A Pushdown Transducer Extension for the OpenFst Library

Cyril Allauzen and Michael Riley

Google Research, 76 Ninth Avenue, New York, NY 10011, USA
{allauzen,riley}@google.com

Abstract. Pushdown automata are devices that can efficiently represent context-free languages, have natural weighted versions, and combine naturally with finite automata. We describe a pushdown transducer extension to OpenFst, a weighted finite-state transducer library. We present several weighted pushdown algorithms, some with clear finite-state analogues, describe their library usage and give some applications of these methods to recognition, parsing and translation.

1 Introduction

OpenFst is an open-source C++ software library for creating, combining, searching and optimizing finite-state transducers (FSTs) [?]. Weighted FSTs have many applications in speech and language processing, computational biology and other areas and the availability of flexible, large-scale algorithms libraries allows rapid experimentation and development [?]. However, there are problems that are not well-represented by finite automata such as aspects of natural language parsing or translation. In particular, a context-free representation may be better suited either because the language considered is not regular or is more compactly represented in a recursive manner.

In these cases, a common approach is to use a weighted context-free grammar as the representation. However, weighted *pushdown automata* offer an attractive alternative. As automata, they are more closely tied to computation and can share and mix with finite automata in a natural way [?]. Our goal here is to present several weighted pushdown algorithms, some with clear finite-state analogues, to describe their realization in a pushdown transducer extension to the OpenFst library and to give some applications of these methods and the library.

2 Definitions

Informally, pushdown transducers are finite-state transducers that have been augmented with a stack. Typically this is done by adding a stack alphabet and labeling each transition with a stack operation (a stack symbol to be pushed onto, popped or read from the stack) in additon to the usual input and output labels [?,?] and weight [?,?]. Our equivalent representation allows a transition to be labeled by a stack operation or regular input/output symbols but not both.

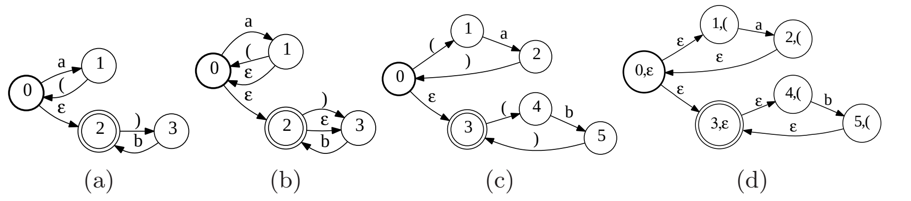


Fig. 1. PDA Examples: (a) Non-rational PDA A_1 accepting $\{a^n b^n | n \in \mathbb{N}\}$. (b) Rational (but not bounded-stack) PDA A_2 accepting a^*b^*. (c) Bounded-stack PDA A_3 accepting a^*b^* and (d) its expansion A_4 as an FSA.

Stack operations are represented by pairs of open and close parentheses (pushing a symbol on and popping it from the stack). The advantage of this representation is that it is identical to the finite-state transducer representation except that certain symbols (the parentheses) have special semantics. As such, several finite-state algorithms either immediately generalize to this PDT representation or do so with minimal changes.

2.1 Dyck Languages

A (restricted) Dyck language consists of “well-formed” or “balanced” strings over a finite number of pairs of parentheses. Thus the string ([( ) ( )] { } [ ]) ( ) is in the Dyck language over three pairs of parentheses (following [**?**]).

More formally, let A and $\overline{A}$ be two finite alphabets such that there exists a bijection f from A to $\overline{A}$. Intuitively, f maps an opening parenthesis to its corresponding closing parenthesis. Let $\bar{a}$ denote $f(a)$ if $a \in A$ and $f^{-1}(a)$ if $a \in \overline{A}$. The *Dyck language* D_A over the alphabet $\widehat{A} = A \cup \overline{A}$ is then the language defined by the following context-free grammar: $S \to \epsilon$, $S \to SS$ and $S \to aS\bar{a}$ for all $a \in A$. We define the mapping $c_A : \widehat{A}^* \to \widehat{A}^*$ as follows. $c_A(x)$ is the string obtained by iteratively deleting from x all factors of the form $a\bar{a}$ with $a \in A$. Observe that $D_A = c_A^{-1}(\epsilon)$.

Let A and B be two finite alphabets such that $B \subseteq A$, we define the mapping $r_B : A^* \to B^*$ by $r_B(x_1 \ldots x_n) = y_1 \ldots y_n$ with $y_i = x_i$ if $x_i \in B$ and $y_i = \epsilon$ otherwise.

2.2 Pushdown Automata and Transducers

Formally, a *weighted pushdown transducer* (PDT) T over the tropical semiring $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$ is a 9-tuple $(\Sigma, \Delta, \Pi, \overline{\Pi}, Q, E, I, F, \rho)$ where Σ and Δ are the finite input and output alphabets, Π and $\overline{\Pi}$ are the finite open and close parenthesis alphabets, Q is a finite set of states, $I \in Q$ the initial state, $F \subseteq Q$ the set of final states, $E \subseteq Q \times (\Sigma \cup \widehat{\Pi} \cup \{\epsilon\}) \times (\Delta \cup \widehat{\Pi} \cup \{\epsilon\}) \times (\mathbb{R} \cup \{\infty\}) \times Q$ a finite set of transitions, and $\rho : F \to \mathbb{R} \cup \{\infty\}$ the final weight function. Let $e = (p[e], i[e], o[e], w[e], n[e])$ denote a transition in E we require that if $i[e] \in \widehat{\Pi}$ or $o[e] \in \widehat{\Pi}$, then $i[e] = o[e]$. We define the *size* of T as $|T| = |Q| + |E|$.

A path π is a sequence of transitions $\pi = e_1 \ldots e_n$ such that $n[e_i] = p[e_{i+1}]$ for $1 \leq i < n$. We then define $p[\pi] = p[e_1]$, $n[\pi] = n[e_n]$, $i[\pi] = i[e_1] \cdots i[e_n]$, $o[\pi] = o[e_1] \cdots o[e_n]$ and $w[\pi] = w[e_1] + \ldots + w[e_n]$.

A path π is accepting if $p[\pi] = I$ and $n[\pi] \in F$. A path π is balanced if $r_{\widehat{\Pi}}(i[\pi]) \in D_{\Pi}$. A balanced path π accepts the pair $(x, y) \in \Sigma^* \times \Delta^*$ if it is a balanced accepting path such that $r_{\Sigma}(i[\pi]) = x$ and $r_{\Delta}(o[\pi]) = y$.

The *weight associated by* T to a pair of strings $(x, y) \in \Sigma^* \times \Delta^*$ is

$$T(x, y) = \min_{\pi \in P(x,y)} w[\pi] + \rho(n[\pi])$$

where $P(x, y)$ denotes the set of balanced paths accepting (x, y). A weighted transduction is recognizable by a weighted pushdown transducer iff it is algebraic [**?**] or equivalently iff it is recognizable by a weighted simple syntax-directed translation [**?**,**?**].

A *weighted pushdown automaton* (PDA) is a pushdown transducer where $i[e] = o[e]$ for all transition $e \in E$. A weighted language is recognizable by a weighted pushdown automaton iff it is context-free [**?**,**?**].

A pushdown transducer T has *bounded stack* if there exists $K \in \mathbb{N}$ such that for any path π from I such that $c_{\Pi}(r_{\widehat{\Pi}}(i[\pi])) \in \Pi^*$:

$$|c_{\Pi}(r_{\widehat{\Pi}}(i[\pi]))| \leq K. \qquad (1)$$

If T has bounded stack, then it represents a rational transduction (see Section 4.1). Figure 1a-c gives examples of non-rational, rational and bounded-stack PDAs.

A pushdown transducer is *deterministic* if at any state with at least two outgoing transitions the input labels of the outgoing transitions are distinct and are either all input symbols (in Σ) or all close parentheses (in $\overline{\Pi}$).

A *weighted finite-state transducer or automaton* (FST or FSA) can be viewed as a PDT or PDA where the open and close parentheses alphabets are empty; see [**?**] for a stand-alone definition.

3 Implementation

The benefit of this definition of PDTs is that a PDT T can be represented as a pair of a FST specification, with input alphabet $\Sigma \cup \widehat{\Pi}$ and output alphabet $\Delta \cup \widehat{\Pi}$, and a parentheses mapping $f : \Pi \to \overline{\Pi}, a \mapsto \overline{a}$. This allows us to fully leverage the OpenFst library [**?**] for representing and manipulating the FST specifications of PDTs.

The PDA A_1 given in Figure 1a can be generated from the three text files given Figure 2. The `pda.txt` file is the textual description of the FSA specification of A_1 in the OpenFst format. The `symbols` file maps each symbol to an integer value used for the internal memory representation. Finally, the `parens` file describes the pair of open and close parentheses. The `fstcompile` binary command can be used to generate a binary file for the FSA specification of A_1:

```
fstcompile --acceptor --isymbols=symbols pda.txt > pda.fst
```

pda.txt	symbols	parens
0 1 a	eps 0	3 4
0 2 eps	a 1	
1 0 (	b 2	
2 3)	(3	
2	) 4	
3 2 b		

Fig. 2. Text files representing the PDA from Figure 1a.

The pair of files (`pda.fst`, `parens`) is then the file representation of the PDA for the purposes of the library. For instance, the reverse of A_1 can then be computed by invoking the following command:

```
pdtreverse --pdt_parentheses=parens pda.fst > reverse-pda.fst
```

Using the C++ interface, a PDT is similarly represented by a pair consisting of an object of type `StdFst` and a `vector<pair<int, int> >` object representing the set of open and close parenthesis pairs. The following C++ code is equivalent to the command given above:

```
StdFst *pda = StdFst::Read("pda.fst");
vector<pair<int, int> > parens(1, make_pair(3,4));
StdVectorFst reverse_pda;
Reverse(*pda, parens, &reverse_pda);
```

Table 1 shows the operations available in the PDT library extension [**?**]. The shared file and memory representations for FSTs and PDTs allows some operations from the OpenFst library, such as Union or Invert for instance, to be applied to PDTs unmodified. Other operations can be implemented with minimal work by leveraging the corresponding FST operation. For instance, PDT reversal can be implemented by first calling the Reverse operation of OpenFst followed by replacing every occurence of a parenthesis $a \in \widehat{\Pi}$ by its matching parenthesis $\overline{a}$ in the resulting machine.

4 Algorithms

In this section, we present PDT algorithms that are not trivially derived from FST analogues. The algorithms that we chose were motivated by analogy to the finite automata or context-free grammar case, by their applications (see Section 5), and by their tractability.

4.1 Expansion

Given a bounded-stack PDT T, the *expansion* of T is the FST T' equivalent to T defined as follows.

A state in T' is a pair (q, z) where q is a state in T and $z \in \Pi^*$. A transition (q, a, b, w, q') in T results in a transition $((q, z), a', b', w, (q', z'))$ in T' only when one of the following conditions hold: (a) $a \in \Sigma \cup \{\epsilon\}$, $z' = z$, $a' = a$ and $b' = b$, (b)

Table 1. Algorithms for manipulating pushdown transducers and the corresponding binary commands.

Operation	Algorithm	Section	Command
Union	FST alg.		`fstunion`
Concatenation	FST alg.*		`fstconcat`
Closure	FST alg.*		`fstclosure`
Reversal	trivial changes to FST alg.		`pdtreverse`
Inversion	FST alg.		`fstinvert`
Projection	FST alg.		`fstproject`
Expansion	PDT-specific alg.$^\diamond$	4.1	`pdtexpand`
Replacement	PDT-specific alg.	4.5	`pdtreplace`
Composition	non-trivial changes to FST alg.	4.2	`pdtcompose`
Determinization	FST alg. useful†		`fstdeterminize`
Epsilon removal	FST alg.		`fstrmepsilon`
Minimization	FST alg. useful‡		`fstminimize`
Shortest distance	PDT-specific alg.$^\diamond$	4.3	N/A
Shortest path	PDT-specific alg.$^\diamond$	4.3	`pdtshortestpath`
Pruned expansion	PDT-specific alg.$^\diamond$	4.4	`pdtexpand`
Pruning	PDT-specific alg. required	4.6	N/A
Connection	PDT-specific alg. required	4.6	N/A

*Assumes the presence of distinguished initial and final parentheses.
$^\diamond$Requires bounded-stack input.
†Reduces the redundancy but does not produce a deterministic PDT.
‡Reduces the size but does not perform PDT minimization.

$a \in \Pi$, $z' = za$, $a' = \epsilon$ and $b' = \epsilon$, or (c) $a \in \overline{\Pi}$, $z = z'\overline{a}$, $a' = \epsilon$ and $b' = \epsilon$. The initial state of T' is $I' = (I, \epsilon)$. A state (q, z) in T' is final iff q is final in T and $z = \epsilon$. We have $\rho'((q, \epsilon)) = \rho(q)$. The set of states of T' is the set of pairs (q, z) that can be reached from an initial state by transitions defined as above. The condition that T has bounded stack ensures that this set is finite (since it implies that for any such pair (q, z), $|z| \leq K$).

The complexity of the algorithm is linear in $O(|T'|) = O(e^{|T|})$. Figure 1d shows the result of the algorithm when applied to the PDA of Figure 1c.

4.2 Composition

The class of weighted pushdown transducers is closed under composition with weighted finite-state transducers [**?**,**?**]. Considering a pair (T_1, T_2) where one element is an FST and the other element a PDT and such that T_1 has input and output alphabets Σ and Δ and T_2 has input and output alphabets Δ and Γ, then there exists a PDT $T_1 \circ T_2$, the *composition* of T_1 and T_2, such that for all $(x, y) \in \Sigma^* \times \Gamma^*$: $(T_1 \circ T_2)(x, y) = \min_{z \in \Delta^*}(T_1(x, z) + T_2(z, y))$. We assume in the following that T_2 is an FST. We also assume that T_2 has no input-ϵ transitions. When T_2 has input-ϵ transitions, an epsilon filter [**?**,**?**] generalized to handle parentheses can be used.

```
ShortestDistance($T$)
1  for each $q \in Q$ and $a \in \Pi$ do
2     $B[q, a] \leftarrow \emptyset$
3  GetDistance($T, I$)
4  return $d[f, I]$

Relax($q, s, w, \mathcal{S}$)
1  if $d[q, s] > w$ then
2     $d[q, s] \leftarrow w$
3     if $q \notin \mathcal{S}$ then
4        Enqueue($\mathcal{S}, q$)
```

```
GetDistance($T, s$)
 1  for each $q \in Q$ do
 2     $d[q, s] \leftarrow \infty$
 3  $d[s, s] \leftarrow 0$
 4  $\mathcal{S}_s \leftarrow s$
 5  while $\mathcal{S}_s \neq \emptyset$ do
 6     $q \leftarrow$ Head($\mathcal{S}_s$)
 7     Dequeue($\mathcal{S}_s$)
 8     for each $e \in E[q]$ do
 9        if $i[e] \in \Sigma \cup \{\epsilon\}$ then    ▷ $i[e]$ is a regular symbol
10           Relax($n[e], s, d[q, s] + w[e], \mathcal{S}_s$)
11        elseif $i[e] \in \overline{\Pi}$ then        ▷ $i[e]$ is a close parenthesis
12           $B[s, \overline{i[e]}] \leftarrow B[s, \overline{i[e]}] \cup \{e\}$
13        elseif $i[e] \in \Pi$ then        ▷ $i[e]$ is an open parenthesis
14           if $d[n[e], n[e]]$ is undefined then
15              GetDistance($T, n[e]$)
16           for each $e' \in B[n[e], i[e]]$ do
17              $w \leftarrow d[q, s] + w[e] + d[p[e'], n[e]] + w[e']$
18              Relax($n[e'], s, w, \mathcal{S}_s$)
```

Fig. 3. PDT shortest distance algorithm. We assume that $F = \{f\}$ and $\rho(f) = 0$ to simplify the presentation

A state in $T = T_1 \circ T_2$ is a pair (q_1, q_2) where q_1 is a state of T_1 and q_2 a state of T_2. The initial state is $I = (I_1, I_2)$. Given a transition $e_1 = (q_1, a, b, w_1, q_1')$ in T_1, transitions out of (q_1, q_2) in T are obtained using the following rules.

If $b \in \Delta$, then e_1 can be matched with a transition (q_2, b, c, w_2, q_2') in T_2 resulting a transition $((q_1, q_2), a, c, w_1 + w_2, (q_1', q_2'))$ in T. If $b = \epsilon$, then e_1 is handled by staying in q_2 resulting in a transition $((q_1, q_2), a, \epsilon, w_1, (q_1', q_2))$. Finally, if $b = a \in \widehat{\Pi}$, e_1 is also handled by staying in q_2, resulting in a transition $((q_1, q_2), a, a, w_1, (q_1', q_2))$ in T.

A state (q_1, q_2) in T is final when both q_1 and q_2 are final, and then $\rho((q_1, q_2)) = \rho_1(q_1) + \rho_2(q_2)$. The complexity of the algorithm is $O(|T_1|\ |T_2|)$ in the worst case.

4.3 Shortest Distance and Shortest Path

A *shortest path* in a PDT T is a balanced accepting path with minimal weight and the *shortest distance* in T is the weight of such a path. We show that when T has bounded stack, the shortest distance and shortest path can be computed in $O(|T|^3 \log |T|)$ time (assuming T has no negative weights) and $O(|T|^2)$ space.

Given a state s in T with at least one incoming open parenthesis transition, we denote by C_s the set of states that can be reached from s by a balanced path. If s has several incoming open parenthesis transitions, a naive implementation might lead to the states in C_s being visited up to exponentially many times. The basic idea of the algorithm is to memoize the shortest distance from s to states in C_s. The pseudo-code is given in Figure 3.

GetDistance(T, s) starts a new instance of the shortest-distance algorithm from s using the queue $\mathcal{S}_s$, initially containing s. While the queue is not empty, a state is dequeued and its outgoing transitions examined (line 5-9). Transitions labeled by non-parenthesis are treated as in Mohri [?] (line 9-10). When the considered transition e is labeled by a close parenthesis, all balancing incoming open parentheses in s labeled by $\overline{i[e]}$ are remembered by adding e to $B[s, \overline{i[e]}]$ (line 11-12). Finally, when e is labeled with an open parenthesis, if its destination has not already been visited, a new instance is started from $n[e]$ (line 14-15). The destination states of all transitions balancing e are then relaxed (line 16-18).

The space complexity of the algorithm is quadratic for two reasons. First, the number of non-infinite $d[q, s]$ is $|Q|^2$. Second, the space required for storing B is at most in $O(|E|^2)$ since for each open parenthesis transition e, the size of $|B[n[e], i[e]]|$ is $O(|E|)$ in the worst case. This last observation also implies that the accumulated number of transitions examined at line 16 is in $O(N|Q|\ |E|^2)$ in the worst case, where N denotes the maximal number of times a state is inserted in the queue for a given call of GetDistance. Assuming the cost of a queue operation is $\Gamma(n)$ for a queue containing n elements, the worst-case time complexity of the algorithm can then be expressed as $O(N|T|^3\ \Gamma(|T|))$. When T contains no negative weights, using a shortest-first queue discipline leads to a time complexity in $O(|T|^3 \log |T|)$. When all the C_s's are acyclic, using a topological order queue discipline leads to a $O(|T|^3)$ time complexity.

When T has been obtained by converting an RTN into a PDA (see Section 4.5), the polynomial dependency in $|T|$ becomes a linear dependency both for the time and space complexities. Indeed, for each q in T, there exists a unique s such that $d[q, s]$ is non-infinite. Moreover, for each open parenthesis transition e, there exists a unique close parenthesis transition e' such that $e' \in B[n[e], i[e]]$. When each component of the RTN is acyclic, the complexity of the algorithm is hence in $O(|T|)$ in time and space.

Similarly, when $T = T_1 \circ T_2$ and T_1 was obtained by converting an RTN into a PDA, the complexity becomes $O(N|T_1||T_2|^3\ \Gamma(|T|))$ in time and $O(|T_1||T_2|^2)$ in space. This follows since for each (q_1, q_2) there exists a unique s_1 such that $d[(q_1, q_2), (s_1, s_2)]$ is non-infinite. Also, for each open parenthesis transition e, there exist at most $|T_2|$ close parenthesis transition e' such that $e' \in B[n[e], i[e]]$.

The algorithm can be modified (without changing the complexity) to compute the shortest path through T by keeping track of parent pointers.

4.4 Pruned Expansion

Given a bounded-stack PDT T, the *pruned expansion* of T with threshold β is an FST T'_β obtained by deleting from T' all states and transitions that belong to no accepting path π in T' such that $\lambda'(p[\pi]) + w[\pi] + \rho'(n[\pi]) \leq d + \beta$ where d is the shortest distance in T. A naive implementation consisting of fully expanding T and then applying the FST pruning algorithm would lead to a complexity in $O(|T'| \log |T'|) = O(e^{|T|}|T|)$.

Assuming that the reverse T^R of T is also bounded-stack, an algorithm whose complexity is in $O(|T|\ |T'_\beta| + |T|^3 \log |T|)$ can be obtained by first applying the shortest distance algorithm from the previous section to T^R and then using this to prune the expansion as it is generated. When invoking the `pdtexpand` command, the `--weight` flag can be used to specify the threshold β and trigger a pruned expansion of the input PDT.

4.5 Replacement

A *recursive transitive network* (RTN) R is specified by $(N, \Sigma, \Delta, (T_\nu)_{\nu \in N}, S)$ where N is an alphabet of nonterminals, Σ and Δ are the input and output alphabets, $(T_\nu)_{\nu \in N}$ is a family of FSTs with input alphabet $\Sigma \cup N$ and output alphabet Δ, and $S \in N$ is the root nonterminal.

A pair $(x, y) \in \Sigma^* \times \Delta^*$ is accepted by R if there exists an accepting path π in T_S such that recursively replacing any transition with input label $\nu \in N$ by an accepting path in T_ν leads to a path π^* with input x and output y. The weight associated by R is the minimum over all such π^* of $w[\pi^*] + \rho_S(n[\pi^*])$.

Given an RTN R, the *replacement* of R is the PDT T equivalent to R defined by the 10-tuple $(\Sigma, \Delta, \Pi, \overline{\Pi}, Q, E, I, F, \sigma, \rho)$ with $\Pi = Q = \bigcup_{\nu \in N} Q_\nu$, $I = I_S$, $F = F_S$, $\rho = \rho_S$, and $E = \bigcup_{\nu \in N} \bigcup_{e \in E_\nu} E^e$ where $E^e = \{e\}$ if $i[e] \notin N$ and otherwise $E^e = \{(p[e], n[e], \epsilon, w[e], I_\mu), (f, \overline{n[e]}, \epsilon, \rho_\mu(f), n[e]) | f \in F_\mu\}$ with $\mu = i[e] \in N$.

The complexity of the construction is in $O(|T|)$. If $|F_\nu| = 1$, then $|T| = O(\sum_{\nu \in N} |T_\nu|) = O(|R|)$. Creating a superfinal state for each T_ν would lead to a T whose size is always linear in the size of R.

4.6 Discussion

The PDT expansion algorithm can result in an FST that is not *trim*: it may contain useless states or transitions not on accepting paths. OpenFst provides the `Connect` operation that performs classical finite-automata trimming (using a depth-first search). By analogy, a PDT can be defined trim if each state and transition lies on a balanced, accepting path. Similarly, a PDT can be defined *pruned with threshold* β if each state and transition lies on a balanced, accepting path with weight $w \leq d + \beta$ where d is the shortest distance in the PDT. In the future, we wish to add algorithms `Connect` to trim a bounded-stack PDT and `Prune` to prune a bounded-stack PDT within threshold β. Note these algorithms are different from the connected or pruned expansion of a PDT, since the results here, in general, are PDTs not FSTs.

5 Applications

5.1 Recognition

Suppose we have an acyclic weighted finite automaton L that represents the likelihood $Pr[x|s]$ of some observation x given a sentence $s \in L$. For example, x could be spoken or written words with $Pr[x|s]$ being acoustically or optically-derived likelihoods from an automatic speech recognition (ASR) or optical character recognition (OCR) system. Further, suppose we have a weighted context-free grammar G that represents the a priori probability $Pr[s]$ of each sentence in the grammar. We wish to compute the maximum a posteriori probability sentence, $\underset{s}{\operatorname{argmax}}\, Pr[x|s]Pr[s]$, given L and G.

To do so, we will first represent G as a pushdown automaton. A weighted context-free grammar (CFG) can be specified by (N, Σ, P, S) where N is an

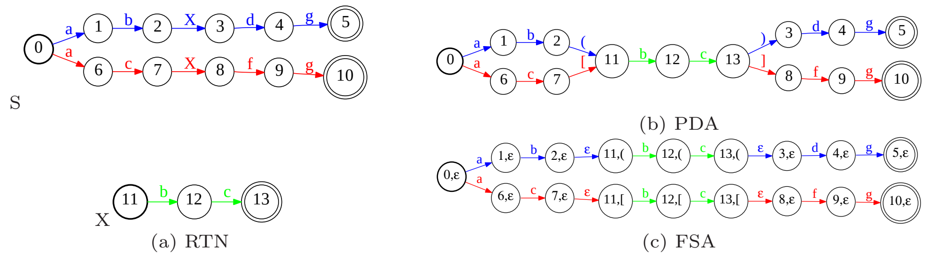


Fig. 4. Automata representations

alphabet of nonterminals, Σ is an alphabet of terminals, $P \subseteq N \times (N \cup \Sigma)^* \times (R \cup \{\infty\})$ are productions and S is the start symbol. A production (ν, α, w) is sometimes written as $\nu \to \alpha/w$.

To create a PDA that represents G, use each production (ν, α, w) to create the linear FSA $A_{\nu,\alpha,w}$ that accepts α with weight w. Then for each non-terminal ν, form the finite-state union $T_\nu = \cup_{(\nu,\alpha,w)\in P} A_{\nu,\alpha,w}$. Then $(N, \Sigma, \Sigma, (T_\nu)_{\nu\in N}, S)$ is an RTN R_G for which each accepting path π is in $1:1$ correspondence with a leftmost derivation of $i(\pi)$ in G [**?**]. Finally, use the construction in Section 4.5 to represent R_G as a PDA T_G.

For example, consider the context-free grammar: $S \to abXdg$, $S \to acXfg$ and $X \to bc$. Figure 4 shows several automata representations of this grammar. Figure 4a shows the RTN representation of this grammar with a 1:1 correspondence between each production in the CFG and each accepting path in the RTN components. Figure 4b shows the pushdown automaton representation generated from the RTN with the replacement algorithm of Section 4.5. Since this grammar's productions have no cyclic dependencies, the PDA has bounded stack and represents a regular language. Figure 4c shows the finite-state automaton representation of this grammar generated by the PDA using the expansion algorithm of Section 4.1.

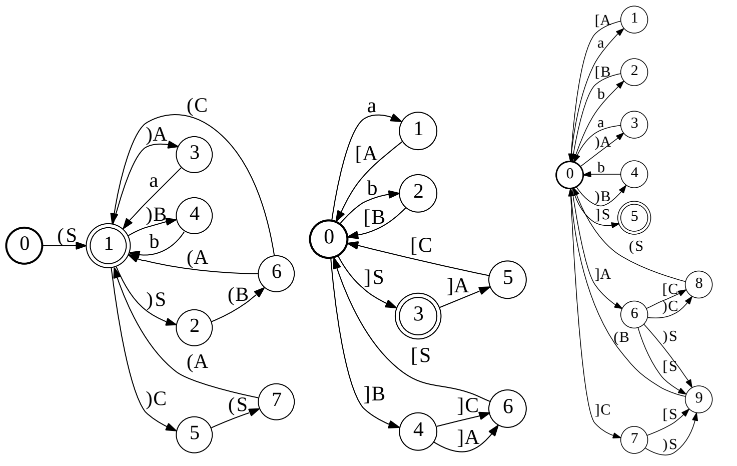
For the probabilistic recognition example, we use negative log probabilities in the weighted finite automaton L and in the construction of the PDT T_G that represents CFG G. Then, the maximum a posteriori sentence can be found with $ShortestPath(L \cap T_G)$. With the command line operations, this becomes:

```
pdtcompose --pdt_parentheses=parens G.pda L.fsa |
pdtshortestpath --pdt_parentheses=parens > Map.fsa
```

since composition between acceptors is intersection.[1] The recognition has time complexity in $O(|L|^3|T_G|)$ and space complexity in $O(|L|^2|T_G|)$ since T_G has bounded stack and is derived from an RTN.

An advantage of the RTN, PDA, and FSA representations is that they can benefit from FSA epsilon removal, determinization and minimization algorithms

[1] The compostion flag `--left_pdt=false` would be required if the arguments were exchanged.



(a) left parser (b) right parser (c) left corner parser

Fig. 5. Different parsing strategies using PDTs.

applied to their components (for RTNs and PDAs) or their entirety (for FSAs). These steps could improve the time and space requirements of the recognition example.

In a real-world example, this approach essentially is used to identify *voice action* queries in the Google Android speech platform. For example, a production could be $S \rightarrow$ `send a message from` X `to` Y where the non-terminals X and Y, for the sender and recipient, are rewritten as people's names. A match identifies a voice query as a messaging action.

5.2 Parsing

In the final example in the last section, we might not only wish to identify a messaging action in a voice query but also want to *parse* the input to find where the sender and recipient names are located. This is very similar to CFG recognition but with the output augmented with the parse bracketing. A classical approach is to augment the output tape of the PDT to include an index for each production [**?**]. We take another approach here: the parentheses are chosen to identify the production (or non-terminal) and the parentheses are retained in the shortest path output. With the command line operations, this is done with the flag `--keep_parentheses`. This does not increase the time or space complexity over recognition.

It has long been known that PDTs can be used to parse and that different parsing *strategies* can be achieved by compiling the CFG into different PDTs [**?**,**?**]. For example, the CFG: $S \rightarrow AB$, $S \rightarrow CB$, $C \rightarrow AS$, $A \rightarrow a$ and $B \rightarrow b$can be left parsed ('top-down') by the PDT in Figure 5a, right parsed ('bottom-up') by the PDT in Figure 5b, and left-corner parsed by the PDT in Figure 5c

[?]. Note an equivalent right parser can be obtained from the left parser by first reversing the right-hand side of the productions and then reversing the transducer.

The classical method to apply these parsers is equivalent to intersecting the PDT with the input string followed by the exponential expansion algorithm of Section 4.1. Lang [?] showed that the cubic tabular method of Earley can be naturally applied to PDTs; others give the weighted generalizations [?,?]. These approaches are closely related to intersecting the PDT with the input string followed by the shortest path algorithm of Section 4.3.

5.3 Translation

Hierarchical phrase-based translation, using a *synchronous context-free translation grammar* (SCFG) G together with an n-gram target language model M, is a popular approach in machine translation [?]. The productions of the SCFG are of the form $S \rightarrow \langle uAvBw, xByAz \rangle$. This production says that $uAvBw$ translates to $xByAz$ where u, v, w, x, y, z are terminal strings and A and B are non-terminals that must be in $1:1$ correspondence in the source and target of the translation but not necessarily in the same order. If all the productions preserved this order, it would be possible to represent the translation grammar as a pushdown *transducer* but for a general SCFG this is not possible [?].

However, the result of the application of the input source string s to the probabilistic translation grammar G, which represents all possible translations of s by G, is compactly represented by a weighted RTN or PDA $T_{s,G}$ [?] [2]. It has bounded-stack, since the input s has already been applied to the SCFG.

Applying the n-gram language model M to $T_{s,G}$ and searching for the best resulting translation, typically the computationally expensive steps in translation, becomes $ShortestPath(T_{s,G} \cap M)$. It has time complexity in $O(|T_{s,G}||M|^3)$ and space complexity in $O(|T_{s,G}||M|^2)$ since $T_{s,G}$ has bounded stack and is derived from an RTN. An alternative approach first expands $T_{s,G}$ to an FSA $F_{s,G}$ and then applies finite-state intersection and shortest path to give a time and space complexity of $O(|e^{|F_{s,G}|}|M|)$. Gonzalo, et al [?] give experimental results comparing these two approaches on a range of grammar and n-gram language model sizes in a large-scale English-Chinese translation system.

5.4 Discussion

For each of these tasks - recognition, parsing, or translation - real-world problems might involve very large CFGs. In these cases, the cubic complexity of the shortest path algorithm may be prohibitive and *inadmissable* or *inexact* methods may be used that are not guaranteed to return the shortest path. One general approach is to prune away unpromising paths [?,?]. Another approach is to use a weaker, smaller grammar in a first pass, output a hypothesis set, and rescore that with the full grammar. For the latter method, the pruned expansion of Section 4.4 can be used to output the hypothesis sets.

[2] Another related representation, *hypergraphs*, are also often used for this purpose [?].

Acknowledgments

We thank Mehryar Mohri for suggesting a PDT algorithms library and discussions and thank Bill Byrne, Adrià de Gispert and Gonzalo Iglesias for working with us to adapt their pioneering automata approach for machine translation to PDTs along with their comprehensive evaluations of these methods.