Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

Those who cannot remember the past are condemned to repeat it.

— Jorge Agustín Nicolás Ruiz de Santayana y Borrás, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

C

Dynamic Programming for Formal Languages and Automata

[Read Chapter 3 first.]

Status: Unfinished

This chapter describes dynamic programming algorithms for several problems involving formal languages and finite-state automata. Although I have strived for a self-contained presentation, this material will likely make sense only to people who are already somewhat familiar with formal languages *and* dynamic programming.

C.1 DFA Minimization

Write this.

★★★

C.2 NFA Acceptance

Recall that a ***nondeterministic finite-state automaton***—or ***NFA*** for short—can be described as a directed graph, whose edges are called *states* and whose edges have *labels* drawn from a finite set Σ called the *alphabet*. Every NFA has a designated *start* state and a subset of *accepting* states. Every walk in this graph has a label, which is a string formed by concatenating the labels of the edges in the walk. A string w is *accepted* by an NFA if and only if there is a walk from the start state to one of the accepting states whose label is w.

More formally (or at least, more symbolically), an NFA consists of a finite set Q of states, a start state $s \in Q$, a set of accepting states $A \subseteq Q$, and a transition function $\delta\colon Q \times \Sigma \to 2^Q$. We recursively extend the transition function to a function $\delta^*\colon Q \times \Sigma^* \to 2^Q$ over strings by defining

$$\delta^*(q,w) = \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \delta(q,a)} \delta^*(r,x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

Our NFA ***accepts*** string w if and only if the set $\delta^*(s,w)$ contains at least one accepting state.

We can express this acceptance criterion more directly in terms of the original transition function δ as follows. Let *Accepts?*$(q,w) =$ True if our NFA would accept string w if it started in state q (instead of the usual start state s), and *Accepts?*$(q,w) =$ False otherwise. The function *Accepts?* has the following recursive definition:

$$\mathit{Accepts?}(q,w) := \begin{cases} \textsc{True} & \text{if } w = \varepsilon \text{ and } q \in A \\ \textsc{False} & \text{if } w = \varepsilon \text{ and } q \notin A \\ \displaystyle\bigvee_{r \in \delta(q,a)} \mathit{Accepts?}(r,x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^* \end{cases}$$

Then our NFA accepts w if and only if *Accepts?*$(s,w) =$ True.

Backtracking

In the magical world of non-determinism, we can imagine that the NFA always makes the right decision when faced with multiple transitions, or perhaps spawns off an independent parallel thread for each possible choice. Alas, real computers are neither clairvoyant nor (despite the increasing use of multiple cores) infinitely parallel. To simulate the NFA's behavior directly, we must recursively explore the consequences of each choice explicitly.

Before we can turn our recursive definition(s) into an algorithm, we need to nail down the precise input representation. Let's assume, without loss of generality, that the alphabet is $\Sigma = \{1, 2, \dots, |\Sigma|\}$, the state set is $Q = \{1, 2, \dots, |Q|\}$, the start state is state 1, and our input consists of three arrays:

- A boolean array $A[1..|Q|]$, where $A[q] =$ True if and only if $q \in A$.

- A boolean array $\delta[1\,..\,|Q|, 1\,..\,|\Sigma|, 1\,..\,|Q|]$, where $\delta[p,a,q] = \textsc{True}$ if and only if $p \in \delta(q,a)$.
- An array $w[1\,..\,n]$ of symbols, representing the input string.

Fixing the input string $w[1\,..\,n]$ lets us simplify the definition of the acceptance function slightly. For any state q and index i, define *Accepts?*$(q,i) = \textsc{True}$ if the NFA accepts the suffix $w[i\,..\,n]$ starting in state q, and *Accepts?*$(q,i) = \textsc{False}$ otherwise.

$$\mathit{Accepts?}(q,i) := \begin{cases} \textsc{True} & \text{if } i > n \text{ and } q \in A \\ \textsc{False} & \text{if } i > n \text{ and } q \notin A \\ \displaystyle\bigvee_{r \in \delta(q,a)} \mathit{Accepts?}(r, i+1) & \text{otherwise} \end{cases}$$

In particular, the NFA accepts w if and only if *Accepts?*$(s,1) = \textsc{True}$. This is only a minor notational change from the previous definition, but it makes the recursive calls more efficient (passing just an integer instead of a string) and eventually easier to memoize. Our new recursive definition translates directly into the following backtracking algorithm.

```
ACCEPTS?($q, i$):
  if $i > n$
      return $A[q]$
  for $r \leftarrow 1$ to $|Q|$
      if $\delta[q, w[i], r]$ and ACCEPTS?($r, i+1$)
          return TRUE
  return FALSE
```

Dynamic Programming

The previous algorithm runs in $O(|Q|^n)$ time in the worst case; fortunately, everything is set up to quickly derive a faster dynamic programming solution. The *Accepts?* function can be memoized into a two-dimensional array *Accepts?*$[1\,..\,|Q|, 1\,..\,n+1]$. Each entry *Accepts?*$[q,i]$ depends only on entries of the form *Accepts?*$[r, i+1]$, so we can fill the memoization table by considering indices i in decreasing order in the outer loop, and states q in arbitrary order in the inner loop. Evaluating each entry *Accepts?*$[q,i]$ requires $O(|Q|)$ time, using an even deeper loop over all states r, and there are $O(n|Q|)$ such entries. Thus, the entire dynamic programming algorithm requires $\boldsymbol{O(n|Q|^2)}$ ***time***.

```
NFAACCEPTS?($A[1..|Q|]$, $\delta[1..|Q|, 1..|\Sigma|, 1..|Q|]$, $w[1..n]$):
  for $q \gets 1$ to $|Q|$
      $Accepts?[q, n+1] \gets A[q]$
  for $i \gets n$ down to 1
      for $q \gets 1$ to $|Q|$
          $Accepts?[q, i] \gets$ FALSE
          for $r \gets 1$ to $|Q|$
              if $\delta[q, w[i], r]$ and $Accepts?[r, i+1]$
                  $Accepts?[q, i] \gets$ TRUE
  return $Accepts?[1, 1]$
```

C.3 Regular Expression Matching

★★★

Write this. Assume given an expression tree.

C.4 CFG Parsing

Another problem from formal languages that can be solved via dynamic programming is the ***parsing*** problem for context-free languages. Given a string w and a context-free grammar G, does w belong to the language generated by G? Recall that a ***context-free grammar*** over the alphabet Σ consists of a finite set Γ of *non-terminals* (disjoint from Σ) and a finite set of *production rules* of the form $A \to w$, where A is a nonterminal and w is a string over $\Sigma \cup \Gamma$. The ***length*** of a context free grammar is the number of production rules.

Real-world applications of parsing normally require more information than just a single bit. For example, compilers require parsers that output a *parse tree* of the input code; some natural language applications require the *number* of distinct parse trees for a given string; others assign probabilities to the production rules and then ask for the *most likely* parse tree for a given string. However, once we have an algorithm for the decision problem, it it not hard to extend it to answer these more general questions.

For any nonterminal A and any string x, define $Gen?(A, x) =$ TRUE if x can be derived from A and $Gen?(A, x) =$ FALSE otherwise. At first glance, it seems that the production rules of the CFL immediately give us a (rather complicated) recursive definition for this function. Unfortunately, there are a few subtle problems.[1]

- Consider the context-free grammar $S \to \varepsilon \mid SS \mid (S)$ that generates all properly balanced strings of parentheses. The most straightforward recursive algorithm for $Gen?(S, w)$ recursively checks whether $x \in L(S)$ and $y \in L(S)$, for *every* possible partition $w = x \bullet y$, *including the trivial partitions* $w = \varepsilon \bullet w$ *and* $w = w \bullet \varepsilon$. Thus, $Gen?(S, w)$ can call itself, leading to an infinite loop.

[1]Similar subtleties arise in induction proofs about context-free grammars.

- Consider another grammar that includes the productions $S \to A$, $A \to B$, and $B \to S$, possibly among others. The "obvious" recursive algorithm for $\mathit{Gen?}(S, w)$ must call $\mathit{Gen?}(A, w)$, which calls $\mathit{Gen?}(B, w)$, which calls $\mathit{Gen?}(S, w)$, and we are again in an infinite loop.

To avoid these issues, we will make the simplifying assumption that our input grammar is in ***Chomsky normal form***, which means that it has the following special structure:

- The starting non-terminal S does not appear on the right side of any production rule.
- The grammar *may* include the production rule $S \to \varepsilon$, where S is the starting non-terminal, but does not contain the rule $A \to \varepsilon$ for any other non-terminal $A \neq S$.
- Otherwise, every production rule has the form $A \to BC$ (two non-terminals) or $A \to a$ (one terminal).

Any context-free grammar can be converted into an equivalent grammar Chomsky normal form; moreover, if the original grammar has length L, an equivalent CFG grammar of length $O(L^2)$ can be computed in $O(L^2)$ time. The conversion algorithm is fairly complex, and we haven't yet seen all the algorithmic tools needed to understand it; for purposes of this chapter, it's enough to know that such an algorithm exists. For example, the language of all properly balanced strings of parentheses is generated by the CNF grammar

$$S \to \varepsilon \mid AA \mid BC \qquad A \to AA \mid BC \qquad B \to LA \qquad C \to RA \qquad L \to \texttt{(} \qquad R \to \texttt{)}$$

⟨⟨This is incorrect.⟩⟩ ⋘⋘

With this simplifying assumption in place, the function *Gen?* has a relatively straightforward recursive definition.

$$\mathit{Gen?}(A, x) = \begin{cases} \text{True} & \text{if } |x| \leq 1 \text{ and } A \to x \\ \text{False} & \text{if } |x| \leq 1 \text{ and } A \not\to x \\ \displaystyle\bigvee_{A \to BC} \bigvee_{y \bullet z = x} \mathit{Gen?}(B, y) \wedge \mathit{Gen?}(C, z) & \text{otherwise} \end{cases}$$

The first two cases take care of terminal productions $A \to a$ and the ε-production $S \to \varepsilon$ (if the grammar contains it). Here the notation $A \not\to x$ means that $A \to x$ is *not* a production rule in the given grammar. In the third case, for all production rules $A \to BC$, and for all ways of splitting x into a *non-empty* prefix y and a *non-empty* suffix z, we recursively check whether $y \in L(B)$ and $z \in L(C)$. Because we pass strictly smaller strings in the second argument of these recursive calls, every branch of the recursion tree eventually terminates.

This recurrence was transformed into a dynamic programming algorithm by Tadao Kasami in 1965, and again independently by Daniel Younger in 1967, and again independently by John Cocke in 1970, so of course the resulting algorithm is known as "Cocke-Younger-Kasami", or more commonly ***the CYK algorithm***, with the names listed in *reverse* chronological order.

We can derive the CYK algorithm from the previous recurrence as follows. As usual for recurrences involving strings, we modify the function to accept index arguments instead of strings, to ease memoization. Fix the input string w, and then let $\mathit{Gen?}(A,i,j) = \textsc{True}$ if and only if the substring $w[i\,..\,j]$ can be derived from non-terminal A. Now our earlier recurrence can be rewritten as follows:

$$\mathit{Gen?}(A,i,j) = \begin{cases} \textsc{True} & \text{if } i=j \text{ and } A \to w[i] \\ \textsc{False} & \text{if } i=j \text{ and } A \not\to w[i] \\ \displaystyle\bigvee_{A \to BC} \bigvee_{k=i}^{j-1} \mathit{Gen?}(B,i,k) \wedge \mathit{Gen?}(C,k+1,j) & \text{otherwise} \end{cases}$$

Then w lies in the language of the grammar if and only if either $\mathit{Gen?}(A,1,n) = \textsc{True}$, or $w = \varepsilon$ and the grammar includes the production $S \to \varepsilon$.

We can memoize the function *Gen?* into a three-dimensional boolean array $\mathit{Gen}[1\,..\,|\Gamma|, 1\,..\,n, 1\,..\,n]$, where the first dimension is indexed by the non-terminals Γ in the input grammar. Each entry $\mathit{Gen}[A,i,j]$ in this array depends on entries of the form $\mathit{Gen}[\,\cdot\,,i,k]$ for some $k < j$, or $\mathit{Gen}[\,\cdot\,,k+1,j]$ for some $k \geq i$. Thus, we can fill the array by increasing j and decreasing i in two outer loops, and considering non-terminals A in arbitrary order in the inner loop. The resulting dynamic programming algorithm runs in ***$O(n^3L)$ time***, where L is the length of the input grammar.

```
CYKParse($w, G$):
  if $w = \varepsilon$
      if $G$ contains the production $S \to \varepsilon$
          return True
      else
          return False
  for $i \leftarrow 1$ to $n$
      for all non-terminals $A$
          if $G$ contains the production $A \to w[i]$
              $Gen[A,i,i] \leftarrow$ True
          else
              $Gen[A,i,i] \leftarrow$ False
  for $j \leftarrow 1$ to $n$
      for $i \leftarrow n$ down to $j+1$
          for all non-terminals $A$
              $Gen[A,i,j] \leftarrow$ False
          for all production rules $A \to BC$
              for $k \leftarrow i$ to $j-1$
                  if $Gen[B,i,k]$ and $Gen[C,k+1,j]$
                      $Gen[A,i,j] \leftarrow$ True
  return $Gen[S,1,n]$
```

Exercises

Add some exercises! ★★★