Lecture 11: Encodings of Other Objects

Ryan Bernstein

1 Introductory Remarks

- Assignment 3 is now available online. It's due a week from Thursday (May 19th)

1.1 Assignment 2 Solutions

1.1.1 Assignment 2 Remarks

- Closure properties are properties of a set *and* an operation. For instance, we say that the regular languages are closed under complement because the complement of any regular language is also regular. It makes no sense to say that a language class — or worse, an individual language — is "closed" without specifying which operation it is closed under.

1.2 Recapitulation

Last Thursday — in a lecture that some might characterize as a "joyless slog" — we discussed variants of Turing machines, and showed that these variants did not make Turing machines more powerful. These variants included:

- Turing machines with the capability to execute a transition without moving the tape head
- Turing machines with multiple tapes
- Nondeterministic Turing machines
- Pushdown automata with two stacks instead of one

To prove equivalence, we provided methods for simulating each type of machine on a standard Turing machine and vice versa.

2 Encodings of Other Objects

During the last lecture, we began to generalize the Turing machines that we've been constructing. Since we were simulating other machines, we couldn't describe the exact movements of the tape head, since we didn't know the details of the languages that we were trying to decide or recognize. All we knew was that we were trying to simulate some other machine.

We'll now take that idea a step further. Every machine that we've seen so far has been composed of finite sets. Even though a Turing machine has an infinite number of possible computational contexts thanks to its infinite tape, we have a formal definition $(Q, \Sigma, \Gamma, \delta, q_{start}, q_{accept}, q_{reject})$ which is composed entirely of finite elements.

We learned in CS 201 that the binary alphabet $\{0, 1\}$ is capable of representing anything that can be represented using finite precision. At the time, it seemed amazing that images, video, programs, and more could be represented using only the characters zero and one, but there was another aspect to this that we took for granted: if anything that can be represented using finite precision can be represented using a string of zeros and ones, this necessarily implies that anything representable with finite precision can be represented *as a string*.

Since all of the machines that we've discussed so far have had formal definitions composed entirely of finite members, *all* of them can be represented using strings. This means in turn that all of them can be passed to any other machine as input.

How do we construct machines that take arbitrary encodings of other objects as input? We know that it's possible to represent these objects as strings, but we have no idea how this particular encoding was done.

This requires us to move to a higher level of abstraction. No longer can we describe our operations on a character-by-character basis. Instead, we treat the formal description of these structures like interfaces in programming. We don't know how these objects were implemented, but we do know what they're capable of. Since Turing machines are representations of algorithms, we can safely use any algorithmic operations on the object's constituent parts.

To represent some unknown encoding of an object, we usually write the name of that object between left and right angle brackets.

Example 1 Show that $A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}$ is Turing-decidable.

As the "interface" of a graph, we'll use the definition that a graph is a tuple (V, E), where:

- V is a set of vertices
- E is a set of edges connecting those vertices.

At a high level of abstraction, we can construct a decider for A as follows: $M =$ "On input $\langle G \rangle$:

1. Pick a vertex at random and mark it
2. Repeat until no new vertices are marked:
    - Mark any vertex connected to a vertex that is already marked
3. Iterate over V. If any vertices are left unmarked, Reject
4. Accept

How do we "mark" a vertex? We don't know, since we don't even know how these vertices are encoded. We *do* know that it should be algorithmically possible, though. Maybe we copy the representation of each marked node to another tape, or maybe we alter the representation directly on the input tape somehow. At this level of abstraction, the details don't matter.

Example 2 Let $A_{DFA} = \{\langle D, w\rangle \mid D \text{ is a DFA and } D \text{ accepts } w\}$. Show that A_{DFA} is decidable.

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

1. Simulate D on w. If D ends in an accepting state, ACCEPT.
2. REJECT"

3 Subroutines

We're going to look at a lot of examples today. While it may seem somewhat overwhelming, these examples serve an additional purpose. We say that Turing machines are capable of doing anything algorithmically possible. At this level of abstraction, we also say that each *step* in a Turing machine must be something algorithmically possible. This means that we can use a machine M_A as a "subroutine" in a larger machine M_B, similar to the way in which we nest function calls in our programs.

This means that in addition to being a useful example, each machine that we construct today will be another item in an arsenal of subroutines that we can use to decide larger problems.

3.1 E_{DFA}

Let $E_{DFA} = \{\langle D\rangle \mid D \text{ is a DFA and } L(D) = \emptyset\}$. Show that E_{DFA} is decidable.

$M =$ "On input $\langle D\rangle$:

1. Mark the start state
2. Repeat until no new states are marked:
   (a) Mark any state reachable from an already marked state
3. Iterate over F. If any accepting state is marked, REJECT
4. ACCEPT

3.2 All_{DFA}

Let $ALL_{DFA} = \{\langle D\rangle \mid D \text{ is a DFA and } L(D) = \Sigma^*\}$. Show that ALL_{DFA} is decidable.

One option would be to construct a machine much like the one we built for E_{DFA}:

$M_1 =$ "On input $\langle D\rangle$:

1. Mark the start state
2. Repeat until no new states are marked:
   (a) Mark any state reachable from an already marked state
3. Iterate over $Q - F$. If any non-accepting state is marked, $REJECT$
4. $ACCEPT$

If we can use machines that we've already constructed as subroutines, though, we have a simpler solution:

M_2 = "On input $\langle D \rangle$:

1. Construct a new DFA D' that decides $\overline{L(D)}$.
2. Run a machine M' that decides $\mathrm{E}_{\mathrm{DFA}}$ on $\langle M' \rangle$. If M' Accepts D', Accept.
3. Reject"

How can we construct D'? When we discussed regular languages, we proved that regular languages were closed under complement by showing that we could construct a DFA that decided the complement of any existing DFA by reversing the accepting and non-accepting states (i.e. replacing F with $Q - F$). Since we've already seen that an algorithm exists to do this, we can do it in a single step.

3.3 Worksheet Exercises: $\mathbf{Loop_{NFA}}$ and $\mathbf{Finite_{DFA}}$

3.4 Properties of Machines vs. Properties of Languages

This means that Turing-decidable languages that take an encoding of a DFA D and Accept or Reject it based on some property of $L(D)$ can also be applied to NFAs, GNFAs, and regular expressions.

Example Let $\textsc{Finite}_{\textsc{Regex}} = \{\langle R \rangle \mid R \text{ is a regular expression and } L(R) \text{ is finite}\}$. Show that $\textsc{Finite}_{\textsc{Regex}}$ is decidable.

M = "On input $\langle R \rangle$:

1. Convert R to an equivalent DFA D using the method discussed in class
2. Run a machine that decides $\mathrm{E}_{\mathrm{DFA}}$ on D.
   - If this machine accepts D, Accept R
   - Reject R"

However, this is *not* true of Turing-decidable languages that partition encodings of DFA or NFAs based on properties of the machines themselves. This should go without saying. Consider the other language on the worksheet, $\textsc{Loop}_{\mathrm{NFA}}$. Since we must include a transition for every state/input pair in a DFA, it follows that every DFA contains a loop, even if the language of that DFA is finite. Similarly, since we can reduce any GNFA until $Q = \{q_{start}, q_{accept}\}$ and there is only one transition between them, we can create a GNFA to decide any regular language that does *not* contain a loop. We can convert between equivalent structures in any decider that operates based on the language of some encoded machine, but not in any decider that operates based on some property of the machine itself.

4 Decidable Problems Concerning Context-Free Languages

Example Let $\mathrm{A}_{\mathrm{CFG}} = \{\langle G, w \rangle \mid G \text{ is a context-free grammar capable of generating } w\}$. Show that A_{CFG} is decidable.

There are two ways in which we can approach this problem. The easiest is to simply convert G to an equivalent pushdown automaton P and then running P on w. Context-free grammars and pushdown automata are equivalent in power, so an algorithm to do this does exist. However, if you think back to the fine details of this algorithm, I'm guessing that most of you don't remember exactly how it works. This is because we didn't actually cover it. Converting grammars to pushdown automata isn't actually that bad, but going in the other direction can be rather tedious, and since we've been steadfastly avoiding drawing PDAs wherever possible, it seemed like there were better things on which we could spend our time.

If you don't know the algorithm for converting CFGs to PDAs — as many of us probably don't — you'll have to take the second approach, which is admittedly slightly more complex. If we convert G to an equivalent grammar G' in Chomsky normal form (which is an algorithm we *have* covered), we can take advantage of the deterministic relationship between the length of w and the number of steps used in any production of w.

Using the same strategy that we used to simulate nondeterministic Turing machines, we can generate every parse tree of a given depth d or smaller. By starting with S and treating every production from a given rule as another branch of computation, we can use an address and simulation tape (or a nondeterministic Turing machine) to try all possible derivations with a given number of steps. Since we've shown on Assignment 2 that a derivation of a string w using a CNF grammar requires exactly $2 \cdot |w| - 1$ steps, we can generate all parse trees of length $2 \cdot |w| - 1$ and see whether any of them generate w.

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

1. Convert G to an equivalent CNF grammar G'
2. Generate all possible parse trees with depth $2 \cdot |w| - 1$
    - If the parse tree we just created generates w, Accept
3. Reject"

Note that this is another way of proving something that we've already shown last week: namely, that every context-free language is also a decidable language.

Since we are taking for granted the idea that CFGs and PDAs both decide the same class of language, it's fine to use conversions between the two on any homework or exam question on which it's relevant. To be honest, I mostly just needed that proof to justify having spent time on Chomsky normal form in the first place.

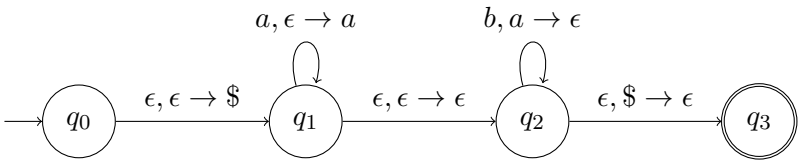
4.1 Reachability in Pushdown Automata

There's another, more legitimate reason that we prefer to work with context-free grammars when possible, which is that "reachability" in pushdown automata is a much trickier concept than it is in DFAs or NFAs.

Example Let $\text{E}_{\text{PDA}} = \{\langle P \rangle \mid P \text{ is a pushdown automaton and } L(P) = \emptyset\}$. Show that E_{PDA} is decidable.

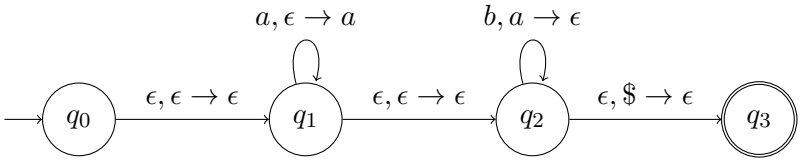
The strategy that we used for E_{DFA} and E_{NFA} was pretty straightforward: we started by marking the start state and then repeatedly marked any state reachable from a marked state. This was doable because state transitions are defined only in terms of what appears in the input string.

In a pushdown automaton, though, this is a bit more complex. Consider the following PDA, which decides $\{a^n b^n \mid n \geq 0\}$:



On this particular PDA, we can use the same strategy as before. We can see that there's a series of transitions that take us from the start state q_0 to the accepting state q_3. This is pretty evident just by following the arrows on the diagram itself.

A minor modification to this PDA complicates things, though:



Here we have a very similar looking diagram. All that changed is the fact that we no longer pushed our delimiting \$ at the beginning of our computation. Since we must pop a \$ to get to q_3, the accepting state is now unreachable, despite the fact that we *still* have a transition from q_0 to q_1, from q_1 to q_2, and from q_2 to q_3. This is no longer sufficient, since we can see by inspection that some of these transitions can never be taken.

Fortunately, since rules in context-free grammars depend only on the variable on their left-hand sides, we can convert our PDA P to an equivalent grammar G and use a similar strategy to solve this problem.

$M =$ "On input $\langle P \rangle$:

1. Convert P to an equivalent context-free grammar G
2. Mark all terminals in G
3. Repeat until no new variables are marked:
    - Mark any variable V that appears on the left-hand side of any rule that generates only marked symbols
4. If the start variable S is marked, Accept
5. Reject"