Algorithms for Weighted Pushdown Automata

Alexandra Butoi[1] **Brian DuSell**[2] **Tim Vieira**[3]
Ryan Cotterell[1] **David Chiang**[2]

[1]ETH Zürich [2]University of Notre Dame [3]Johns Hopkins University

alexandra.butoi@inf.ethz.ch {bdusell1, dchiang}@nd.edu
{tim.f.vieira, ryan.cotterell}@gmail.com

Abstract

Weighted pushdown automata (WPDAs) are at the core of many natural language processing tasks, like syntax-based statistical machine translation and transition-based dependency parsing. As most existing dynamic programming algorithms are designed for context-free grammars (CFGs), algorithms for PDAs often resort to a PDA-to-CFG conversion. In this paper, we develop novel algorithms that operate directly on WPDAs. Our algorithms are inspired by Lang's algorithm, but use a more general definition of pushdown automaton and either reduce the space requirements by a factor of $|\Gamma|$ (the size of the stack alphabet) or reduce the runtime by a factor of more than $|Q|$ (the number of states). When run on the same class of PDAs as Lang's algorithm, our algorithm is both more space-efficient by a factor of $|\Gamma|$ and more time-efficient by a factor of $|Q| \cdot |\Gamma|$.

https://github.com/rycolab/wpda

1 Introduction

Weighted pushdown automata (WPDAs) are widespread in natural language processing (NLP), primarily in syntactic analysis. For instance, WPDAs have found use in syntax-based statistical machine translation (Allauzen et al., 2014), and many transition-based dependency parsers (Nivre, 2004; Chen and Manning, 2014; Weiss et al., 2015; Dyer et al., 2015; Andor et al., 2016; Shi et al., 2017; Ma et al., 2018; Fernández-González and Gómez-Rodríguez, 2019) are special cases of WPDAs. In addition, PDAs have been used in computational psycholinguistics as models of human sentence processing (Resnik, 1992). Despite their ubiquity, there has been relatively little research on the theory of WPDAs themselves. In some ways, WPDAs are treated as second-class citizens compared to their equivalent cousins, weighted context-free grammars (WCFGs), for which a variety of dynamic programming algorithms exists (Bar-Hillel et al., 1961; Earley, 1970; Stolcke, 1995). To help fill this gap, this paper offers several new and improved algorithms for computing with WPDAs.

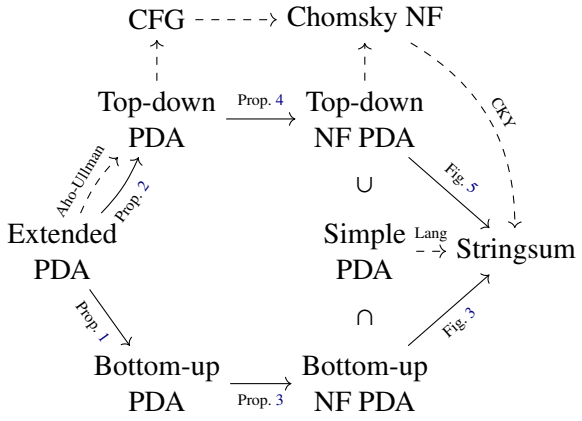

Figure 1: Roadmap of the paper. Solid lines are new results in this paper; dashed lines are old results. We are aware of two existing methods for PDA stringsums, via CFG and via Lang's algorithm; our algorithms are faster and/or more general than both.

Figure 1 gives an overview of most of our results. We start by defining a weighted version of the extended PDAs of Aho and Ullman (1972, p. 173) and two special cases: the standard definition (Hopcroft et al., 2006), which we call top-down, and its mirror image, which we call bottom-up. Both top-down and bottom-up WPDAs have been used in NLP. Roark's (2001) generative parser is a top-down PDA as is Dyer et al.'s (2016). Most transition-based dependency parsers, both arc-standard (Nivre, 2004; Huang et al., 2009) and arc-eager (Nivre, 2003; Zhang and Clark, 2008), are bottom-up WPDAs.

Next, we give a normal form for WPDAs analogous to Chomsky normal form, and we derive new dynamic programming algorithms to compute the weight of a string under top-down and bottom-up WPDAs in normal form. We are only aware of one

previous recognition algorithm for PDAs, that of Lang (1974), which we generalize to the weighted case and improve in the following ways:

- On PDAs more general than those Lang considers, our algorithm is more space-efficient by a factor of $|\Gamma|$ (the stack alphabet size);
- We can speed up our algorithm to be more time-efficient by a factor of more than $|Q|$ (the number of states), but without the space-complexity improvement;
- On the same PDAs that Lang considers, which we call **simple**, our sped-up algorithm is more efficient by a factor of $|\Gamma|$ in space and $|Q| \cdot |\Gamma|$ in time.

Compared with the pipeline of standard procedures for converting a top-down PDA to a CFG, converting to Chomsky normal form, and parsing with CKY, our top-down algorithm is faster by a factor of more than $O(|Q|^3)$.

Finally, we present iterative algorithms for computing the total weight of all runs of a WPDA.

2 Weighted Pushdown Automata

2.1 Preliminaries

Let $[i{:}j]$ denote the sequence of integers $(i, \ldots, j)$. If $\boldsymbol{s}$ is a string, we write $|\boldsymbol{s}|$ for the length of $\boldsymbol{s}$, $\boldsymbol{s}_i$ for the i^{th} symbol of $\boldsymbol{s}$, and $\boldsymbol{s}(i{:}j]$ for the substring $\boldsymbol{s}_{i+1} \cdots \boldsymbol{s}_j$.

Definition 1. *A **monoid** is a tuple $(A, \odot, \boldsymbol{1})$, where A is a set, $\odot$ is an associative binary operation, and $\boldsymbol{1} \in A$, called the **identity** element, satisfies $\boldsymbol{1} \odot a = a \odot \boldsymbol{1} = a$ for all $a \in A$. If $a \odot b = b \odot a$ for all a, b, we say that the monoid is **commutative**.*

Definition 2. *A **semiring** is a tuple $\mathcal{W} = (A, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ such that $(A, \oplus, \boldsymbol{0})$ is a commutative monoid and $(A, \otimes, \boldsymbol{1})$ is a monoid. Additionally, $\otimes$ distributes over $\oplus$, that is, $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$ and $(a \oplus b) \otimes c = a \otimes c \oplus b \otimes c$, and $\boldsymbol{0}$ is absorbing with respect to $\otimes$, that is, $\boldsymbol{0} \otimes a = a \otimes \boldsymbol{0} = \boldsymbol{0}$. If $\otimes$ is commutative then we say that $\mathcal{W}$ is **commutative**.*

We also sometimes assume $\mathcal{W}$ is **continuous**; please see the survey by Droste and Kuich (2009) for a definition.

2.2 Definition

Our definition of weighted PDA is more general than usual definitions, in order to accommodate the top-down and bottom-up variants introduced in §2.3. It is roughly a weighted version of extended PDAs of Aho and Ullman (1972, p. 173) and the PDAs of Lewis and Papadimitriou (1997, p. 131).

Definition 3. *A **weighted pushdown automaton** (WPDA) over a semiring $\mathcal{W} = (A, \oplus, \otimes, \boldsymbol{0}, \boldsymbol{1})$ is a tuple $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, (\iota, \boldsymbol{\gamma}_I), (f, \boldsymbol{\gamma}_F))$, where:*

- *Q is a finite set of states;*
- *Σ is a finite set of input symbols, called the input alphabet;*
- *Γ is a finite set of stack symbols, called the stack alphabet;*
- *$\delta \colon Q \times \Gamma^* \times (\Sigma \cup \{\varepsilon\}) \times Q \times \Gamma^* \to A$ is called the transition weighting function;*
- *$(\iota, \boldsymbol{\gamma}_I)$ is called the initial configuration and $(f, \boldsymbol{\gamma}_F)$ is called the final configuration, where $\iota, f \in Q$ and $\boldsymbol{\gamma}_I, \boldsymbol{\gamma}_F \in \Gamma^*$.*

Stacks are represented as strings over Γ, from bottom to top. Thus, in the stack $\boldsymbol{\gamma} = X_1 X_2 \cdots X_n$, the symbol X_1 is at the bottom of the stack, while X_n is at the top.

Definition 4. *A **configuration** of a WPDA is a pair $(q, \boldsymbol{\gamma})$, where $q \in Q$ is the current state and $\boldsymbol{\gamma} \in \Gamma^*$ is the current contents of the stack.*

The initial and final configurations of a WPDA are examples of configurations; it is possible to generalize the initial and final stacks to (say) regular expressions over Γ, but the above definition suffices for our purposes.

A WPDA moves from configuration to configuration by following transitions of the form $q, \boldsymbol{\gamma}_1 \xrightarrow{a/w} r, \boldsymbol{\gamma}_2$, which represents a move from the state q to state r, while popping the sequence of symbols $\boldsymbol{\gamma}_1 \in \Gamma^*$ from the top of the stack and pushing the sequence $\boldsymbol{\gamma}_2 \in \Gamma^*$.

Definition 5. *If $\delta(p, \boldsymbol{\gamma}_1, a, q, \boldsymbol{\gamma}_2) = w$, then we usually write $\delta(p, \boldsymbol{\gamma}_1 \xrightarrow{a} q, \boldsymbol{\gamma}_2) = w$ or that δ has transition $(q, \boldsymbol{\gamma}_1 \xrightarrow{a/w} p, \boldsymbol{\gamma}_2)$. We sometimes let τ stand for a transition, and we define $\delta(\tau) = w$. We say that τ **scans** a, and if $a \neq \varepsilon$, we call τ **scanning**; otherwise, we call it **non-scanning**. We say that τ is k-pop, l-push if $|\boldsymbol{\gamma}_1| = k$ and $|\boldsymbol{\gamma}_2| = l$.*

Definition 6. *If $(q_1, \boldsymbol{\gamma}\boldsymbol{\gamma}_1)$ and $(q_2, \boldsymbol{\gamma}\boldsymbol{\gamma}_2)$ are configurations, and τ is a transition $q_1, \boldsymbol{\gamma}_1 \xrightarrow{a/w} q_2, \boldsymbol{\gamma}_2$, we write $(q_1, \boldsymbol{\gamma}\boldsymbol{\gamma}_1) \Rightarrow_\tau (q_2, \boldsymbol{\gamma}\boldsymbol{\gamma}_2)$.*

Definition 7. *A **run** of a WPDA $\mathcal{P}$ is a sequence of configurations and transitions*

$$\boldsymbol{\pi} = (q_0, \boldsymbol{\gamma}_0), \tau_1, (q_1, \boldsymbol{\gamma}_1), \ldots, \tau_n, (q_n, \boldsymbol{\gamma}_n)$$

where, for $i = 1, \ldots, n$, we have $(q_{i-1}, \boldsymbol{\gamma}_{i-1}) \Rightarrow_{\tau_i} (q_i, \boldsymbol{\gamma}_i)$. (Sometimes it will be convenient to treat $\boldsymbol{\pi}$ as a sequence of only configurations or only

transitions.) A run is called ***accepting*** *if* $(q_0, \boldsymbol{\gamma}_0)$ *is the initial configuration and* $(q_n, \boldsymbol{\gamma}_n)$ *is the final configuration. If, for* $i = 1, \ldots, n$, τ_i *scans* a_i, *then we say that* $\boldsymbol{\pi}$ *scans the string* $a_1 \cdots a_n$. *We write* $\Pi(\mathcal{P}, \boldsymbol{s})$ *for the set of runs that scan* $\boldsymbol{s}$ *and* $\Pi(\mathcal{P})$ *for the set of all accepting runs of* $\mathcal{P}$.

2.3 Subclasses of PDAs

Next, we define two special forms for WPDAs, which we call **top-down** and **bottom-up**, because they can be used as top-down and bottom-up parsers for CFGs, respectively. The most common definition of PDA (Hopcroft et al., 2006; Autebert et al., 1997) corresponds to top-down PDAs,[1] while the machine used in an LR parser (Knuth, 1965) corresponds to bottom-up PDAs.

Definition 8. *A WPDA is called* ***bottom-up*** *if it has only 1-push transitions. Moreover, the initial configuration is* (ι, ε) *and the final configuration is* (f, S) *for some* $\iota, f \in Q$ *and* $S \in \Gamma$.

Proposition 1. *Every WPDA is equivalent to some bottom-up WPDA.*

Proof. Add states ι', f' and a stack symbol S', and make (ι', ε) and (f', S') the new initial and final configurations, respectively. Add transitions

$$\iota', \varepsilon \xrightarrow{\varepsilon/\mathbf{1}} \iota, S'\boldsymbol{\gamma}_I$$
$$f, S'\boldsymbol{\gamma}_F \xrightarrow{\varepsilon/\mathbf{1}} f', S'.$$

For each k-pop, l-push transition $p, \boldsymbol{\gamma} \xrightarrow{a/w} r, X_1 \cdots X_l$ where $l > 1$, create $(l-1)$ new states $q_1, \ldots, q_{l-1}$ and replace the transition with

$$p, \varepsilon \xrightarrow{\varepsilon/\mathbf{1}} q_1, X_1$$
$$q_{i-1}, \varepsilon \xrightarrow{\varepsilon/\mathbf{1}} q_i, X_i \quad i = 2, \ldots, l-1$$
$$q_{k-1}, \boldsymbol{\gamma} \xrightarrow{a/w} r, X_k.$$

For each k-pop, 0-push transition $q, \boldsymbol{\gamma} \xrightarrow{a/w} p, \varepsilon$, replace it with the $(k+1)$-pop, 1-push transitions $q, X\boldsymbol{\gamma} \xrightarrow{a/w} p, X$ for every $X \in \Gamma \cup \{S'\}$. □

If the original WPDA had transitions that push at most l symbols, the resulting WPDA has $O(l \cdot |\delta| \cdot |Q|)$ states and $O((l + |\Gamma|) \cdot |\delta|)$ transitions.

[1] This definition goes back to Chomsky's (1963) original definition, which also allows 0-pop transitions.

Definition 9. *A WPDA is called* ***top-down*** *if it has only 1-pop transitions. Moreover, the initial configuration is* (ι, S) *and the final configuration is* (f, ε) *for some* $\iota, f \in Q$ *and* $S \in \Gamma$.

Proposition 2. *Every WPDA is equivalent to some top-down WPDA.*

Proof. Similar to the bottom-up case. □

This conversion crucially makes use of nondeterminism to guess the top k stack symbols. Aho and Ullman (1972, p. 174) give a different algorithm that uses the state to keep track of the top k stack symbols. Although this does not require nondeterminism, it creates $O(|\Gamma|^k \cdot |Q|)$ states.

Finally, Lang (1974) considers a still more restricted subclass of PDAs.[2]

Definition 10. *A WPDA is called* ***simple*** *if it only has* k*-pop,* l*-push transitions for* $k \leq 1$ *and* $l \leq 1$.

Because simple PDAs do not condition pushes on the top stack symbol, they can be weighted, but not probabilistic.

2.4 Stringsums and Allsums

Definition 11. *The* ***weight*** $\mathbf{w}(\boldsymbol{\pi})$ *of a run* $\boldsymbol{\pi} \in \Pi(\mathcal{P})$ *is the product of the weights of its transitions,*

$$\mathbf{w}(\boldsymbol{\pi}) \stackrel{\text{def}}{=} \bigotimes_{\tau \in \boldsymbol{\pi}} \delta(\tau).$$

Definition 12. *The* ***stringsum*** $\mathbf{w}(\mathcal{P}, \boldsymbol{s})$ *of a string* $\boldsymbol{s}$ *for a WPDA* $\mathcal{P}$ *is the total weight of all runs of* $\mathcal{P}$ *that scan* $\boldsymbol{s}$,

$$\mathbf{w}(\mathcal{P}, \boldsymbol{s}) \stackrel{\text{def}}{=} \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathcal{P}, \boldsymbol{s})} \mathbf{w}(\boldsymbol{\pi}).$$

Definition 13. *The* ***allsum*** $\mathbf{w}(\mathcal{P})$ *of a WPDA* $\mathcal{P}$ *is the total weight of all runs of* $\mathcal{P}$,

$$\mathbf{w}(\mathcal{P}) \stackrel{\text{def}}{=} \bigoplus_{\boldsymbol{\pi} \in \Pi(\mathcal{P})} \mathbf{w}(\boldsymbol{\pi}).$$

2.5 Push and Pop Computations

Our algorithms for bottom-up WPDAs make heavy use of **push computations**. Intuitively, a push computation is a run that pushes exactly one symbol without touching the stack symbols below it.

[2] This definition is also used by Sipser (2012) and seems to go back to Evey's (1963) original definition of PDAs, which doesn't allow 1-pop, 1-push transitions. The term "simple PDA" has been used at least twice for two different purposes (Schützenberger, 1963; Lewis and Papadimitriou, 1997); we apologize for introducing a third.

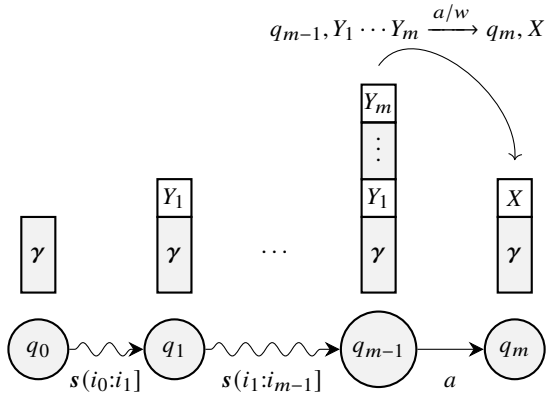

Figure 2: A push computation is a sequence of transitions that pushes exactly one symbol (X) without touching the stack symbols below (γ). The curly edges indicate sequences of transitions (which are themselves push computations) while the straight edge indicates a single transition.

Definition 14 (Push computation)**.** *Let $\mathcal{P}$ be a bottom-up WPDA and $s \in \Sigma^*$ an input string. A **push computation** of type $[i, p, X, j, q]$, where $0 \le i \le j \le |s|$, $p, q \in Q$, and $X \in \Gamma$, is a run $\pi = (q_0, \gamma_0), \ldots, (q_m, \gamma_m)$ that scans $s(i{:}j]$, where $\gamma_m = \gamma_0 X$, $q_0 = p$, $q_m = q$, and for all $l > 0$, $|\gamma_l| \ge |\gamma_m|$.*

Fig. 2 shows an example of a push computation. Notice that this *push* of X might be the result of possibly many transitions that can manipulate the stack. Every symbol other than X that is pushed onto the stack during this computation must be popped later by another transition.

The mirror image of a push computation is a **pop computation**, used in algorithms for top-down WPDAs; we defer its definition to §5.

3 Normal Form

In this section we present a series of semantics-preserving transformations for converting an arbitrary pushdown automaton into a normal form that is analogous to Chomsky normal form for context-free grammars. This will help us obtain a fast algorithm for computing stringsums.

Definition 15. *A bottom-up WPDA is in **normal form** if all of its scanning transitions are k-pop, 1-push for $k \le 2$, and all of its non-scanning transitions are 2-pop, 1-push. Similarly, a top-down WPDA is in **normal form** if all of its scanning transitions are 1-pop, k-push for $k \le 2$, and all of its non-scanning transitions are 1-pop, 2-push.*

3.1 Binarization

Recall that top-down and bottom-up WPDAs have 1-pop, k-push transitions and k-pop, 1-push transitions, respectively. Since the runtime of our string-sum algorithm depends highly on k, we convert the WPDA into an equivalent one with $k \le 2$. We call this procedure **binarization** because it is entirely analogous to binarization in CFGs. It is symmetric for top-down and bottom-up WPDAs.

Proposition 3. *Every bottom-up WPDA is equivalent to a bottom-up WPDA whose transitions are k-pop, 1-push where $k \le 2$.*

Proof. For each k-pop, 1-push transition $p, Y_1 \cdots Y_k \xrightarrow{a/w} q, X$ such that $k > 2$ we introduce $k-2$ new states $r_1, \cdots, r_{k-2}$ and we replace the original transition with the following:

$$p, Y_1 Y_2 \xrightarrow{a/w} r_1, Y_2$$
$$r_{i-1}, Y_i Y_{i+1} \xrightarrow{\varepsilon/\mathbf{1}} r_i, Y_{i+1} \quad i \in [2{:}k-2]$$
$$r_{k-2}, Y_{k-1} Y_k \xrightarrow{\varepsilon/\mathbf{1}} q, X.$$

□

If the original WPDA had transitions that pop at most k symbols, the resulting WPDA has $O(k \cdot |\delta| \cdot |Q|)$ states and $O(k \cdot |\delta|)$ transitions.

Proposition 4. *Every top-down WPDA is equivalent to a top-down WPDA whose transitions are 1-pop, k-push where $k \le 2$.*

3.2 Nullary Removal

In this section, we discuss the removal of **nullary** transitions from WPDAs:

Definition 16. *In a bottom-up WPDA, a transition is called **nullary** if it is of the form $p, \varepsilon \xrightarrow{\varepsilon/w} q, X$.*

Although nullary transitions are analogous to nullary productions in a CFG, the standard procedure for removing nullary productions from CFGs does not have an exact analogue for PDAs, and the procedure we describe here is novel.

We assume a bottom-up WPDA, but an identical construction exists for top-down WPDAs. We also assume that the WPDA has been binarized, and semiring $\mathcal{W}$ is commutative and continuous.

The construction consists of three steps: partitioning, precomputation, and removal.

Partitioning. For every symbol $X \in \Gamma$, we replace X with two stack symbols X^{ε} and $X^{\not\varepsilon}$. A push computation that pushes a X^{ε} scans ε, and a

push computation that pushes a $X^{\not\varepsilon}$ scans a string that is not ε. To do this, we replace every k-pop transition $p, X_1 \cdots X_k \xrightarrow{a/w} q, Y$ with 2^k new transitions $p, X_1^{\nu_1} \cdots X_k^{\nu_k} \xrightarrow{a/w} q, Y^{\nu}$, where $\nu = \varepsilon$ iff $\nu_i = \varepsilon$ for all i and $a = \varepsilon$. For instance, we replace transition $p, XY \xrightarrow{\varepsilon/w} q, Z$ with the following $2^2 = 4$ transitions

$$p, X^{\varepsilon}Y^{\varepsilon} \xrightarrow{\varepsilon/w} q, Z^{\varepsilon} \qquad p, X^{\not\varepsilon}Y^{\varepsilon} \xrightarrow{\varepsilon/w} q, Z^{\not\varepsilon}$$
$$p, X^{\varepsilon}Y^{\not\varepsilon} \xrightarrow{\varepsilon/w} q, Z^{\not\varepsilon} \qquad p, X^{\not\varepsilon}Y^{\not\varepsilon} \xrightarrow{\varepsilon/w} q, Z^{\not\varepsilon}.$$

Precomputation. We compute the weight of all non-scanning push computations by solving the quadratic system of equations:

$$\begin{aligned} N_{pXq} &= \delta(p, \varepsilon \xrightarrow{\varepsilon} q, X) \\ &\quad \oplus \bigoplus_{Y,r} N_{pYr} \otimes \delta(r, Y \xrightarrow{\varepsilon} q, X) \\ &\quad \oplus \bigoplus_{Y,Z,s} N_{pYZs} \otimes \delta(s, YZ \xrightarrow{\varepsilon} q, X) \\ N_{pYZs} &= \bigoplus_{r} N_{pYr} \otimes N_{rZs}. \end{aligned}$$

See §6 for details on solving such systems of equations, which assumes that $\mathcal{W}$ is continuous. Then N_{pXq} is the total weight of all push computations of type $[i, p, X, q, i]$ for any i.

Removal. First, delete every transition that pushes X^{ε} for each $X \in \Gamma$. If the PDA accepts ε with weight w, add $\iota, \varepsilon \xrightarrow{\varepsilon/w} f, S^{\varepsilon}$ as the sole nullary transition. (For correctness, we must also ensure that no transition pops S^{ε}, no transition enters ι, and no transition leaves f.)

Sometimes an X^{ε} is popped immediately after it is pushed (that is, with no input symbols scanned between the push and the pop). To handle these cases, for the following transitions, we create new versions in which popped X^{ε} symbols are removed, and their corresponding weight multiplied in.

For each:	Replace with ($\forall t \in Q$):
$p, Y^{\varepsilon} \xrightarrow{a/w} q, X^{\not\varepsilon}$	$t, \varepsilon \xrightarrow{a/N_{tYp} \otimes w} q, X^{\not\varepsilon}$
$p, Y^{\varepsilon}Z^{\varepsilon} \xrightarrow{a/w} q, X^{\not\varepsilon}$	$t, \varepsilon \xrightarrow{a/N_{tYZp} \otimes w} q, X^{\not\varepsilon}$
$p, Y^{\not\varepsilon}Z^{\varepsilon} \xrightarrow{a/w} q, X^{\not\varepsilon}$	$t, Y^{\not\varepsilon} \xrightarrow{a/N_{tZp} \otimes w} q, X^{\not\varepsilon}$

(Note that $a \in \Sigma \cup \{\varepsilon\}$, but the partitioning step only allows $a = \varepsilon$ for the third type above.)

However, we have missed one type of transition, those of the form $p, Y^{\varepsilon}Z^{\not\varepsilon} \xrightarrow{a/w} q, X^{\not\varepsilon}$. Create new stack symbols ${}_{rs}Z$ for all $r, s \in Q$ and $Z \in \Gamma$. This stands for a sequence of zero or more non-scanning push computations that goes from state r to s, followed by a push computation that pushes Z. The transition that pushes Z must be a 0-pop transition, because all other transitions expect a symbol of the form $X^{\not\varepsilon}$ on the top of the stack. So we modify (again) the 0-pop transitions to first simulate zero or more nullary transitions:

For each:	Replace with ($\forall s \in Q$):
$t, \varepsilon \xrightarrow{a/N_{tYp} \otimes w} q, X^{\not\varepsilon}$	$s, \varepsilon \xrightarrow{a/N_{tYp} \otimes w} q, {}_{st}X$
$t, \varepsilon \xrightarrow{a/N_{tYZp} \otimes w} q, X^{\not\varepsilon}$	$s, \varepsilon \xrightarrow{a/N_{tYZp} \otimes w} q, {}_{st}X$

And for each transition of the form $p, Y^{\varepsilon}Z^{\not\varepsilon} \xrightarrow{a/w} q, X^{\not\varepsilon}$ (where $a \in \Sigma \cup \{\varepsilon\}$), we create transitions for all $r, s, t \in Q$:

$$p, {}_{rt}Z \xrightarrow{a/N_{sYt} \otimes w} q, {}_{rs}X.$$

(This step is where commutativity is needed.) Finally, add transitions to remove the state annotations, for all p, X, q:

$$q, {}_{pp}X \xrightarrow{\varepsilon/\mathbf{1}} q, X^{\not\varepsilon}.$$

3.3 Unary Removal

The final step in conversion to normal form is removal of **unary** transitions, so called by analogy with unary productions in a CFG.

Definition 17. *A transition is called **unary** if it is of the form* $p, Y \xrightarrow{\varepsilon/w} q, X$.

We assume that $\mathcal{W}$ is equipped with a star operation satisfying $a^* = \mathbf{1} \oplus a \otimes a^* = \mathbf{1} \oplus a^* \otimes a$. If $\mathcal{W}$ is continuous, then $a^* = \bigoplus_{i=0}^{\infty} a^i$.

Unary transitions can form cycles that can be traversed an unbounded number of times, which is problematic for a dynamic programming algorithm. Therefore, we precompute the weights of all runs of unary transitions. Define the matrix $U \in \mathcal{W}^{(Q\times\Gamma)\times(Q\times\Gamma)}$:

$$U_{pY,qX} = \delta(p, Y \xrightarrow{\varepsilon} q, X)$$

and form its transitive closure U^* (Lehmann, 1977). Then $U^*_{pY,qX}$ is the total weight of all runs of unary transitions from configuration (p, Y) to (q, X).

Then we remove all unary transitions and modify every non-unary transition as follows:

For each non-unary:	Replace with:
$p, \boldsymbol{\gamma} \xrightarrow{a/w} q, X$	$p, \boldsymbol{\gamma} \xrightarrow{a/w \otimes U^*_{qX,rY}} r, Y$

We give details on the complexity of this transformation in App. A.

Item form

$$[i, p, X, j, q] \qquad 0 \le i \le j \le n,\ p, q \in Q; X \in \Gamma$$

Inference rules

$$\frac{[i, p, Y, k, r] \quad [k, r, Z, j-|a|, s]}{[i, p, X, j, q]} \qquad s, YZ \xrightarrow{a/w} q, X,\ \boldsymbol{s}(j-|a|{:}j] = a$$

$$\frac{[i, p, Y, j-1, r]}{[i, p, X, j, q]} \qquad r, Y \xrightarrow{\boldsymbol{s}_j/w} q, X$$

$$\frac{}{[i, p, X, j, q]} \qquad p, \varepsilon \xrightarrow{\boldsymbol{s}_j/w} q, X,\ j = i+1$$

Goal

$$[0, \iota, S, n, f]$$

Figure 3: Deductive system for stringsums of bottom-up WPDAs in normal form.

4 Stringsums in Bottom-up WPDAs

In this section, we give dynamic programming algorithms for computing the stringsum of an input string $\boldsymbol{s}$ (with $|\boldsymbol{s}| = n$) of bottom-up WPDAs in normal form. We give a basic version of the algorithm, which has the same runtime as Lang's algorithm but improved space requirements, and a fast version that has the same space complexity and runs asymptotically faster. On simple PDAs (for which Lang's algorithm was designed), the latter version has both improved space and time complexity.

4.1 Basic Algorithm

The algorithm computes stringsums efficiently by exploiting the structural similarities among the WPDA runs. Fig. 3 shows a deductive system (Shieber et al., 1995; Goodman, 1999) for deriving items corresponding to push computations.

The items have the form $[i, p, X, j, q]$ for $p, q \in Q$, $X \in \Gamma$, $0 \le i \le j \le n$. If our algorithm derives this item with weight w, then the push computations of type $[i, p, X, j, q]$ have total weight w.

We distinguish three categories of push computations, based on their final transition, and we include an inference rule for each. First are those consisting of a single 0-pop, 1-push transition. The other two categories are those ending in a 1-pop transition and a 2-pop transition, respectively. These can be built recursively from shorter push computations.

The goal item is $[0, \iota, S, n, f]$, which stands for all runs from the initial configuration to the final configuration that scan $\boldsymbol{s}$.

Alg. 1 shows how to compute item weights according to these rules. At termination, the weight of the goal item is the sum of the weights of all accepting runs that scan $\boldsymbol{s}$.

Algorithm 1 Compute the weights of all push computations of a bottom-up WPDA on an input string.

```
1.  w ← 0
2.  n ← |s|
3.  for i ∈ [0:n − 1] :
4.    j ← i + 1
5.    ▷0-pop, 1-push
6.    for (p, ε --s_j/w--> q, X) ∈ δ :
7.      w[i, p, X, j, q] ← w
8.  for ℓ ∈ [2:n] :
9.    for i ∈ [0:n − ℓ + 1] :
10.     j ← i + ℓ
11.     ▷1-pop, 1-push
12.     for p ∈ Q :
13.       for (r, Y --s_j/w--> q, X) ∈ δ :
14.         w[i, p, X, j, q] ⊕= w[i, p, Y, j−1, r] ⊗ w
15.     ▷2-pop, 1-push
16.     for p, r ∈ Q :
17.       for (s, YZ --a/w--> q, X) ∈ δ with s(j−|a|:j] = a :
18.         for k ∈ [i + 1:j − |a| − 1] :
19.           w[i, p, X, j, q] ⊕= (w[i, p, Y, k, r] ⊗
                  w[k, r, Z, j − |a|, s] ⊗ w)
20. return w[0, ι, S, n, f]
```

4.2 Correctness

Theorem 1. *Let $\mathcal{P}$ be a WPDA and $\boldsymbol{s} \in \Sigma^*$ an input string. The weight $\mathbf{w}[i, p, X, j, q]$ is the total weight of all push computations of $\mathcal{P}$ of type $[i, p, X, j, q]$.*

Proof. By induction on the span length, $\ell = j - i$.

Base Case. Assume that $j - i = 1$. The only push computations from state p to q that push X and scan $\boldsymbol{s}(i{:}j]$ are ones that have the single transition $\tau = p, \varepsilon \xrightarrow{\boldsymbol{s}_j/w} q, X$. There cannot exist others, because normal form requires that any additional non-scanning transitions would decrease the stack height. So the total weight of all such push computations is w, and the algorithm correctly sets $\mathbf{w}[i, p, X, j, q] = w$ at line 7.

Inductive Step. Assume that the statement holds for any spans of length at most $(\ell - 1)$ and consider a span of length ℓ. For such spans, the algorithm computes the total weight of all push computations $\boldsymbol{\pi}$ of type $[i, p, X, j, q]$, for all $X \in \Gamma$, $p, q \in Q$, and $j - i = \ell$. This weight must be the sum of weights of three types of push computations: those that end with 0-pop transitions, with 1-pop transitions, and with 2-pop transitions.

But ending with a 0-pop transition is impossible, because such push computations must have only

one transition and therefore $j - i \leq 1$. The 1-pop and 2-pop parts of the sum are computed at lines 12–14 and 16–19 of the algorithm, respectively.

Ending with 1-pop transition. The algorithm sums over all possible ending transitions $\tau_{\text{end}} = r, Y \xrightarrow{s_j/w} q, X$. (Normal form requires that this transition be scanning.) Let Π be the set of all push computations of type $[i, p, X, j, q]$ ending in τ_{end}, and let Π' be the set of all push computations of type $[i, p, Y, j-1, r]$. Every push computation in Π must be of the form $\boldsymbol{\pi} = \boldsymbol{\pi}' \circ \tau_{\text{end}}$, where $\boldsymbol{\pi}' \in \Pi'$, and conversely, for every $\boldsymbol{\pi}' \in \Pi'$, we have $\boldsymbol{\pi}' \circ \tau_{\text{end}} \in \Pi$. By the induction hypothesis, the total weight of Π' was computed in a previous iteration. Then, by distributivity, we have:

$$\begin{aligned}\bigoplus_{\boldsymbol{\pi}\in\Pi} \bigotimes_{\tau\in\boldsymbol{\pi}} \delta(\tau) &= \bigoplus_{\boldsymbol{\pi}'\in\Pi'} \bigotimes_{\tau\in\boldsymbol{\pi}'} \delta(\tau) \otimes \delta(\tau_{\text{end}}) \\ &= \left(\bigoplus_{\boldsymbol{\pi}'\in\Pi'} \bigotimes_{\tau\in\boldsymbol{\pi}'} \delta(\tau)\right) \otimes \delta(\tau_{\text{end}}) \\ &= \mathbf{w}[i, p, Y, j-1, r] \otimes \delta(\tau_{\text{end}}) .\end{aligned}$$

Ending with 2-pop transition. The algorithm sums over all possible ending transitions $\tau_{\text{end}} = s, YZ \xrightarrow{a/w} q, X, a \in \{s_j, \varepsilon\}$. Every push computation $\boldsymbol{\pi}$ that ends with τ_{end} decomposes uniquely into $\boldsymbol{\pi}' \circ \boldsymbol{\pi}'' \circ \tau_{\text{end}}$, where $\boldsymbol{\pi}'$ and $\boldsymbol{\pi}''$ are push computations of type $[i, p, Y, k, r]$ and $[k, r, Z, j-|a|, s]$, respectively, for some $k \in [i+1{:}j-|a|-1]$ and $r \in Q$. We call (k, r) the **split point** of $\boldsymbol{\pi}$.

The algorithm sums over all split points (k, r). Let Π be the set of all push computations of type $[i, p, X, j, q]$ ending in τ_{end} with split point (k, r), and let Π' and Π'' be the sets of all push computations of type $[i, p, Y, k, r]$ and $[k, r, Z, j-|a|, s]$, respectively. Every $\boldsymbol{\pi} \in \Pi$ must be of the form $\boldsymbol{\pi}' \circ \boldsymbol{\pi}'' \circ \tau_{\text{end}}$, where $\boldsymbol{\pi}' \in \Pi'$ and $\boldsymbol{\pi}'' \in \Pi''$, and conversely, for every $\boldsymbol{\pi}' \in \Pi'$ and $\boldsymbol{\pi}'' \in \Pi''$, $\boldsymbol{\pi}' \circ \boldsymbol{\pi}'' \circ \tau_{\text{end}} \in \Pi$. Because $i < k$, we must have $j - |a| - k \leq j - k < j - i$, and because $k < j - |a|$, we must have $k - i < j - |a| - i \leq j - i$. By the induction hypothesis, the total weight of Π' and Π'' were fully computed in a previous iteration. As in the previous case, by distributivity we have

$$\begin{aligned}\bigoplus_{\boldsymbol{\pi}\in\Pi} \bigotimes_{\tau\in\boldsymbol{\pi}} \delta(\tau) &= \mathbf{w}[i, p, Y, k, r] \\ &\otimes \mathbf{w}[k, r, Z, j-|a|, s] \otimes \delta(\tau_{\text{end}}) .\end{aligned}$$ □

4.3 Stack Automaton

The distribution over possible configurations that $\mathcal{P}$ can be in after reading $\boldsymbol{s}(0{:}m]$ can be generated by a weighted finite-state automaton M. The states of M are of the form (i, q), with start state $(0, s)$ and accept states (m, q) for all $q \in Q$. There is a transition $(i, q) \xrightarrow{X/w} (j, r)$ for every item $[i, q, X, j, r]$ with weight w. Then if an accepting run of M scans $\boldsymbol{\gamma}$ and ends in state (m, q) with weight w, then $\mathcal{P}$ can be in configuration $(q, \boldsymbol{\gamma})$ with weight w.

Item form

$$[i, p, XY, j, q] \qquad 0 \leq i < j \leq n,\ p, q \in Q; X, Y \in \Gamma$$

Inference rules

$$\frac{}{[0, \iota, \$\$, 0, \iota]}$$

$$\frac{[i, p, XY, j-|a|, r]}{[i, p, XY, j, q]} \quad r, \varepsilon \xrightarrow{a/w} q, \varepsilon,\ \boldsymbol{s}(j-|a|{:}j] = a$$

$$\frac{[k, r, ZX, i, p]}{[i, p, XY, j, q]} \quad p, \varepsilon \xrightarrow{a/w} q, Y,\ \boldsymbol{s}(i{:}j] = a$$

$$\frac{[i, p, XY, k, r] \quad [k, r, YZ, j-|a|, s]}{[i, p, XY, j, q]} \quad s, Z \xrightarrow{a/w} q, \varepsilon,\ \boldsymbol{s}(j-|a|{:}j] = a$$

$$\frac{[i, p, XZ, j-|a|, r]}{[i, p, XY, j, q]} \quad r, Z \xrightarrow{a/w} q, Y,\ \boldsymbol{s}(j-|a|{:}j] = a$$

Goal

$$[0, \iota, \$\$, n, f]$$

Figure 4: Deductive system for Lang's algorithm.

4.4 Complexity Analysis and Speedup

For comparison with our algorithm, we show the original algorithm of Lang (1974) in Fig. 4. It has items of the form $[i, q, XY, j, r]$, which stands for push computations that start with X as the top stack symbol and push a Y on top of it.

Our algorithm stores a weight for each item $[i, p, X, j, q]$, giving a space complexity of $O\left(n^2|Q|^2|\Gamma|.\right)$ This is more efficient than Lang's algorithm, which requires $O\left(n^2|Q|^2|\Gamma|^2\right)$ space.

Computing the weight of each new item requires, in the worst case (the inference rule for 2-pop transitions), iterating over stack symbols $Y, Z \in \Gamma$, indices $j \in [0{:}n]$ and states $q, r \in Q$, resulting in a runtime of $O(n|Q|^2|\Gamma|^2)$ per item. So the algorithm has a runtime of $O(n^3|Q|^4|\Gamma|^3)$, the same as Lang's algorithm.

This runtime can be improved by splitting the

inference rule for 2-pop transitions into two rules:[3]

$$\frac{[k, r, Z, j-|a|, s]}{\langle k, r, Y\backslash X, j, q\rangle} \quad s, YZ \xrightarrow{a/w} q, X \quad \boldsymbol{s}(j-|a|{:}j] = a$$

$$\frac{[i, p, Y, k, r] \quad \langle k, r, Y\backslash X, j, q\rangle}{[i, p, X, j, q]}$$

The first rule has $O(n^2|Q|^3|\Gamma|^3)$ instantiations and the second rule has $O(n^3|Q|^3|\Gamma|^2)$. So, although we have lost the space-efficiency gain, the total time complexity is now in $O((n^3|\Gamma|^2 + n^2|\Gamma|^3)|Q|^3)$, a speedup of a factor of more than $|Q|$. We show in App. B an alternative deductive system that achieves a similar speedup.

Furthermore, Lang's algorithm only works on simple PDAs. To make the algorithms directly comparable, we can assume in the 2-pop, 1-push case that $X = Y$. This reduces the space complexity by a factor of $|\Gamma|$ again. Moreover, it reduces the number of instantiations of the inference rules above to $O(n^2|Q|^3|\Gamma|^2)$ and $O(n^3|Q|^3|\Gamma|)$, respectively. So the total time complexity is in $O(n^3|Q|^3|\Gamma|^2)$, which is a speedup over Lang's algorithm by a factor of $|Q| \cdot |\Gamma|$.

5 Stringsums in Top-down WPDAs

Stringsums of weighted top-down WPDAs can be computed by the left/right mirror image of our bottom-up algorithm. Instead of finding push computations, this algorithm finds **pop computations**, which decrease (rather than increase) the stack size by exactly one.

Definition 18 (Pop computation)**.** *Let $\mathcal{P}$ be a top-down WPDA and $\boldsymbol{s} \in \Sigma^*$ an input string. A **pop computation** of type $[i, p, X, j, q]$, where $0 \le i \le j \le |\boldsymbol{s}|$, $p, q \in Q$, and $X \in \Gamma$, is a run $\boldsymbol{\pi} = (q_0, \boldsymbol{\gamma}_0), \ldots, (q_m, \boldsymbol{\gamma}_m)$ that scans $\boldsymbol{s}(i{:}j]$, where $q_0 = p$, $q_m = q$, $\boldsymbol{\gamma}_0 = \boldsymbol{\gamma}_m X$, and for all $l < m$, $|\boldsymbol{\gamma}_l| \ge |\boldsymbol{\gamma}_0|$.*

Fig. 5 shows the inference rules used by the dynamic program, which are the mirror image of the rules in Fig. 3. Each item $[i, p, X, j, q]$, which stands for a set of pop computations, is derived using a transition and items corresponding to pop computations that happen *later* in the run.

[3]The \ operator, which is just a punctuation mark and does not require any particular interpretation, was chosen to evoke the \ in categorial grammar (using Lambek's "result on top" convention): $Y\backslash X$ is an X missing a Y on the left.

Item form

$$[i, p, X, j, q] \qquad 0 \le i < j \le n,\ p, q \in Q; X \in \Gamma$$

Inference rules

$$\frac{[i{+}|a|, r, Y, k, s] \quad [k, s, Z, j, q]}{[i, p, X, j, q]} \quad p, X \xrightarrow{a/w} r, YZ \quad \boldsymbol{s}(i{:}i{+}|a|] = a$$

$$\frac{[i{+}1, r, Y, j, q]}{[i, p, X, j, q]} \quad p, X \xrightarrow{\boldsymbol{s}_{i+1}/w} r, Y$$

$$\frac{}{[i, p, X, j, q]} \quad p, X \xrightarrow{\boldsymbol{s}_{i+1}/w} q, \varepsilon \quad i = j - 1$$

Goal

$$[0, \iota, S, n, f]$$

Figure 5: Deductive system for stringsums of top-down WPDAs in normal form.

5.1 Comparison with Lang's algorithm

Since top-down PDAs are more standard, and the only direct PDA stringsum algorithm in the literature is Lang's algorithm, it might have seemed natural to extend Lang's algorithm to top-down PDAs, as is done by DuSell and Chiang (2020). Like Lang's algorithm, their algorithm has items of the form $[i, q, XY, j, r]$, but unlike Lang's algorithm, it requires the X in order to support 1-pop, 2-push transitions. As a result, their algorithm has space complexity $O(n^2|Q|^2|\Gamma|^2)$ and time complexity $O(n^3|Q|^4|\Gamma|^3)$, like Lang's algorithm. But if they had used our algorithm for top-down WPDAs, using pop computations, they would have saved a factor of $|\Gamma|$ space, and because their 1-pop, 2-push transitions never change the popped symbol, they would have also saved a factor of $|Q| \cdot |\Gamma|$ time.

5.2 Experiment

To give a concrete example, we consider the renormalizing nondeterministic stack RNN (RNS-RNN) of DuSell and Chiang (2022), which uses Lang's algorithm (Fig. 4) on a top-down PDA. Since the RNN must process the string from left to right, we cannot use the bottom-up stringsum algorithm, but we can still apply the speedup of §4.4, splitting the 1-pop, 0-push rule of Fig. 4 into two rules. Again, this decreases the time complexity from $O(n^3|Q|^4|\Gamma|^3)$ to $O((n^3|\Gamma|^2 + n^2|\Gamma|^3)|Q|^3)$. When we compare the two implementations on a corpus of strings whose lengths were drawn from $[40, 80]$ on a NVIDIA GeForce RTX 2080 Ti GPU, when $|Q| = 5$ and $|\Gamma| = 3$, the new version is about 10 times as fast (Figure 6).

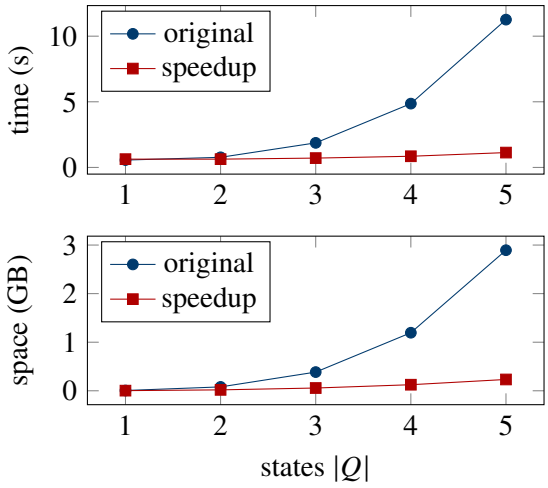

Figure 6: Applying our speedup to the RNS-RNN, which uses Lang's algorithm adapted to top-down PDAs, yields dramatic time and space savings.

5.3 Comparison with CFG/CKY

We also compare our stringsum algorithm with converting a top-down PDA to a CFG and computing stringsums using CKY. The usual conversion from top-down normal form PDAs to CFGs (Hopcroft et al., 2006) creates a CFG with $O(|Q|^2|\Gamma|)$ nonterminal symbols, so CKY would take $O(n^3|Q|^6|\Gamma|^3)$ time. Our algorithm thus represents a speedup of more than $|Q|^3$. Of course, various optimizations could be made to improve this time, and in particular there is an optimization (Eisner and Blatz, 2007) analogous to the speedup in §4.4.

6 Allsums in Bottom-up WPDAs

We can use a notion of push computation similar to Def. 14 to derive a space-efficient algorithm for computing allsums in bottom-up WPDAs. The item $[p, X, q]$ stands for runs from state p to state q that push the symbol X on top of the stack while leaving the symbols underneath unchanged.

Definition 19 (Push computation)**.** *Let $\mathcal{P}$ be a bottom-up WPDA. A **push computation** of type $[p, X, q]$, where $p, q \in Q$, and $X \in \Gamma$, is a run $\boldsymbol{\pi} = (q_0, \boldsymbol{\gamma}_0), \ldots, (q_n, \boldsymbol{\gamma}_n)$, where $q_0 = p$, $q_n = q$, $\boldsymbol{\gamma}_n = \boldsymbol{\gamma}_0 X$, and for all $i > 0$, $|\boldsymbol{\gamma}_i| \geq |\boldsymbol{\gamma}_n|$.*

These items closely resemble those used for computing stringsums, but discard the two variables i, j that we used for indexing substrings of the input, as we are interested in computing the total weight of runs that scan *any* string.

Definition 20. *Let $\Pi(p, X, q)$ be the set containing all push computations from state p to state q that push X. The allsum $\mathbf{w}[p, X, q]$ is defined as*

$$\mathbf{w}[p, X, q] = \bigoplus_{\boldsymbol{\pi} \in \Pi(p,X,q)} \mathbf{w}(\boldsymbol{\pi}).$$

The allsum of a set of push computations can be expressed in terms of other allsums:

$$\mathbf{w}[p, X, q] = \bigoplus_{a \in \Sigma \cup \{\varepsilon\}} \delta(p, \varepsilon \xrightarrow{a} q, X)$$
$$\oplus \bigoplus_{\substack{Y \in \Gamma \\ r \in Q \\ a \in \Sigma \cup \{\varepsilon\}}} \mathbf{w}[p, Y, r] \otimes \delta(r, Y \xrightarrow{a} q, X)$$
$$\oplus \bigoplus_{\substack{Y, Z \in \Gamma \\ r, s \in Q \\ a \in \Sigma \cup \{\varepsilon\}}} \mathbf{w}[p, Y, r] \otimes \mathbf{w}[r, Z, s] \otimes \delta(s, YZ \xrightarrow{a} q, X)$$

In general, allsums cannot be computed recursively, as $\mathbf{w}[p, X, q]$ may rely on allsums that are yet to be computed. Instead, we assume that $\mathcal{W}$ is continuous and solve the system of nonlinear equations using fixed-point iteration or the semiring generalization of Newton's method (Esparza et al., 2007).

This algorithm computes $O(|Q|^2|\Gamma|)$ items. An allsum algorithm based on Lang's algorithm would have computed $O(|Q|^2|\Gamma|^2)$ items; thus we have reduced the space complexity by a factor of $|\Gamma|$.

7 Conclusion

Our study has contributed several results and algorithms whose weighted CFG analogues have long been known, but have previously been missing for weighted PDAs—a normal form analogous to Chomsky normal form and a stringsum algorithm analogous to weighted CKY. But it has also revealed some important differences, confirming that the study of weighted PDAs is of interest in its own right. Most notably, we identified two different normal forms and two corresponding stringsum algorithms (and two allsum algorithms). Since the only existing PDA stringsum algorithm we are aware of, Lang's algorithm, is better suited to bottom-up PDAs, whereas the more standard definition of PDAs is of top-down PDAs, our algorithm for top-down WPDAs fills a significant gap.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CCF-2019291. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Limitations

Removal of nullary transitions, while similar to removal of nullary rules from a WCFG, is conceptually more complicated, and while it has the same asymptotic complexity, it's currently unknown how the two would compare in practice. Additionally, our nullary removal construction requires a commutative semiring, while removal of nullary productions from a WCFG does not.

Our algorithm for top-down WPDAs processes a string from right to left, which may be undesirable in some NLP applications and in models of human sentence processing.

Ethics Statement

The authors foresee no ethical concerns with the research presented in this paper.

References

Alfred V. Aho and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling*, volume 1. Prentice-Hall.

Cyril Allauzen, Bill Byrne, Adrià de Gispert, Gonzalo Iglesias, and Michael Riley. 2014. Pushdown automata in statistical machine translation. *Computational Linguistics*, 40(3):687–723.

Daniel Andor, Chris Alberti, David Weiss, Aliaksei Severyn, Alessandro Presta, Kuzman Ganchev, Slav Petrov, and Michael Collins. 2016. Globally normalized transition-based neural networks. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2442–2452.

Jean-Michel Autebert, Jean Berstel, and Luc Boasson. 1997. Context-free languages and pushdown automata. In *Handbook of Formal Languages*, volume 1, pages 111–174.

Y. Bar-Hillel, M. Perles, and E. Shamir. 1961. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172.

Danqi Chen and Christopher Manning. 2014. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 740–750.

Noam Chomsky. 1963. Formal properties of grammars. In *Handbook of Mathematical Psychology*, volume 2, pages 323–418. John Wiley & Sons.

Manfred Droste and Werner Kuich. 2009. Semirings and formal power series. In *Handbook of Weighted Automata*, pages 3–28.

Brian DuSell and David Chiang. 2020. Learning context-free languages with nondeterministic stack RNNs. In *Proceedings of the 24th Conference on Computational Natural Language Learning*, pages 507–519.

Brian DuSell and David Chiang. 2022. Learning hierarchical structures with differentiable nondeterministic stacks. In *International Conference on Learning Representations*.

Chris Dyer, Miguel Ballesteros, Wang Ling, Austin Matthews, and Noah A. Smith. 2015. Transition-based dependency parsing with stack long short-term memory. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 334–343.

Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A. Smith. 2016. Recurrent neural network grammars. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 199–209.

Jay Earley. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

Jason Eisner and John Blatz. 2007. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proceedings of the 11th Conference on Formal Grammar*, pages 45–85.

Javier Esparza, Stefan Kiefer, and Michael Luttenberger. 2007. On fixed point equations over commutative semirings. In *24th Annual Symposium on Theoretical Aspects of Computer Science*, pages 296–307.

R. James Evey. 1963. Application of pushdown-store machines. In *AFIPS '63: Proceedings of the November 12–14, 1963, Fall Joint Computer Conference*, pages 215—227.

Daniel Fernández-González and Carlos Gómez-Rodríguez. 2019. Left-to-right dependency parsing with pointer networks. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 710–716.

Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics*, 25(4):573–606.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation*, 3rd edition. Addison-Wesley Longman Publishing Co.

Liang Huang, Wenbin Jiang, and Qun Liu. 2009. Bilingually-constrained (monolingual) shift-reduce parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, pages 1222–1231.

Donald E. Knuth. 1965. On the translation of languages from left to right. *Information and Control*, 8(6):607–639.

Bernard Lang. 1974. Deterministic techniques for efficient non-deterministic parsers. In *ICALP 1974: Automata, Languages and Programming*, pages 255–269.

Daniel J. Lehmann. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76.

Harry R. Lewis and Christos H. Papadimitriou. 1997. *Elements of the Theory of Computation*, 2nd edition. Prentice-Hall.

Xuezhe Ma, Zecong Hu, Jingzhou Liu, Nanyun Peng, Graham Neubig, and Eduard Hovy. 2018. Stack-pointer networks for dependency parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1403–1414.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the Eighth International Conference on Parsing Technologies*, pages 149–160.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57.

Philip Resnik. 1992. Left-corner parsing and psychological plausibility. In *COLING 1992 Volume 1: The 14th International Conference on Computational Linguistics*, pages 191–197.

Brian Roark. 2001. Probabilistic top-down parsing and language modeling. *Computational Linguistics*, 27(2):249–276.

Marcel-Paul Schützenberger. 1963. On context-free languages and push-down automata. *Information and Control*, 6(3):246–264.

Tianze Shi, Liang Huang, and Lillian Lee. 2017. Fast(er) exact decoding and global training for transition-based dependency parsing via a minimal feature set. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 12–23.

Stuart M. Shieber, Yves Schabes, and Fernando C.N. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24:3–36.

Michael Sipser. 2012. *Introduction to the Theory of Computation*, 3rd edition. Cengage Learning.

Andreas Stolcke. 1995. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201.

David Weiss, Chris Alberti, Michael Collins, and Slav Petrov. 2015. Structured training for neural network transition-based parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 323–333.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 562–571.

A Details of Unary Removal

Since U is a $|Q| \times |\Gamma|$ matrix, computing its transitive closure takes $O((|Q||\Gamma|)^3)$ time. However, if we perform nullary removal first, the stack alphabet could grow by a factor of $|Q|^2$ because of the stack symbols ${}_{rs}Z$, which would seem to make the transitive closure take $O((|Q|^3|\Gamma|)^3)$ time.

For comparison, if we converted the PDA to a CFG, it would have $O(|Q|^2|\Gamma|)$ nonterminals, so computing the transitive closure of the unary rules would take $O((|Q|^2|\Gamma|)^3)$ time.

But the graph formed by the unary transitions can be decomposed into several strongly connected components (SCCs), many of which are identical, so the transitive closure can be sped up considerably. Define three matrices for three different forms of unary transitions:

$$U^1_{ptZ,qsX} = \delta(p, {}_{rt}Z \xrightarrow{\varepsilon} q, {}_{rs}X)$$
$$U^2_{qpX,qX} = \delta(q, {}_{pp}X \xrightarrow{\varepsilon} q, X^{\not\in})$$
$$U^3_{pY,qX} = \delta(p, Y^{\not\in} \xrightarrow{\varepsilon} q, X^{\not\in}).$$

There are no transitions of the form $p, Y^{\not\in} \xrightarrow{\varepsilon/w} q, {}_{rs}X$. Note that in the first equation, the transition weight does not depend on r, so r does not occur on the left-hand side. Then let

$$V = U^{1^*} U^2 U^{3^*}$$

so that $V_{psY,qX}$ is the total weight of runs from configurations of the form $(p, {}_{rs}Y)$ to configurations of the form $(q, X^{\not\in})$.

Finally, we remove the unary transitions and modify the non-unary transitions as follows:

For each non-unary:	Replace with:
$p, \boldsymbol{\gamma} \xrightarrow{a/w} q, {}_{rs}X$	$p, \boldsymbol{\gamma} \xrightarrow{a/w \otimes U^{1*}_{qsX,tuY}} t, {}_{ru}Y$
	$p, \boldsymbol{\gamma} \xrightarrow{a/w \otimes V_{qsX,tY}} t, Y^{\not\in}$
$p, \boldsymbol{\gamma} \xrightarrow{a/w} q, X^{\not\in}$	$p, \boldsymbol{\gamma} \xrightarrow{a/w \otimes U^{3*}_{qX,rY}} r, Y^{\not\in}$

Since V can be computed in $O((|Q|^2|\Gamma|)^3)$ time, the asymptotic time complexity of removing nullary and unary transitions is the same when performed directly on the WPDA as compared with converting to a WCFG and removing nullary and unary rules.

Item form

$$[i, p, \boldsymbol{\gamma}, j, q] \qquad 0 \le i < j \le n; p, q \in Q;\ \boldsymbol{\gamma} \in \Gamma^*; |\boldsymbol{\gamma}| \in [1{:}2]$$

Inference rules

$$\frac{[i, p, Y, k, r] \quad [k, r, Z, j', s]}{[i, p, YZ, j', s]}$$

$$\frac{[i, p, YZ, j{-}|a|, s]}{[i, p, X, j, q]} \qquad s, YZ \xrightarrow{a/w} q, X;\ \boldsymbol{s}(j{-}|a|{:}j] = a$$

$$\frac{[i, p, Y, j{-}1, r]}{[i, p, X, j, q]} \qquad r, Y \xrightarrow{\boldsymbol{s}_j/w} q, X$$

$$\frac{}{[i, p, X, j, q]} \qquad p, \varepsilon \xrightarrow{\boldsymbol{s}_j/w} q, X;\ j = i + 1$$

Goal

$$[0, \iota, S, n, f]$$

Figure 7: Deductive system corresponding to the alternative sped-up algorithm for stringsums in bottom-up WPDAs in normal form.

B Fast Bottom-up Stringsum Algorithm

Fig. 7 shows an alternative deductive system for parsing in bottom-up WPDAs. The algorithm that can be derived from this deductive system achieves a runtime improvement by a factor of $|Q|$ and has the same space complexity as Lang's algorithm. This algorithm, however, does not achieve further time and space complexity improvements on the special type of automaton used by Lang.