

Ling 5801: Lecture Notes 11

From CFG Recognition to Probabilistic Parsing

Contents

11.1 Generalization of algorithms using semiring substitution	1
11.2 Generalized parsing	2
11.3 From recognition to parsing	3
11.4 Weight calculation	5
11.5 Weighted Parsing	7
11.6 FSA can also be generalized	8
11.7 Where do weights come from?	8
11.8 A case against the dynamic programming parser as a human model	8

11.1 Generalization of algorithms using semiring substitution

Operations in an algorithm can be replaced, keeping the same structure.

For ‘dynamic programming’ algorithms, this can be done using semiring substitution:

A **semiring** is a tuple $\langle V, \oplus, \otimes, v_{\perp}, v_{\top} \rangle$ such that:

- V is a domain of values
 - \oplus is a function $V \times V \rightarrow V$ such that:
 - \oplus is **associative** (parens in sequences of operands don’t matter):

$$v \oplus (v' \oplus v'') = (v \oplus v') \oplus v''$$
 - \oplus is **commutative** (order of operands doesn’t matter):

$$v \oplus v' = v' \oplus v$$
 - \otimes is a function $V \times V \rightarrow V$ such that:
 - \otimes is **associative** (parens in sequences of operands don’t matter):

$$v \otimes (v' \otimes v'') = (v \otimes v') \otimes v''$$
 - \otimes **distributes** over \oplus (that is, \otimes with common operands can jump outside \oplus):

$$(v \otimes v') \oplus (v \otimes v'') = v \otimes (v' \oplus v''),$$

$$(v' \otimes v) \oplus (v'' \otimes v) = (v' \oplus v'') \otimes v$$
- or in the case of limit operators (which we often use in dynamic programming):

$$\bigoplus_{v'} v \otimes v' = v \otimes \bigoplus_{v'} v'$$

e.g. products involving variables not bound by sums may move outside sum ‘loop’:

$$\sum_{p'} p \cdot p' = p \cdot \sum_{p'} p' \quad (5 \cdot 1 + 5 \cdot 2 = 5 \cdot (1 + 2)) \text{ a.k.a. } \sum_{p' \in \{1,2\}} 5 \cdot p' = 5 \cdot \sum_{p' \in \{1,2\}} p'$$

or conjuncts may move outside disjunct ‘loop’:

$$\bigvee_{b'} b \wedge b' = b \wedge \bigvee_{b'} b'$$

- v_{\perp} is an **identity** element for \oplus and **annihilator** for \otimes (like 0 in reals):

- $v_{\perp} \in V$
- $v \oplus v_{\perp} = v$ and $v_{\perp} \oplus v = v$
- $v \otimes v_{\perp} = v_{\perp}$ and $v_{\perp} \otimes v = v_{\perp}$

- v_{\top} is an **identity** element for \otimes (like 1 in reals):

- $v_{\top} \in V$
- $v \otimes v_{\top} = v$ and $v_{\top} \otimes v = v$

A parser can generalize, using different semirings for operators \oplus, \otimes and initial values of V :

- **boolean** semiring $\langle \{\text{TRUE}, \text{FALSE}\}, \vee, \wedge, \text{FALSE}, \text{TRUE} \rangle$: get original recognizer
- **state sequences** $\langle Q^*, |, \circ, q_{\perp}, \epsilon \rangle$: get set of possible trees/sequences
- **forward/inside** $\langle \mathbb{R}_0^{\infty}, +, \cdot, 0, 1 \rangle$: get probability
- **tropical** semiring $\langle \mathbb{R}_{-\infty}^0 \cup \{-\infty\}, \min, +, -\infty, 0 \rangle$: get best tree/sequence prob
- **state sequence** \times **tropical**: best tree/sequence and probability
- ...

11.2 Generalized parsing

Any time you want to calculate something of the form:

$$f(c, x_i \dots x_j) = \bigoplus_{\tau \text{ w. root } \langle c, i, j \rangle} \bigotimes_{\langle c', i', j' \rangle \in \tau} \begin{cases} \text{if } i' = j' : \begin{cases} \text{if } c' = x_{i'} : v_{\top} \\ \text{if } c' \neq x_{i'} : v_{\perp} \end{cases} \\ \text{if } i' < j' : \bigoplus_{\substack{k, d', e' \text{ s.t. } \langle d', i', k \rangle, \langle e', k+1, j' \rangle \in \tau}} R(c' \rightarrow d' e') \end{cases}$$

you can apply generalized distributive axiom (pull meta-conjunct out of meta-disjunction):

$$f(c, x_i..x_j) = \begin{cases} \text{if } i=j : \begin{cases} \text{if } c=x_i : v_{\top} \\ \text{if } c \neq x_i : v_{\perp} \end{cases} \\ \text{if } i < j : \bigoplus_{k,d,e} R(c \rightarrow d e) \otimes \left(\bigoplus_{\tau' \text{ w. root } \langle d,i,k \rangle} \bigotimes_{\langle c',i',j' \rangle \in \tau'} \{ \dots \} \right) \otimes \left(\bigoplus_{\tau'' \text{ w. root } \langle e,k+1,j \rangle} \bigotimes_{\langle c'',i'',j'' \rangle \in \tau''} \{ \dots \} \right) \end{cases}$$

and identify recursive instances of $f(c, x_i..x_j)$:

$$f(c, x_i..x_j) = \begin{cases} \text{if } i=j : \begin{cases} \text{if } c=x_i : v_{\top} \\ \text{if } c \neq x_i : v_{\perp} \end{cases} \\ \text{if } i < j : \bigoplus_{k,d,e} R(c \rightarrow d e) \otimes f(d, x_i..x_k) \otimes f(e, x_{k+1}..x_j) \end{cases}$$

then code, memoize, tabularize using dynamic programming, still preserving the generality:

```
def Parse(cS, X) :
    T = len(X)
    for j in range(0, T) :
        for i in range(j, -1, -1) :
            for c in C :
                if i == j :
                    if ( c==X[i] ) : V[c, i, j] = v⊤
                    else : V[c, i, j] = v⊥
                else :
                    V[c, i, j] = v⊥
                    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] ⊕ ⊗ (val(c, d, e),
                                                                    V[d, i, k],
                                                                    V[e, k+1, j])
    return V[cS, 0, T-1]
```

11.3 From recognition to parsing

Semiring basis lets us substitute the Boolean semiring of recognizer $\langle \{T, F\}, \vee, \wedge, F, T \rangle$ with union / Cartesian product: $\langle \text{set of trees}, \cup, \times, \emptyset, \{\langle \rangle\} \rangle$

Tree sets:

$$f(c, x_i..x_j) = \bigcup_{\tau \text{ w. root } \langle c,i,j \rangle} \bigotimes_{\langle c',i',j' \rangle \in \tau} \begin{cases} \text{if } i'=j' : \begin{cases} \text{if } c'=x_{i'} : \{\langle \rangle\} \\ \text{if } c' \neq x_{i'} : \emptyset \end{cases} \\ \text{if } i' < j' : \bigcup_{k',d',e' \text{ s.t. } \langle d',i',k' \rangle, \langle e',k'+1,j' \rangle \in \tau} R(c' \rightarrow d' e') \end{cases}$$

can be computed with:

```
import sys
import re
```

```

import model

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

V = {}

def val(c,d,e):
    return [c]

def prod(l1,l2,l3) :
    lo = []
    for e1 in l1 :
        for e2 in l2 :
            for e3 in l3 :
                lo = lo + [(e1,e2,e3)]
    return lo

def Parse(cS,X) :
    T = len(X)
    for j in range(0,T) :
        for i in range(j,-1,-1) :
            for c in C :
                if i == j :
                    if ( c==X[i] ) : V[c,i,j] = [X[i]]
                    else : V[c,i,j] = []
                else :
                    V[c,i,j] = []
                    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] + prod(val(c,d,e),
                                                                    V[d,i,k],
                                                                    V[e,k+1,j])

    return V[cS,0,T-1]

for line in sys.stdin:
    S.read(line)
    C.read(line)
    R.read(line)

print Parse('S',re.split(' +','the cat hit the toy off the mat'))

```

run on the 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 : under = 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

```

gives output (indented by me to help you see what happened):

```

[ ('S', ('NP', 'the', 'cat'), ('VP', 'hit', ('NP', ('NP', 'the', 'toy'),
                                                ('PP', 'off', ('NP', 'the', 'mat'))))),
  ('S', ('NP', 'the', 'cat'), ('VP', ('VP', 'hit', ('NP', 'the', 'toy')),
                                ('PP', 'off', ('NP', 'the', 'mat'))))]

```

You can turn any recognizer into an analyzer/parser with this trick!

(‘real’ parsers use probability weights to choose a single tree; but that’s another semiring)

Correctness: mostly the same

loop invariant: each c, i, j computes set of trees with root c spanning $x_i..x_j$

Complexity: same (with assumptions)

no change to program structure (assuming `prod` implemented w. references, which this ain’t)

Worked example: (blackboard)

11.4 Weight calculation

Define weights for trees based on (product of) weights for rules:

$$P(x_i..x_j | c) = \sum_{\tau \text{ w. root } \langle c, i, j \rangle} \prod_{\langle c', i', j' \rangle \in \tau} \begin{cases} \text{if } i' = j' : \begin{cases} \text{if } c' = x_{i'} : 1.0 \\ \text{if } c' \neq x_{i'} : 0.0 \end{cases} \\ \text{if } i' < j' : \sum_{k', d', e' \text{ s.t. } \langle d', i', k' \rangle, \langle e', k'+1, j' \rangle \in \tau} R(c' \rightarrow d' e') \end{cases}$$

can be computed with:

```

import sys
import re
import model

```

```

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

V = {}

def val(c,d,e):
    return R[c,d,e]

def Parse(cS,X) :
    T = len(X)
    for j in range(0,T) :
        for i in range(j,-1,-1) :
            for c in C :
                if i == j :
                    if ( c==X[i] ) : V[c,i,j] = 1.0
                    else : V[c,i,j] = 0.0
                else :
                    V[c,i,j] = 0.0
                    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] + (val(c,d,e) *
                                                                V[d,i,k] *
                                                                V[e,k+1,j])

    return V[cS,0,T-1]

for line in sys.stdin:
    S.read(line)
    C.read(line)
    R.read(line)

print Parse('S',re.split(' +','the cat hit the toy off the mat'))

```

run on the weighted 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 : under = 1
C : mat = 1

R : S NP VP = 1.0
R : VP VP PP = .5
R : VP hit NP = .5

```

```
R : PP off NP = 1
R : NP NP PP = .25
R : NP the cat = .25
R : NP the toy = .25
R : NP the mat = .25
```

outputs the combined weight of the string, given these rule weights:

```
0.005859375
```

11.5 Weighted Parsing

Choose a single tree using weighted rules:

```
import sys
import re
import model

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

V = {}

def val(c,d,e):
    return (R[c,d,e],c)

def max_argmax(pt1,pt2) :
    if pt1[0]>=pt2[0] : return pt1
    else : return pt2

def prod_pair(pt1,pt2,pt3) :
    return ( pt1[0]*pt2[0]*pt3[0], (pt1[1],pt2[1],pt3[1]) )

def Parse(cS,X) :
    T = len(X)
    for j in range(0,T) :
        for i in range(j,-1,-1) :
            for c in C :
                if i == j :
                    if ( c==X[i] ) : V[c,i,j] = (1.0,X[i])
                    else : V[c,i,j] = (0.0,())
                else :
                    V[c,i,j] = (0.0,())
                    for k in range(i,j) :
                        for d in C :
                            for e in C :
                                if (c,d,e) in R :
                                    V[c,i,j] = max_argmax(V[c,i,j],
                                                            prod_pair(val(c,d,e),
                                                            V[d,i,k],
                                                            V[e,k+1,j]))

    return V[cS,0,T-1]
```

```

for line in sys.stdin:
    S.read(line)
    C.read(line)
    R.read(line)

print Parse('S',re.split(' +','the cat hit the toy off the mat'))

```

This prints most weighty tree for this string, and its weight:

```

(0.00390625, ('S', ('NP', 'the', 'cat'), ('VP', ('VP', 'hit', ('NP', 'the', 'toy')),
                                             ('PP', 'off', ('NP', 'the', 'mat')))))

```

Worked example: (blackboard)

11.6 FSA can also be generalized

A_{FSA} can now be generalized:

```

# initialize table of possible states at each time step using start states
V = {}
for q in Q:
    V[0,q] = S.get(q, v_1)

# for each possible state qP in V at time t, for each qP,x,q in M, add q
for t in range(1,len(Input)):
    for qP in Q:
        for q in Q:
            V[t,q] = V.get((t,q), v_1) ⊕ (V[t-1,qP] ⊗ M.get((qP,Input[t-1],q), v_1))

```

11.7 Where do weights come from?

Weights are well defined as probabilities.

In this view, (human/machine) parsers estimate probability of speakers generating utterances.

11.8 A case against the dynamic programming parser as a human model

DP/‘chart’ parsers are simple and tractable, but cognitively implausible:

1. human language processing uses short-term working memory:
 - Just and Carpenter: memory load affects processing [Just and Carpenter, 1992]
2. short-term working memory is very limited:
 - Miller: 7 +/- 2 ‘chunks’ [Miller, 1956]

- Cowan: 4 +/- 1 [Cowan, 2001]
 - Lewis: 2 +/- 1 [Lewis, 1996]
 - McElree and Doshier: 1, but continuous [McElree and Doshier, 2001]
3. short-term memory is short-term (no trees in memory):
- Sachs: can't remember words between sentences [Sachs, 1967]
 - Jarvella: can't remember words within sentences [Jarvella, 1971]
4. reference interacts incrementally with processing
- Tanenhaus et al.: cand-..., frog on ... (can't do bottom-up) [Tanenhaus et al., 1995]
5. don't need more than working memory anyway:
- Schuler et al.: parse treebank using 3-4 chunks [Schuler et al., 2010]

Let's implement an incremental comprehension model...

References

- [Cowan, 2001] Cowan, N. (2001). The magical number 4 in short-term memory: A reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24:87–185.
- [Jarvella, 1971] Jarvella, R. J. (1971). Syntactic processing of connected speech. *Journal of Verbal Learning and Verbal Behavior*, 10:409–416.
- [Just and Carpenter, 1992] Just, M. A. and Carpenter, P. A. (1992). A capacity theory of comprehension: Individual differences in working memory. *Psychological Review*, 99:122–149.
- [Lewis, 1996] Lewis, R. L. (1996). Interference in short-term memory: The magical number two (or three) in sentence processing. *The Journal of Psycholinguistic Research*, 25:93–115.
- [McElree and Doshier, 2001] McElree, B. and Doshier, B. A. (2001). The focus of attention across space and across time. *Behavioral and Brain Sciences*, 24:129–130.
- [Miller, 1956] Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63:81–97.
- [Sachs, 1967] Sachs, J. (1967). Recognition memory for syntactic and semantic aspects of connected discourse. *Perception and Psychophysics*, 2:437–442.
- [Schuler et al., 2010] Schuler, W., AbdelRahman, S., Miller, T., and Schwartz, L. (2010). Broad-coverage incremental parsing using human-like memory constraints. *Computational Linguistics*, 36(1):1–30.

[Tanenhaus et al., 1995] Tanenhaus, M. K., Spivey-Knowlton, M. J., Eberhard, K. M., and Sedivy, J. C. E. (1995). Integration of visual and linguistic information in spoken language comprehension. *Science*, 268:1632–1634.