Ling 5801: Lecture Notes 10
From Recursion to Tractable CFG Recognition

Contents

10.1 We can use recursion to implement a CFG recognizer

Recall the definition of $L(G)$ for any CFG $G = \langle C, X, S, R \rangle$:

$$L(G) = \{x_1..x_n \mid \exists_{c \in S}\ c \xrightarrow[G]{*} x_1..x_n\}$$

where:

$$c \xrightarrow[G]{*} x_i..x_j \text{ iff } \begin{cases} \text{if } i = j : c = x_i \\ \text{if } i < j : \exists_{k,d,e}\ c \rightarrow d\ e \in R \wedge d \xrightarrow[G]{*} x_i..x_k \wedge e \xrightarrow[G]{*} x_{k+1}..x_j \end{cases}$$

For CFG G, we can convert this to a recursive function to recognize $\mathcal{L}(G)$:

```
import sys
import re
import model

S = model.Model('S')
C = model.Model('C')
R = model.Model('R')

def Rec(c,i,j,X):
    if i == j:
        return (c==X[i])
    else:
        v = False
        for k in range(i,j):
            for d in C:
                for e in C:
                    if (c,d,e) in R:
                        v = v or (R[c,d,e] and Rec(d,i,k,X) and Rec(e,k+1,j,X))
    return v

for line in sys.stdin:
    S.read(line)
    C.read(line)
    R.read(line)
    m = re.search('I (.*)',line)
    if m != None:
        I = re.split(' +',m.group(1))
```

```
        print ( Rec('S',0,len(I)-1,I) )
```

When run on the model file `cfg.model`:

```
S : S = 1

C : S = 1
C : VP = 1
C : NP = 1
C : PP = 1
C : the = 1
C : cat = 1
C : hit = 1
C : toy = 1
C : off = 1
C : mat = 1

R : S NP VP = 1
R : VP VP PP = 1
R : VP hit NP = 1
R : PP off NP = 1
R : NP NP PP = 1
R : NP the cat = 1
R : NP the toy = 1
R : NP the mat = 1
```

and the input file `cat-toy.in`:

```
I the cat hit the toy off the mat
```

(e.g. `cat cfg.model cat-toy.in | python parser.py`)

produces:

```
True
```

Correctness:

Use 'recursion invariant': `Rec(c,i,j,X)` computes $c \xrightarrow[G]{*} x_i..x_j$

Complexity:

$\tau(\texttt{Rec}, n) \;\geq\; n \cdot |C| \cdot |C| \cdot \tau(\texttt{Rec}, n{-}1) \;\geq\; n! \cdot |C|^{2n} \;\notin\; \mathcal{O}(n^k)$ — not polynomial!

(NOTE: technically, 'lazy evaluation' saves us, but only for boolean semirings)

10.2 Memoized algorithm: record partial results

Avoid duplication of effort by recording partial results in `V`, checking for duplicates:

```
V = {}

def Rec(c,i,j,X):
    if (c,i,j) not in V:
```

```
        if i == j:
            return (c==X[i])
        else:
            V[c,i,j] = False
            for k in range(i,j):
                for d in C:
                    for e in C:
                        if (c,d,e) in R:
                            V[c,i,j] = V[c,i,j] or (R[c,d,e] and
                                                    Rec(d,i,k,X) and
                                                    Rec(e,k+1,j,X))
    return V[c,i,j]
```

Correctness:

Same recursion invariant: `Rec(c,i,j,X)` computes $c \xrightarrow[G]{*} x_i..x_j$

Only change was to add first line to check for duplicates

Complexity:

Recursion only explored once for each (c, i, j)

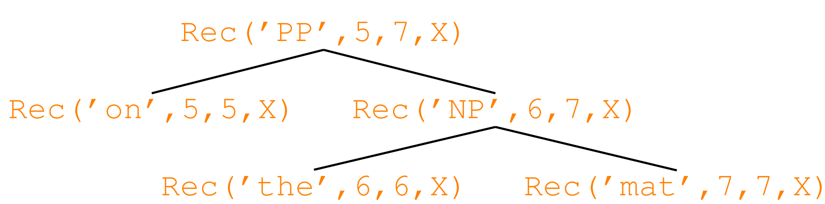
Since only $|C| \cdot n \cdot n$ possible instances of (c, i, j):

$\tau(\texttt{Rec}, n) \;\in\; \mathcal{O}(|C| \cdot n \cdot n \cdot n \cdot |C| \cdot |C|) \;=\; \mathcal{O}(|C|^3 n^3)$ — now it is polynomial!

This is called ‘dynamic programming’

10.3 Tabular/‘bottom-up’ dynamic programming algorithm

Consider ‘recursion tree’ defined by usage of program stack in memoized DP algo:



This can be simplified to remove function recursion.

Loop over each `Rec(c,i,j,X)` from bottom to top of recursion tree
(has to be bottom-up to ensure sub-solutions are there when you need them):

```
V = {}

def Rec(cS,_,n,X):
    for j in range(0,n+1):
        for i in range(j,-1,-1):
            for c in C:
                if i == j:
```

```
                V[c,i,j] = (c==X[i])
            else:
                V[c,i,j] = False
                for k in range(i,j):
                    for d in C:
                        for e in C:
                            if (c,d,e) in R:
                                V[c,i,j] = V[c,i,j] or (R[c,d,e] and
                                                        V[d,i,k] and
                                                        V[e,k+1,j])
    return V[cS,0,n]
```

Any memoized recursive algorithm can be rewritten this way!

Here's an example run:

V:	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	the	NP			S			S
i=1		cat						
i=2			hit		VP			VP
i=3				the	NP			NP
i=4					toy			
i=5						off		PP
i=6							the	NP
i=7								mat

Correctness:

Loop invariant instead of recursion, but still: `c,i,j` computes $c \xrightarrow[G]{*} x_i..x_j$

Only change in outer loops

Complexity:

nested loops: $\tau(\texttt{Rec}, n) \;\in\; \mathcal{O}(n \cdot n \cdot |C| \cdot n \cdot |C| \cdot |C|) \;=\; \mathcal{O}(|C|^3 n^3)$ — still polynomial!