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 \( \langle V, \oplus, \otimes, v_\bot, v_\top \rangle \) such that:

- \( V \) is a domain of values
- \( \oplus \) is a function \( V \times V \to V \) 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 \( V \times V \to V \) 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’:
    \[ \sum_{p' \in \{1, 2\}} \sum_{p'} p \cdot p' = p \cdot \sum_{p' \in \{1, 2\}} p' \cdot 5 \cdot (1 + 2) = 5 \cdot \sum_{p' \in \{1, 2\}} p' \cdot 5 \]
    
    or conjuncts may move outside disjunct ‘loop’:
    \[ \bigvee_{b'} b \land b' = b \land \bigvee_{b'} b' \]

- \( v_\bot \) is an identity element for \( \oplus \) and annihilator for \( \otimes \) (like 0 in reals):
  - \( v_\bot \in V \)
  - \( v \oplus v_\bot = v \) and \( v_\bot \oplus v = v \)
  - \( v \otimes v_\bot = v_\bot \) and \( v_\bot \otimes v = v_\bot \)

- \( v_\top \) is an identity element for \( \otimes \) (like 1 in reals):
  - \( v_\top \in V \)
  - \( v \otimes v_\top = v \) and \( v_\top \otimes v = v_\top \)

Parser can generalize, using different semirings for operators \( \oplus, \otimes \) and initial values of \( V \):

- boolean semiring \( \langle \{\text{TRUE, FALSE}\}, \lor, \land \rangle \): get original recognizer
2. Generalized parsing:

Any time you want to calculate something of the form:

\[
f(c, x_i, x_j) = \bigoplus_{\tau \text{ w. root } (c, i, j)} \bigotimes_{(c', i', j') \in \mathcal{R}} \begin{cases} 
    \text{if } i' = j': & \{ \text{if } c' = x_{i'} : v_{\tau} \\
    \text{if } c' \neq x_{i'} : v_{\perp} \}
\end{cases} \bigotimes_{k, d, e} R(c' \rightarrow d' e')
\]

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

\[
f(c, x_i, x_j) = \begin{cases} 
    \text{if } i = j: & \{ \text{if } c = x_i : v_{\tau} \\
    \text{if } c \neq x_i : v_{\perp} \}
\end{cases} \bigotimes_{\tau' \text{ w. root } (c, i, k)} \bigotimes_{(c', i', j') \in \mathcal{R}'} \bigotimes_{k, d, e} R(c' \rightarrow d' e')
\]

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

\[
f(c, x_i, x_j) = \begin{cases} 
    \text{if } i = j: & \{ \text{if } c = x_i : v_{\tau} \\
    \text{if } c \neq x_i : v_{\perp} \}
\end{cases} \bigotimes_{k, d, e} R(c \rightarrow d e) \bigotimes f(d, x_{i+1}, x_j) \bigotimes f(e, x_{k+1}, x_j)
\]

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

```python
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_{\tau}
                    else:
                        V[c, i, j] = v_{\perp}
                else:
                    V[c, i, j] = v_{\perp}
                    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] \oplus (\text{val}(c, d, e),
                                    V[d, i, k],
                                    V[e, k+1, j])
    return V[cS, 0, T-1]
```
3. From recognition to parsing:

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

Tree sets:

\[
f(c, x_i \ldots x_j) = \bigcup_{\tau \text{ w. root } \langle c, i, j \rangle} \times \left\{ \begin{array}{ll}
\text{if } i' = j' : & \{ \text{if } c' = x_{i'} : \{\langle \rangle\} \}
\text{if } c' \neq x_{i'} : \emptyset \\
\text{if } i' < j' : & \bigcup_{k, d, e \text{ s.t. } (d, i, k), (e, k+1, j') \in \tau} R(c' \to d' e')
\end{array} \right. \]

can be computed with:

```python
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:
    ...
```
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','the','toy'))],
 ('PP','off',('NP','the','mat'))],
[['S',('NP','the','cat'),('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)
4. Weight calculation:

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

\[ P(x_{i..j}|c) = \sum_{\tau \in \text{root}(c, i...j)} \prod_{(c', i', j') \in \tau} \begin{cases} \text{if } i' = j' : & \begin{cases} \text{if } c' = x_r : 1.0 \\
\text{if } c' \neq x_r : 0.0 \end{cases} \\ 
\text{if } i' < j' : & \sum_{k', d', e' \text{ s.t. } (d', k, e', k' + 1, j') \in \tau} \end{cases} R(c' \rightarrow d' e') \]

can be computed with:

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

0.005859375

5. Weighted Parsing:

Choose a single tree using weighted rules:

```python
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[S,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)

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⊥)

# 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⊥) ⊕ (V[t-1,qP] ⊗ M.get((qP,Input[t-1],q),v⊥))

7. Where do weights come from?

Weights are well defined as probabilities.
In this view, parser (human or machine) estimates prob. of speaker generating utterance.
Probability in this view is a subjective measure of belief about speaker behavior.

Specifically, belief of proposition $x$ in domain $X$:
- Domain: set of mutually exclusive possible propositions (e.g. FSA states / PDA store-states)
- Belief: given an infinite number of trials of $X$, $x$ would happen $p$ of the time

Notation of propositions:
- $x, y, u, v$: uncertain true/false proposition (e.g. *Kim said ‘cost’*), believed w. some probability
- $X$: a domain of possible mutually-exclusive propositions (e.g. {*Kim said ‘caused’, ... *})
- $x \lor x'$: either $x$ or $x'$ is true (e.g. *Kim said ‘cost’ or Kim said ‘caused’*)
- $x, x'$: both $x$ and $x'$ are true (separate variables; e.g. *Kim said ‘cost’ and Pat said ‘caused’*)
- $\top$: tautology / empty proposition

Notation of limit operators:
- $\sum_{x \in X} \phi$: sum of $\phi$ over all $x$ in $X$
- $\prod_{x \in X} \phi$: product of $\phi$ over all $x$ in $X$
- $\max_{x \in X} \phi$: maximum of $\phi$ over all $x$ in $X$
- $\arg\max_{x \in X} \phi$: value of $x$ that maximizes $\phi$ over all values in $X$

Notation of probability terms:
- $\tilde{F}(x)$: frequency of $x$ in trials
- $P(x)$ or $P(x \mid \top)$: prior probability
  $$P(x) = \frac{\tilde{F}(x)}{\sum_{x \in X} \tilde{F}(x)}$$
- $P(x \mid y)$: conditional probability
  $$P(x \mid y) = \frac{\tilde{F}(x, y)}{\sum_{x \in X} \tilde{F}(x, y)}$$
- $P_\pi(x)$ or $P_\theta(x \mid y)$: prior/conditional probability as defined in some model $\pi$ or $\theta$

Probability axioms: all probabilities $P(x \mid y)$ are real numbers such that...

- $0 \leq P(x \mid y) \leq 1$
- $\sum_{x \in X} P(x \mid y) = 1$
- $\forall x, x' \in X \ P(x \lor x' \mid y) = P(x \mid y) + P(x' \mid y)$

This means, if $X = V \times U$:

- $P(u \lor v \mid y) = P(u \mid y) + P(v \mid y) - P(u, v \mid y)$ (and $x' = y$ may be underspecified)

E.g., if $V = \{ *Kim said ‘cost’, ..., ‘caused’* \}$ and $U = \{ *Pat said ‘cost’, ... ‘caused’* \}$:

- $x_0 = K:caus, P:caus, x_1 = K:caus, P:cost, x_2 = K:cost, P:caus, x_3 = K:cost, P:cost$
- $v = K:cost, u = P:cost$
- $P(x_1 \lor x_2 \lor x_3 \mid y) = P(x_2 \lor x_3 \mid y) + P(x_1 \lor x_3 \mid y) - P(x_3 \mid y)$
Probabilities of grammar rule expansions:
\[ P(c \rightarrow d \; e \mid c) \] probability speaker decided to expand \( c \) into \( d \) followed by \( e \)
‘branching process model’ assigns probability to any tree / sentence
widely used in comp ling / comp psycholing

8. A case against the dynamic programming parser as a human model:
DP/‘chart’ parsers are simple and tractable, but cognitively implausible:

(a) human language processing uses short-term working memory:
   - Just and Carpenter: memory load affects processing [Just and Carpenter, 1992]

(b) 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 Dosher: 1, but continuous [McElree and Dosher, 2001]

(c) 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]

(d) reference interacts incrementally with processing
   - Tanenhaus et al.: can-..., frog on ... (can’t do bottom-up) [Tanenhaus et al., 1995]

(e) 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**


