded by the set of all variables and terminals. At each iteration the set of unmarked variables increases until it terminates.

EQ_X: Equality tests

Decidable algorithms on equality

(Can we **always** test if two elements of X are equal?)

EQ_{DFA}: DFA Equality

> The set of DFAs that are equal to one another.

$$EQ_{\mathsf{DFA}} = \{\langle A, B\rangle \mid A, B \text{ are DFAs and } L(A) = L(B)\}$$

Theorem 4.5. EQ_{DFA} is a decidable language.

Proof.

Theorem 4.5. EQ_{DFA} is a decidable language.

Proof.

Let the symmetric difference be defined as:

$$A \triangle B = (A - B) \cup (B - A)$$

1. It is easy to see that $L(A) = L(B)$ if, and only if, $L(A) \triangle L(B) = \emptyset$.
2. $A \triangle B$ is closed under the set of regular languages†
3. The algorithm then becomes testing the emptiness of the automaton $A \triangle B$, which we know to be decidable.*

---

†: The set difference and the union are closed under the set of regular languages.

*: The automaton $A \triangle B$ can be trivially defined as automaton-operations.

EQ_{CFG}: CFG Equality

EQ_{CFG}: CFG Equality

Undecidable

- We know that E_{CFG} is decidable
- Why not use the symmetric difference?

EQ_{CFG}: CFG Equality

Undecidable

- We know that E_{CFG} is decidable
- Why not use the symmetric difference?
- Set difference is **not** closed with CFGs

Counter-example:

- $A = \{a^x b^y c^z \mid x = y\}$ is CF
- $B = \{a^x b^y c^z \mid x = z\}$ is CF
- However, $A \cap B = \{a^x b^y c^z \mid x = y \land x = z\}$ is not CF!