CS21
Decidability and Tractability

Lecture 8
January 23, 2017
Outline

- deterministic PDAs
- deciding CFLs
- Turing Machines and variants
Deterministic PDA

• A NPDA is a 6-tuple \((Q, \Sigma, \Gamma, \delta, q_0, F)\)
  where:
  - \(\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow \varnothing (Q \times (\Gamma \cup \{\varepsilon\}))\)
    is a function called the transition function

• A deterministic PDA has only one option at every step:
  - for every state \(q \in Q, a \in \Sigma, \) and \(t \in \Gamma,\)
    exactly 1 element in \(\delta(q, a, t),\) or
  - exactly 1 element in \(\delta(q, \varepsilon, t),\) and \(\delta(q, a, t)\)
    empty for all \(a \in \Sigma\)
Deterministic PDA

• A technical detail: we will give our deterministic machine the ability to detect end of input string
  – add special symbol ■ to alphabet
  – require input tape to contain x■

• language recognized by a deterministic PDA is called a deterministic CFL (DCFL)
Example deterministic PDA

\[ L = \{0^n1^n : n \geq 0\} \]

(unpictured transitions go to a “reject” state and stay there)
Deterministic PDA

**Theorem:** DCFLs are closed under complement

(Complement of L in $\Sigma^*$ is $(\Sigma^* - L)$)

**Proof attempt:**

- swap accept/non-accept states
- problem: might enter infinite loop before reading entire string
- machine for complement must accept in these cases, and read to end of string
Example of problem

Language of this DPDA is $0\Sigma^*$
Example of problem

Language of this DPDA is $\{\varepsilon\}$
Deterministic PDA

Proof:

– convert machine into “normal form”
  • always reads to end of input
  • always enters either an accept state or single distinguished “reject” state
– step 1: keep track of when we have read to end of input
– step 2: eliminate infinite loops
Deterministic PDA

step 1: keep track of when we have read to end of input
Deterministic PDA

step 1: keep track of when we have read to end of input

for accept state q': replace outgoing “ε, ? → ?” transition with self-loop with same label
Deterministic PDA

step 2: eliminate infinite loops

– add new “reject” states

\[ a, t \rightarrow t \text{ (for all } a, t) \]
\[ \varepsilon, t \rightarrow t \text{ (for all } t) \]
\[ \text{, } t \rightarrow t \text{ (for all } t) \]
Deterministic PDA

step 2: eliminate infinite loops
– on input x, if infinite loop, then:

infinite sequence \( i_0 < i_1 < i_2 < \ldots \) such that for all \( k \), stack height never decreases below \( \text{ht}(i_k) \) after time \( i_k \)
Deterministic PDA

step 2: eliminate infinite loops
– infinite seq. $i_0 < i_1 < \ldots$ such that for all $k$, stack height never decreases below $ht(i_k)$ after time $i_k$
– infinite subsequence $j_0 < j_1 < j_2 < \ldots$ such that same transition is applied at each time $j_k$

• never see any stack symbol below $t$ from $j_k$ on
• we are in a periodic, deterministic sequence of stack operations independent of the input
Deterministic PDA

step 2: eliminate infinite loops
– infinite subsequence \( j_0 < j_1 < j_2 < \ldots \) such that same transition is applied at each time \( j_k \)
– safe to replace:

\[
\begin{align*}
\epsilon, t \rightarrow s \\
a, t \rightarrow t \quad \text{(for all } a, t) \\
\epsilon, t \rightarrow t \quad \text{(for all } t) \\
\end{align*}
\]
Deterministic PDA

– finishing up…
– have a machine M with no infinite loops
– therefore it always reads to end of input
– either enters an accept state q’, or enters “reject” state r’

– now, can swap: make r’ unique accept state to get a machine recognizing complement of L
Deciding CFLs

• Useful to have an efficient algorithm to decide whether string $x$ is in given CFL
  – e.g. programming language often described by CFG. Determine if string is valid program.

• If CFL recognized by deterministic PDA, just simulate the PDA.
  – but not all CFLs are (homework)…

• Can simulate NPDA, but this takes exponential time in the worst case.
Deciding CFLs

- Convert CFG into Chomsky Normal form.
- Parse tree for string $x$ generated by nonterminal $A$:

  If $A \Rightarrow^k x$ ($k > 1$) then there must be a way to split $x$:

  $x = yz$

  - $A \rightarrow BC$ is a production and
  - $B \Rightarrow^i y$ and $C \Rightarrow^j z$ for $i, j < k$
Deciding CFLs

- An algorithm:

  \textbf{IsGenerated}(x, A)
  
  if $|x| = 1$, then return YES if $A \rightarrow x$ is a production, else return NO
  
  for all $n-1$ ways of splitting $x = yz$
    
    for all $\leq m$ productions of form $A \rightarrow BC$
      
      if IsGenerated($y, B$) and IsGenerated($z, C$), return YES

  return NO

- worst case running time?
Deciding CFLs

• worst case running time $\exp(n)$

• Idea: avoid recursive calls
  – build table of YES/NO answers to calls to IsGenerated, in order of length of substring
  – example of general algorithmic strategy called dynamic programming
  – notation: $x[i,j] = \text{substring of } x \text{ from } i \text{ to } j$
  – table: $T(i, j)$ contains
    
    \{A: A nonterminal such that $A \Rightarrow^* x[i,j]$\}
Deciding CFLs

IsGenerated(x = x_1 x_2 x_3 ... x_n, G)
for i = 1 to n
  \( T[i, i] = \{A: \text{“}A \rightarrow x_i\text{” is a production in } G\} \)
for k = 1 to n - 1
  for i = 1 to n - k
    for k splittings \(x[i, i+k] = x[i,i+j]x[i+j+1, i+k]\)
      \( T[i, i+k] = \{A: \text{“}A \rightarrow BC\text{” is a production in } G \text{ and } B \in T[i,i+j] \text{ and } C \in T[i+j+1,i+k] \} \)
output “YES” if \(S \in T[1, n]\), else output “NO”
Deciding CFLs

\textbf{IsGenerated}(x = x_1x_2x_3...x_n, G)

\begin{enumerate}
  \item for \( i = 1 \) to \( n \)
    \begin{itemize}
      \item \( T[i, i] = \{ A: \text{"A } \rightarrow \text{x}_i \text{" is a production in G} \} \)
    \end{itemize}
  \item for \( k = 1 \) to \( n - 1 \)
  \item for \( i = 1 \) to \( n - k \)
    \begin{itemize}
      \item for \( k \) splittings \( x[i, i+k] = x[i,i+j]x[i+j+1, i+k] \)
        \begin{itemize}
          \item \( T[i, i+k] = \{ A: \text{"A } \rightarrow \text{BC" is a production in G and B } \in T[i,i+j] \text{ and C } \in T[i+j+1,i+k] \} \}
        \end{itemize}
    \end{itemize}
  \end{itemize}
\end{enumerate}

output “YES” if \( S \in T[1, n] \), else output “NO”

\( O(nm) \) steps

\( O(n^3m^3) \) steps
Summary

• Nondeterministic Pushdown Automata (NPDA)
• Context-Free Grammars (CFGs) describe Context-Free Languages (CFLs)
  – terminals, non-terminals
  – productions
  – yields, derivations
  – parse trees
Summary

- grouping determined by grammar
- ambiguity
- Chomsky Normal Form (CNF)

• NDPAs and CFGs are equivalent
• CFL Pumping Lemma is used to show certain languages are not CFLs
Summary

• deterministic PDAs recognize DCFLs
• DCFLs are closed under complement

• there is an efficient algorithm (based on dynamic programming) to determine if a string \( x \) is generated by a given grammar \( G \)
So far…

- several models of computation
  - finite automata
  - pushdown automata
- fail to capture our intuitive notion of what is computable
So far…

• We proved (using constructions of FA and NPDAs and the two pumping lemmas):

\{w : w \in \{a,b\}^* \text{ has an even } \# \text{ of } b's\} \quad \{a^n b^n : n \geq 0 \}\} 

\{a^n b^n c^n : n \geq 0 \}\}

regular languages

context free languages

all languages
A more powerful machine

• limitation of NPDA related to fact that their memory is stack-based (last in, first out)

• What is the simplest alteration that adds general-purpose “memory” to our machine?

• Should be able to recognize, e.g., \( \{a^n b^n c^n : n \geq 0 \} \)