Cache Transition Systems for Graph Parsing

Daniel Gildea*
University of Rochester

Giorgio Satta**
Università di Padova

Xiaochang Peng†
University of Rochester

Motivated by the task of semantic parsing, we describe a transition system that generalizes standard transition-based dependency parsing techniques to generate a graph rather than a tree. Our system includes a cache with fixed size m*, and we characterize the relationship between the parameter* m *and the class of graphs that can be produced through the graph-theoretic concept of tree decomposition. We find empirically that small cache sizes cover a high percentage of sentences in existing semantic corpora.*

1. Introduction

As statistical natural language processing systems have progressed to provide deeper representations, there has been renewed interest in graph-based representations of semantic structures and in algorithms to produce them. Typically, these algorithms behave similarly to standard parsing algorithms for retrieving syntactic representations: They take as input a sentence and produce as output a graph representation of the semantics of the sentence itself.

At the same time, recent years have seen a general trend from chart-based syntactic parsers toward stack-based transition systems, as the accuracy of transition systems has increased, and as speed has become increasingly important for real-world applications. On the syntactic side, stack-based transition systems for projective dependency parsing run in time $\mathcal{O}(n)$, where n is the sentence length; for a general overview of these systems, see, for instance, the presentation of Nivre (2008). There have also been

* Computer Science Department, University of Rochester, Rochester NY 14627.
E-mail: gildea@cs.rochester.edu.
** Dipartimento di Ingegneria dell'Informazione, Università di Padova, Via Gradenigo 6/A, 35131 Padova, Italy. E-mail: satta@dei.unipd.it.
† Computer Science Department, University of Rochester, Rochester NY 14627.
E-mail: xpeng@cs.rochester.edu.s

Submission received: 19 December 2016; revised version received: 26 June 2017; accepted for publication: 7 July 2017.

doi:10.1162/COLI_a_00308

a number of extensions of stack-based transition systems to handle non-projective trees (e.g., Attardi 2006; Nivre 2009; Choi and McCallum 2013; Gómez-Rodríguez and Nivre 2013; Pitler and McDonald 2015).

Stack-based transition systems can produce general graphs rather than trees. Perhaps the simplest way to generate graphs is to shift one word at a time onto the stack, and then consider building all possible arcs between each word on the stack and the next word in the buffer. This is essentially the algorithm of Covington (2001), generalized to produce graphs rather than non-projective trees. This algorithm was also cast as a stack-based transition system by Nivre (2008). The algorithm runs in time $\mathcal{O}(n^2)$, and requires the system to discriminate the arcs to be built from a large set of possibilities, potentially leading to errors.

Traditional stack-based parsing, which is restricted to trees, and the Covington algorithm as generalized to graph parsing can be thought of as two extremes, with a wide set of possible intermediate approaches staking out different trade-offs between expressiveness, on the one hand, and time and the discrimination required of machine learning components on the other. In this article, we mathematically explore this trade-off and precisely characterize the relationship between parsing systems and the set of graphs they can build. We describe a parsing system based on adding a working set, which we refer to as a **cache**, to the traditional stack and buffer. With cache size 2, our algorithm can only build trees, while with unbounded cache, our algorithm can build any graph, because it is then equivalent to the Covington algorithm generalized to graphs. We speculate that small, fixed cache sizes provide a good trade-off for fast and accurate string-to-graph parsing.

We analyze the class of graphs that can be successfully constructed by our parsing system, making use of the graph-theoretic notion of treewidth. The treewidth of a graph gives a measure of how tightly interconnected it is: Trees have treewidth 1, and fully connected graphs on n vertices have treewidth $n - 1$. We show that the class of graphs constructed by our parser is precisely characterized by treewidth: A transition system of cache size m can produce graphs of treewidth $m - 1$. Our framework assumes an input order of vertices, corresponding to the word order of the string, and we define a concept of relative treewidth to characterize the set of graphs that the parser can produce given a fixed input order of vertices. Finally, we develop an oracle algorithm for our parsing system, and prove its correctness. We also provide an algorithm for computing the minimal cache size needed to parse a given data set.

In general, a graph's relative treewidth with respect to an input order may be much higher than its absolute treewidth. However, if relative treewidth with respect to the real English word order is low, and not significantly higher than the absolute treewidth, this indicates that the word order provides valuable information about the graph structure to be predicted, and that efficient parsing is possible by making use of this information. We test this hypothesis with experiments on Abstract Meaning Representation (Banarescu et al. 2013), a semantic formalism where the meaning of a sentence is encoded as a directed graph. We find that, for English sentences, these structures have low relative treewidth with respect to the English word order, and can thus be parsed efficiently using a transition-based parser with small cache size. In order to compare across a wider variety of the semantic representations that have been

proposed (Kuhlmann and Oepen 2016), we also experiment with three sets of semantic dependencies from the Semeval 2015 semantic dependency parsing task (Oepen et al. 2015). With these data sets, which are generally closer to the surface string structure than Abstract Meaning Representation, we find somewhat higher relative treewidth. In every data set that we analyzed, over 99% of sentences can be covered with a cache size of eight.

2. Tree Decomposition and Treewidth

In this section we introduce and define the notions of tree decomposition and treewidth, as well as a few related concepts that we will use throughout this article. As usual, we denote an undirected graph as $G = (V, E)$, where V is the set of vertices and E is the set of edges. Each edge is represented as an unordered pair (u, v) with $u, v \in V$.

The theory developed in this article is based on the graph theoretical notion of tree decomposition, which has been independently developed in several areas of computer science and discrete mathematics. From an application-oriented perspective, tree decomposition has proven very useful in discrete optimization and in the design of polynomial time algorithms using dynamic programming techniques.

The intuitive idea behind the notion of tree decomposition can be explained as follows. At this point, we use the term "interconnection" in a rather informal way; the precise meaning of this notion will be mathematically defined later. A tree is a special kind of graph where the vertices are arranged in a hierarchical way, with the property that the set of vertices in any subtree have only one interconnection with the set of the remaining vertices. In contrast, for a general graph this is not possible, meaning that we cannot group vertices in a hierarchical structure and pretend that there are a small number of interconnections between vertices in any subtree and the remaining vertices. This is apparent for a complete graph, that is, a graph where each vertex is connected with every other vertex. More interestingly, this is also true for a grid-like graph, where any hierarchical decomposition of the set of vertices will always lead to some subtree with a number of interconnections with the remaining structure that is not bounded by a constant. Note that, in contrast with a complete graph, where each vertex has a number of neighbors that is not bounded by a constant, in a grid each vertex has at most four neighbors. Still, the internal structure of a grid is unfavorable for this type of hierarchical arrangement. The notion of tree decomposition of a graph, and the related notion of treewidth, provide us precisely with the information we need: To what degree is it possible to arrange the vertices of a graph into some hierarchical structure, with the property that interconnections between vertices in any subtree and the remaining vertices are kept to a minimum?

More precisely, a tree decomposition of a graph G is a type of tree having a subset of G's vertices at each node. To avoid confusion, when describing tree decompositions we use the terms **node** and **arc**, and when describing graphs we use the terms **vertex** and **edge**. In a tree decomposition T, the set of nodes is denoted I and the set of arcs is denoted F. The subset of V associated with node $i \in I$ is referred to as a **bag**, and is denoted by X_i. Formally, a **tree decomposition** of a graph $G = (V, E)$

is defined as a pair $(\{X_i \mid i \in I\}, T = (I, F))$ where tree T satisfies all of the following properties.

- *Vertex cover:* The nodes of the tree T cover all the vertices of G: $\bigcup_{i \in I} X_i = V$.
- *Edge cover:* Each edge in G is included in some node of T. That is, for all edges $(u, v) \in E$, there exists an $i \in I$ with $u, v \in X_i$.
- *Running intersection:* The nodes of T containing a given vertex of G form a connected subtree. Mathematically, for all $i, j, k \in I$, if j is on the (unique) path from i to k in T, then $X_i \bigcap X_k \subseteq X_j$.

The **width** of a tree decomposition $(\{X_i\}, T)$ is $\max_i |X_i| - 1$. The **treewidth** of a graph is the minimum width over all tree decompositions

$$\mathbf{tw}(G) = \min_{(\{X_i\}, T) \in TD(G)} \max_i |X_i| - 1$$

where $TD(G)$ is the set of valid tree decompositions of G. We refer to a tree decomposition achieving the minimum possible width as being **optimal**.

In general, more densely interconnected graphs have higher treewidth. For instance, any tree has treewidth 1, a graph consisting of a single cycle with three or more vertices has treewidth 2, and a fully connected graph of n vertices has treewidth $n - 1$. Low treewidth thus indicates some treelike structure underlying the graph. When certain properties of the graph must be checked, the tree decomposition is often helpful in organizing computation (see, e.g., results of Courcelle [1990] and Arnborg, Lagergren, and Seese [1991]). Finding the treewidth of a graph is an NP-complete problem (Arnborg, Corneil, and Proskurowski 1987).

Example 1
Consider the graph G in Figure 1 with vertex set $V = \{A, B, C, \ldots, Q, R, S\}$. At first sight, G's structure seems rather intricate, with edges scattered all over the picture. However, an optimal tree decomposition of G reveals that there is a tree-like structure underlying G. An optimal tree decomposition T of G is displayed in Figure 2(a), where we use gray

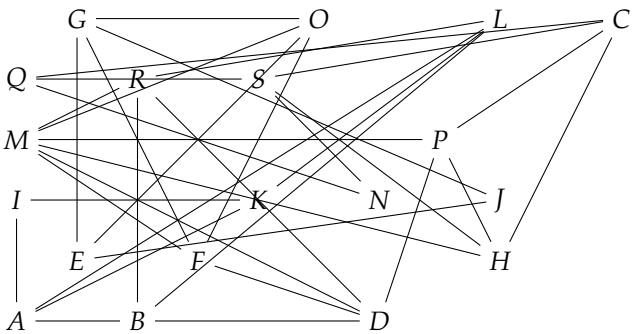
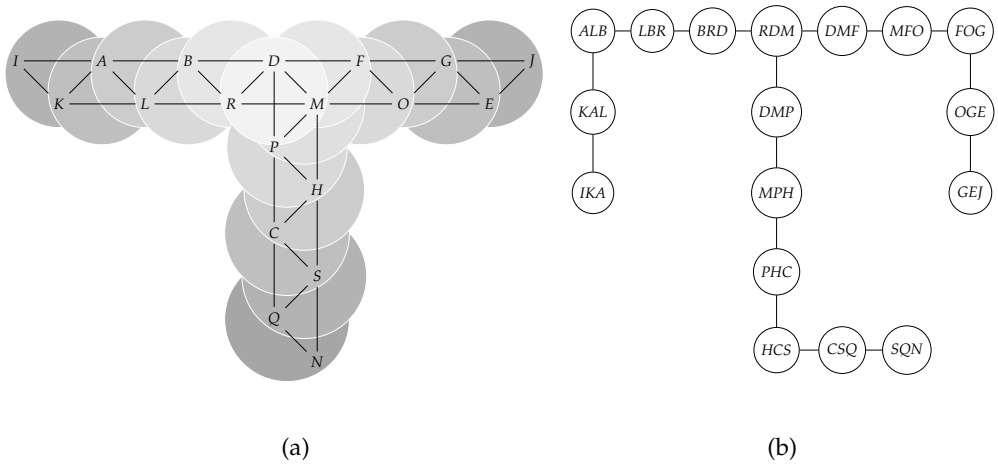


Figure 1
Graph G with vertex set $V = \{A, B, C, \ldots, Q, R, S\}$.



(a) (b)

Figure 2
(a) An optimal tree decomposition of graph G in Figure 1; this is a set of overlapping clusters of G's vertices, arranged in a tree. (b) The high-level treelike structure of G becomes apparent when it is drawn ignoring G's edges.

circles to indicate the sets of vertices of G that represent the bags of T. Because adjacent bags of T share some of their vertices, our gray circles partially overlap. The tree-like structure underlying G is apparent from this overlapping. This tree decomposition has bags of three vertices each, and thus the graph's treewidth is 2. In this representation, it is also apparent that there is a low number of interconnections among vertices in a subtree of T and the remaining vertices of G.

An alternative representation of the same tree decomposition is shown in Figure 2(b), where we focus on the vertices and ignore the edges of the graph. It is easy to see that the vertex cover and the edge cover conditions in the definition of tree decomposition are both satisfied by T. As an example of the running intersection property, note that the vertex S appears in three adjacent nodes of the tree decomposition.

Although general tree decompositions are undirected trees, in this article we will work with rooted, directed tree decompositions, in which one node is designated as the root, and the children of each node are ordered. We say that a rooted, ordered tree decomposition of graph G having width k is **smooth** if each bag contains exactly $k + 1$ vertices, and each bag contains the same vertices as its parent bag, with exactly one vertex removed and one vertex added. The tree decomposition in Figure 2(b) is smooth.

The concept of smooth tree decompositions, for standard unrooted tree decompositions, was introduced by Bodlaender (1996). Throughout this article, we also require that the root of a smooth tree decomposition contains $k + 1$ copies of the special symbol \$, with vertices of G being added one at a time in the bags below the root. It is easy to see that the size of a smooth tree decomposition (i.e., the number of nodes of the tree) is the number of vertices in the graph plus one.

Lemma 1
Any tree decomposition T of graph G can be transformed into a smooth tree decomposition T' of G of equal width.

Proof. Let k be the width of T. At each bag having fewer than $k+1$ vertices, continue adding vertices from adjacent bags until all bags have the same size. If two adjacent bags B_1 and B_2 end up having the same vertices, collapse B_1 and B_2 into a single bag, and merge the children of the two bags in a way that preserves their order. If two adjacent bags B_1 and B_2 differ by more than one vertex in their contents, add intermediate bags by adding vertices from B_2 and removing vertices from B_1 one at a time. Finally, choose a bag B as the root of the tree constructed so far. Add a new root containing $k+1$ instances of the special symbol \$, and intermediate bags connecting the root to B adding one vertex of B at a time, and removing instances of \$. □

As already discussed in the Introduction, in natural language processing applications, we are not provided with a graph structure as input, and we are not asked to recognize whether that graph belongs to some formal language. We are instead given as input an ordered sequence of vertices of some graph, or a superset thereof, and we are asked to retrieve the graph itself. Although this latter problem is apparently more difficult than the former, since the edges of the graph must be decoded, the input ordering of the vertices plays an important role and can be used to restrict the search space, ultimately ending up with a more efficient computation. This idea is at the basis of the algorithms for graph parsing developed in this article. We therefore now introduce the notion of relative treewidth with respect to a given order of the vertices of a graph, which is original to this article.

Let $G = (V, E)$ be some graph and let T be a smooth tree decomposition of G. We define the **vertex order** $\pi(T)$ of T to be the sequence of vertices produced by visiting T in a preorder, left to right traversal and by listing the vertices newly introduced at the visited bags. Each vertex of V will appear exactly once in $\pi(T)$. We will analyze the behavior of our parser when given a fixed input order over the vertices in terms of a notion of relative treewidth with respect to the input order. We define the **relative treewidth** of G with respect to an order π of G's vertices to be the minimum width of any tree decomposition of G whose vertex order is π. Formally, we write

$$\mathbf{rtw}(G, \pi) = \min_{\substack{(\{X_i\}, T) \in TD(G), \\ \pi(T) = \pi}} \max_i |X_i| - 1.$$

Example 2
Consider the chain-like graph G in Figure 3, with vertex set $\{1, 2, 3, 4\}$. In the top row, G is presented in vertex order $\pi = (1, 2, 3, 4)$, along with a tree decomposition T such that $\pi(T) = \pi$. In the bottom row G is presented in vertex order $\pi' = (1, 2, 4, 3)$, with the tree decomposition T' achieving the minimum width such that $\pi(T') = \pi'$. The order π' increases the relative treewidth: $\mathbf{rtw}(G, \pi) = 1$, while $\mathbf{rtw}(G, \pi') = 2$.

We note that, for any graph G, there exists a vertex order achieving its optimal width

$$\mathbf{tw}(G) = \min_{\pi} \mathbf{rtw}(G, \pi)$$

This is because, by Lemma 1, any optimal tree decomposition of G can always be converted to a smooth tree decomposition of equal width, which provides us with the optimal order.

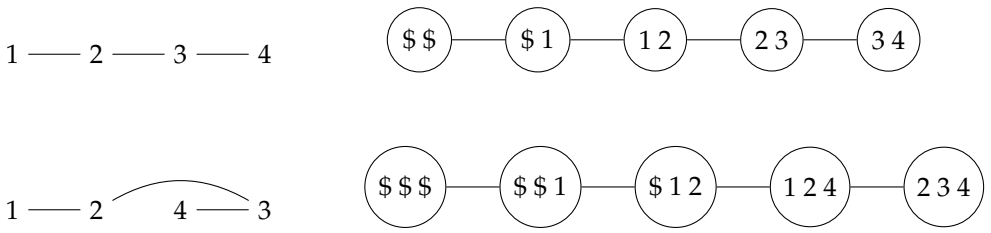


Figure 3
A chain-like graph G, presented at top left in vertex order $\pi = (1, 2, 3, 4)$ and at bottom left in vertex order $\pi' = (1, 2, 4, 3)$. Tree decompositions with minimum relative treewidth with respect to π and π' are displayed at the right. We have $\mathbf{rtw}(G, \pi) = 1$ and $\mathbf{rtw}(G, \pi') = 2$.

Although our notion of relative treewidth with respect to a vertex order is superficially similar to the standard vertex elimination algorithm for finding a tree decomposition (Bodlaender 2006), which also takes as input a vertex order, these orders are in fact distinct. We will not make use of the vertex elimination algorithm in this article, but we describe it briefly here for readers interested in the connection between these two concepts. In the vertex elimination algorithm, vertices are processed in the input order, by adding edges connecting the current vertex's remaining neighbors and then eliminating the current vertex. Each vertex along with its neighbors at the time of its elimination form one bag of the tree decomposition. The order of the vertex elimination algorithm corresponds to the order in which vertices are introduced in some outside–in traversal of the tree decomposition, whereas the order of our concept of relative treewidth corresponds to a pre-order traversal of a smooth tree decomposition.

3. Cache Transition Parser

In this section, we introduce a nondeterministic computational model for graph-based parsing, which we call a **cache transition parser**. The model takes as input an ordered sequence of vertices, reads it strictly from left to right, and incrementally produces a graph as output. Our model is an extension of the transition-based parsing framework described by Nivre (2008) for dependency tree parsing. We assume the reader is familiar with such a framework. We also provide a characterization of our cache transition parsers using the notions of tree decomposition and width that have been introduced in Section 2. Throughout this section, for integer $m \geq 1$ we write $[m]$ to denote the set $\{1, \ldots, m\}$.

Informally, a cache transition parser is a transition-based parser that processes input vertices and produces an output graph. The graph is defined on the input vertices, or on a subset thereof. Besides its stack and buffer, the parser also uses a cache. A cache is a fixed-size array of $m \geq 1$ elements and, along with the stack, represents the storage of the parser. At any time during the computation, a vertex that is in the storage of the parser is either in the cache or else in the stack, but not in both at the same time. The graph vertices in the input buffer are shifted into the cache *before* entering the stack. While in the cache, vertices can be directly accessed and edges between these vertices can be constructed.

Because the cache has fixed size, in order to be able to read a new vertex from the buffer and shift it into the cache, we need to make new room in the cache by moving some other vertex v from the cache into the stack. Once v is in the stack, it is no longer accessible for the operations of edge construction. Typically, the parser moves v out of the cache and into the stack when it predicts that, in the process of edge construction, v does not need to be accessed for a while. For instance, this happens when v's neighbors that still need to be processed are all placed at a far distance in the buffer.

Crucially, the cache is not governed by a first-in first-out policy as in a queue: The vertex v that we move out of the cache might not be the "oldest" vertex that has been introduced in the cache itself. As a consequence, the choice of the vertex that is moved out of the cache and into the stack at each step may considerably alter the original ordering of the vertices in the input.

In addition to this operation, it is also possible to pop some vertex from the stack and put it back into the cache. Again, because the cache has fixed size, in order to be able to do this we need to make new room in the cache. This time this is done by permanently removing some vertex from the cache, meaning that this vertex is dropped out of the parser storage. This happens when the parser decides that all of the edges impinging on a vertex have been processed, and the vertex itself is no longer needed. Going back to our running example about vertex v, when the far distance neighbors of v will reach the foremost position of the buffer and will be shifted into the cache, we can exploit the previous operation, pop v from the stack, and move it back into the cache, where it will be available for the construction of the new edges. Altogether, the combination of these two operations has the effect of repeatedly moving v back and forth between the cache and the stack.

Formally, a **cache transition parser** consists of a stack, a cache, and an input buffer. The stack is a sequence σ of vertices and integers, as explained subsequently, with the topmost element always at the rightmost position. The buffer is a sequence of vertices β containing a suffix of the input, with the first element to be read at the leftmost position. Finally, the cache is a sequence of vertices η. The element at the leftmost position is called the first element of the cache, and the element at the rightmost position is called the last element.

Operationally, the functioning of the parser can be described in terms of configurations and two transitions. Each transition is a binary relation defined on the set of configurations. A **configuration** of our parser has the form:

$$c = (\sigma, \eta, \beta, E)$$

where σ, η, and β are as described earlier, and E is the set of edges being built. The initial configuration of the parser is $([], [\$, \ldots, \$], [v_1, \ldots, v_n], \emptyset)$, meaning that the stack and edge set are initially empty, and the cache is filled with m occurrences of the special symbol \$. The final configuration is $([], [\$, \ldots, \$], [], E_G)$, where the stack and the cache are as in the initial configuration and the buffer is empty. The constructed graph has set of vertices $\{v_1, \ldots, v_n\}$ and set of edges E_G.

The **transitions** of the parser are specified as follows.

- push(i, C) is parameterized by a position in the cache $i \in [m]$ and a set of positions in the cache $C \subseteq [m] \setminus \{i\}$. It takes a configuration:

$$(\sigma, [v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_m], v|\beta, E)$$

and moves to a configuration:

$$(\sigma|i|v_i, [v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_m, v], \beta, E')$$
$$E' = E \cup \{(v_k, v) \mid k \in C\}$$

Here, we have shifted the next vertex v out of the buffer and moved it into the last position of the cache. We have also taken the vertex v_i appearing in position i in the cache and pushed it onto the stack σ, along with the integer i recording the position in the cache from which it came. Finally, we have added some edges to the graph being built, where the new edges connect the shifted vertex v with some subset of the other vertices in the cache. This subset is specified by the parameter C.

- pop takes a configuration:

$$(\sigma|i|v, [v_1, \ldots, v_m], \beta, E)$$

and moves to a configuration:

$$(\sigma, [v_1, \ldots, v_{i-1}, v, v_i, \ldots, v_{m-1}], \beta, E)$$

Here we have popped a vertex v from the stack, along with the integer i recording the position in the cache that it originally came from. We place v in position i in the cache, shifting the remainder of the cache one position to the right, and discarding the last element in the cache.

Example 3
Consider the sentence "John wants Mary to succeed" and the associated semantic representation displayed as a graph in Figure 4. Note that the graph, with vertices in the order of the English sentence, corresponds to the graph at the bottom row of Figure 3. When given the vertex sequence [j, w, m, t, s] as input, our nondeterministic parser will be able to construct the given graph using the run displayed in Figure 5. For instance, when the parser reaches the configuration ([1, $, 1, $, 1, $], [j, w, m], [s], E_1), the transition push$(1, \{1, 2\})$ pushes the vertex j from the cache into the stack, shifts the vertex s from the buffer into the cache, and constructs the two new edges (w, s) and (m, s). The resulting configuration is then ([1, $, 1, $, 1, $, 1, j], [w, m, s], [], $E_1 \cup$ {(w, s), (m, s)}).

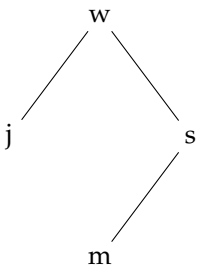


Figure 4
Graph for the semantic representation of the sentence "John wants Mary to succeed." Vertex w represents word token *wants*, vertex j represents *John*, vertex s represents *succeed*, and vertex *m* represents *Mary*.

We have described our transition system as producing undirected graphs with unlabeled edges, but it can be easily extended to produce directed graphs and labeled edges. Directed graphs can be produced by modifying the parameter C of the push transition to be defined as a set of tuples, where each tuple consists of an integer k and a binary variable that specifies whether to produce edge (v, v_k) or edge (v_k, v). Similarly, labeled edges can be produced by adding a value to each tuple specifying the edge's label. Notice that the tuple representation could also be used to allow multiple arcs between the same two nodes, with different directions and labels. These extensions do not fundamentally change the set of graphs that can be produced with a given cache size; a directed (or labeled) graph can be produced if and only if its undirected (or unlabeled) counterpart can be produced. For this reason, we treat our graphs as undirected (and unlabeled) in the remainder of this article.

We now show that, in our parser, each pop transition reverses the effect of some previous push transition, in a sense that will be specified below. For $s \geq 1$, consider a

stack	cache	buffer	edges	resulting from action
[]	[$, $, $]	[j, w, m, t, s]	$\emptyset$	—
[1, $]	[$, $, j]	[w, m, t, s]	$\emptyset$	push$(1, \emptyset)$
[1, $s, 1, $]	[$, j, w]	[m, t, s]	E_1	push$(1, \{2\})$
[1, $, 1, $, 1, $]	[j, w, m]	[t, s]	E_1	push$(1, \emptyset)$
[1, $, 1, $, 1, $, 2, w]	[j, m, t]	[s]	E_1	push$(2, \emptyset)$
[1, $, 1, $, 1, $]	[j, w, m]	[s]	E_1	pop
[1, $, 1, $, 1, $, 1, j]	[w, m, s]	[ ]	E_2	push$(1, \{1, 2\})$
[1, $, 1, $, 1, $]	[j, w, m]	[ ]	E_2	pop
[1, $, 1, $]	[$, j, w]	[ ]	E_2	pop
[1, $,]	[$, $, j]	[ ]	E_2	pop
[ ]	[$, $, $]	[ ]	E_2	pop

Figure 5
Example run of the cache transition system constructing the graph of Figure 4. We have set $E_1 = \{(w, j)\}$ and $E_2 = E_1 \cup \{(w, s), (m, s)\}$.

sequence of $2s$ transitions $\gamma = t_1, \ldots, t_{2s}$. We say that γ is **minimal reversing** if it consists of s push transitions intermixed with s pop transitions, with the property that in any proper prefix of γ of the form $t_1, \ldots, t_k$, $k \in [2s-1]$, the number of push transitions is strictly greater than the number of pop transitions. It is not difficult to see that, if γ is minimal reversing, t_1 must be a push transition and t_{2s} must be a pop transition.

Lemma 2
Let c be a configuration of the parser with stack σ and cache η. Let also γ be a minimal reversing sequence of transitions. If we apply to c the transitions of γ in the given order, we reach a configuration c' with stack $\sigma' = \sigma$ and cache $\eta' = \eta$.

Proof. Let $\gamma = t_1, \ldots, t_{2s}$. We proceed by induction on s. If $s = 1$, γ must be composed by a push followed by a pop. The definition of the pop transition exactly restores the stack and the cache of the configuration c to which the push applied.

If $s > 1$, let $\gamma' = t_2, \ldots, t_{2s-1}$. It is not difficult to see that t_2 must be a push transition and t_{2s-1} must be a pop transition. However, a proper prefix of γ' might now have a number of push transitions that equals the number of pop transitions, making γ' not minimal reversing. If this is the case, we split γ' exactly at that point, and apply the same reasoning to the two subsequences, until γ' is divided into subsequences that are all minimal reversing. Assume now that c_1 is the configuration obtained by applying t_1 to c, and c_{2s-1} is the configuration obtained by applying γ' to c_1. Using the inductive hypothesis on each of the minimal reversing subsequences of γ', we obtain that the stack and the cache of c_1 and c_{2s-1} are equal. We have already observed that a pop transition applied to c_1 would restore the stack and the cache of c. Because the stack and the cache of c_1 and c_{2s-1} are the same, we conclude that the pop transition t_{2s} applied to c_{2s-1} produces configuration c_{2s} with exactly the same stack and cache as c. □

Consider now a complete run of the parser, that is, a run starting at the initial configuration for a given input, and ending in a final configuration. Lemma 2 suggests that such run can be represented by means of some underlying tree structure, as described in what follows. Each configuration of the cache reached at some timestep in the run is a node of the tree. Each push transition descends from one node of the tree to some of its children, and each pop transition returns to the parent node. We call this underlying tree structure the **derivation tree**. The derivation tree represents the history of the parsing process that produces the output graph, and it is possible to show that the set of derivation trees associated with the runs of a cache parser on any input can be generated by a context-free grammar. This follows from the fact that our parser is a special kind of push-down automaton.[1]

1 The term "derivation tree" has been used in the literature to denote an underlying derivation process associated with a generative grammar (a rewriting system); see, for instance, the definition of tree-adjoining grammars (Joshi and Schabes 1997). Because we have a recognition device here, rather than a grammar, we are making a slight abuse of terminology.

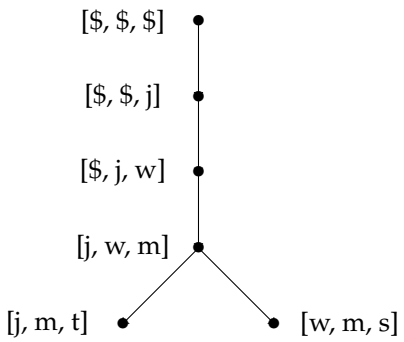


Figure 6
Derivation tree representing the run in Figure 5.

Example 4
Consider again the run displayed in Figure 5. We represent this run by means of the derivation tree displayed in Figure 6. Note that a walk through the tree that combines a preorder and a postorder visit exactly provides the sequence with the content of the cache at each timestep in the original run. Observe that each subtree of the derivation tree corresponds to a minimal reversing sequence within the run.

We now list some important properties of the derivation trees representing the runs of a cache transition parser, which are used subsequently. All these properties are direct consequences of the definition of the push and pop transitions and are rather intuitive; we therefore omit a formal proof.

1. The bag at each node contains the same items as its parent, with one vertex removed and one vertex added.
2. Every edge of the graph being built by the run of the parser can be associated with some bag that contains both of the edge's endpoints, with one of the endpoints in the m-th position of the cache.
3. The bags containing a vertex v form a connected subgraph of the tree. This is in turn a subtree rooted at the bag where the vertex is first pushed into the cache (and also eventually deleted from the cache), and having as leaves the bags where the vertex is removed from the cache and pushed onto the stack (or equivalently, the bags where the vertex is popped from the stack and pushed back into the cache).

We can now provide a characterization of the runs/derivation trees of a cache transition parser in terms of the notions of tree decomposition and width of the graph being constructed by the parser itself.

Lemma 3
Consider a cache transition parser with cache size m, and consider a run of the parser with input a vertex sequence π and with output the constructed graph G. Let T be the

derivation tree representing the run. Then T forms a smooth tree decomposition of G having width $m-1$ and having vertex order $\pi(T)=\pi$.

Proof. Properties 1 to 3 guarantee that T is a smooth tree decomposition of G. Each bag is first created by a push transition, which adds one vertex to the cache and removes one vertex from the cache. Because the bags of T have size m, the size of the cache, the width of T is $m-1$. Recall that the vertex order $\pi(T)$ is the sequence of vertices produced by visiting T in a preorder traversal and listing the vertices newly introduced at the visited bags. Since the derivation tree T is constructed depth first by pushing vertices from the input buffer into the cache, $\pi(T)$ is exactly the order of the vertices in π. □

We can also prove the inverse of the previous lemma.

Lemma 4
Consider a graph G with a smooth tree decomposition T having width $m-1$, and let $\pi(T)$ be the vertex order of T. Then T is a derivation tree of a cache transition parser with cache size m, and G is constructed by the associated run given $\pi(T)$ as input.

Proof. Let the cache transition parser take a sequence of transitions corresponding to a depth-first traversal of T, pushing an element from $\pi(T)$ into the cache each time it descends one level in T, and popping each time it ascends. Let (u,v) be an edge of G. Because T is a tree decomposition of G, there is a bag of T containing both u and v. Without loss of generality, let u be the vertex that was introduced before v along the path from the root of T to the bag containing both u and v. Let b_v be the bag at which v is introduced. Because v can only appear in bags in the subtree of T rooted at b_v, this bag containing both u and v must appear in this subtree. Furthermore, by the running intersection property, since u appears in a bag at or below b_v, and is introduced above b_v, u must appear in b_v. Thus, because bags of T correspond to the cache at each step of the parser, the parser's cache will contain u at the step at which v is pushed into the rightmost position of the cache. Therefore, the automaton can build each edge of G. □

Combining Lemmas 3 and 4, and using Lemma 1 from Section 2, we have the following main result, which is a characterization of the relative treewidth of a graph with respect to an ordering of its vertices.

Theorem 1
Let G be some graph and let π be some ordering of its vertices. The relative treewidth of G with respect to π is $m-1$ if and only if a transition parser with input π can construct G using cache size m but not using cache size $m-1$.

The computational problem of deciding whether a transition parser with cache size m and with input π can construct G is treated in Section 4. Furthermore, the problem of efficiently computing the smallest cache size m that allows a transition parser to construct G from input π is treated in Section 5.

Similarly to Theorem 1, the following result provides a characterization of the treewidth of a graph. Again, the result is a direct consequence of Lemmas 3, 4, and 1.

Theorem 2
A graph G has treewidth $m-1$ if and only if a transition parser with cache size m can construct G for some input ordering of G's vertices, and for no ordering of G's vertices a transition parser with cache size $m-1$ can construct G.

4. Oracle Algorithm

A cache transition parser is a nondeterministic automaton: For a fixed vertex sequence π, the parser could construct several graphs, all having tree decompositions with vertex order π (see Lemma 4). Even for an individual graph G, there may be several runs of the parser on π, each constructing G through a tree decomposition having vertex order π. This is usually called spurious ambiguity.

In this section we develop an algorithm that can be used to drive a cache transition parser with cache size m, in such a way that the parser becomes deterministic. This means that at most one computation is possible for each pair of G and π. More precisely, our algorithm takes as input a configuration c of the parser obtained when running on π, and a graph G to be constructed. Then the algorithm computes the unique transition that should be applied to c in order to construct G according to a canonical tree decomposition of width $m-1$ having vertex order π. If such tree decomposition does not exist, then the algorithm fails at some configuration obtained when running on π.

In the literature on transition-based parsing, algorithms of this type are called **oracles** (Nivre 2008). Oracles are used to produce training data for the parser out of gold target structures. In our case, if we are given a data set of vertex sequences paired with gold graphs, an oracle can be used to provide a set of canonical transition sequences for training a classifier to predict the best transition at each configuration. The oracle algorithm can also be used to support Theorem 1 in computing the relative treewidth of G with respect to some vertex order π. Finally, we will later use the oracle algorithm (in Section 5) to compute the minimal cache size needed to parse a given data set of gold graphs.

Let E_G be the set of edges of the gold graph G. The oracle algorithm can look into E_G in order to decide which transition to use at c, or else to decide that it should fail. This decision is based on three mutually exclusive rules, listed below. Assume that c has cache $\eta = [v_1, \ldots, v_m]$ and buffer β. The first rule is given by:

1. If there is no edge (v_m, v) in E_G such that vertex v is in β, the oracle chooses transition pop.

This rule means that, as soon as we encounter a vertex in the rightmost position of the cache with no forward-pointing edges (in the input sequence π) that are still unprocessed, we go back to the stack and attempt to process other pending vertices.

In order to introduce the remaining rules, we need to develop some additional notation. For $j \in [|\beta|]$, we write β_j to denote the j-th vertex in β. We choose a vertex v_{i^*} in η such that:

$$i^* = \underset{i \in [m]}{\text{argmax}} \ \min \ \{ j \mid (v_i, \beta_j) \in E_G \} \tag{1}$$

In words, v_{i^*} is the vertex from the cache whose closest neighbor in the buffer β is furthest to the right in β. Ties in the min and argmax operators can be resolved arbitrarily: In what follows we assume some fixed criterion for tie resolution, in order to make the parser deterministic; because all of our results are independent of the specific criterion we use, we do not further specify here our choice. We also adopt the convention that the minimum of an empty set of natural numbers is infinity. In this way, if a vertex in η has no edges pointing to vertices in β, that vertex can be selected. The main idea here is that we want to process vertices by giving higher priority to those vertices with closer forward neighbors. We therefore move out of the cache vertex v_{i^*} and push it into the stack, for later processing.

Let us now construct the set of indices that would be needed if we decide for a push transition in configuration c:

$$C = \{ i \mid i \in [m] \setminus \{ i^* \}, \ (v_i, \beta_1) \in E_G \} \tag{2}$$

The remaining two rules are then given by:

2. If Rule 1 does not apply, and there is no edge (v, β_1) in E_G such that vertex v is in the stack or $v = v_{i^*}$, the oracle chooses transition $\mathsf{push}(i^*, C)$.
3. If Rule 1 and Rule 2 do not apply, the oracle fails.

Rule 2 checks that it is possible to construct all of the backward-pointing edges from vertex β_1 using the cache. If this is not the case, it means that G cannot be produced with the given cache size, and thus the parser rejects it.

The restrictions on the transitions imposed by the oracle algorithm lead to certain properties in the tree decompositions that a cache transition parser produces when running in oracle mode. For a graph G we define an **eager tree decomposition** of G to be a smooth tree decomposition T produced by the parser running on input π in oracle mode, where π is some sequence of G's vertices. We now show that the eager tree decomposition is a normal form for tree decompositions of graphs, preserving both the width and the vertex order.

Lemma 5
Any smooth tree decomposition T of graph G can be transformed into an eager tree decomposition T' of G of equal width. Moreover, we have $\pi(T') = \pi(T)$.

Proof. Because T is a smooth tree decomposition, by Lemma 4 there exists a cache transition parser with cache size equal to the width of $T + 1$, such that a run of this

parser on $\pi(T)$ produces T. If the transitions of this run do not violate Rules 1 to 3 in the definition of our oracle, then T is also an eager tree decomposition. In case the run shows some violations of the three rules, we change T in order to eliminate these violations from the run, in a way that does not increase the width/cache size and preserves the order.

Suppose that our run contains some push transition that occurs when the rightmost vertex v in the cache η has no forward-pointing edge leading to some vertex in the buffer. This represents a violation of Rule 1 of the oracle. Let I be the set of nodes of T, and let $i \in I$ be the node of T with rightmost vertex v in the cache, to which this push transition applies; see Figure 7. If there are several push transitions out of node i, those that represent a violation of Rule 1 must all be grouped at the right. We then choose the rightmost one. Let $i_1 \in I$ be the node of T produced by this push transition, and let T_1 be the subtree of T rooted at i_1. The vertices of G that are pushed into the cache in the run associated with T_1 cannot contain any neighbor of v. Thus v is not needed in T_1. We can therefore reattach subtree T_1 to the parent node of i, $p(i)$, in such a way that i_1 becomes the immediate right sibling of i; see again Figure 7. Furthermore, we can replace all occurrences of v in T_1 with copies of the vertex introduced at $p(i)$.

Let T' be the tree resulting from the above transformation of T. Because our transformation has not changed the size of the bags of T, T' is still a smooth tree decomposition of G, with the same width as T. Since our transformation has moved T_1 one level up in T without "jumping over" any other subtree of T, we must have $\pi(T') = \pi(T)$. Note that this transformation of T has removed from our run the alleged violation of Rule 1.

Suppose now that our run violates Rule 2 of the oracle. Because the run produces G, this can only happen if the parser does not push into the stack the vertex from the cache that will be needed furthest in the future. Let then v_1 be the vertex that is pushed onto the stack, and let $v_2 \neq v_1$ be the vertex that is needed furthest in the future. Let also $i_1 \in I$ be the node of T that is created at this step, and let T_1 be the subtree of T rooted at i_1. Because v_1 is removed from the cache when i_1 is created, v_1 does not appear anywhere in T_1, and none of the vertices that are pushed in T_1 are neighbors of v_1 in G. If v_1 is not a neighbor of the vertices that are pushed in T_1, then v_2 cannot be a neighbor of these vertices either, since v_2's first neighbor occurs strictly after v_1's first neighbor

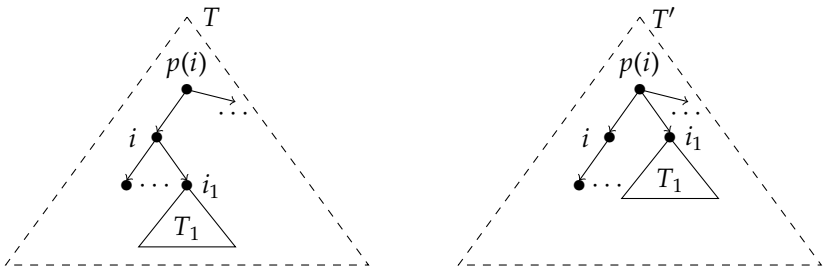


Figure 7
Transformation of tree decomposition T (left) into T' (right), eliminating a push transition at node $i \in I$ that violates Rule 1. Node $i_1 \in I$ is the rightmost child of i in T and is the immediate right sibling of i in T'.

in β. Therefore, although v_2 appears in subtree T_1, it is never used there to construct an edge of G. All occurrences of v_2 in T_1 can then be replaced by occurrences of v_1.

Let T' be the tree resulting from the second transformation above. Again, tree T' is a smooth tree decomposition of G, with the same width as T. Furthermore, the replacement of v_2 by v_1 does not affect the bags of T where these nodes have been introduced for the first time. Therefore we must have $\pi(T') = \pi(T)$. Note that this transformation of T has removed from our run the alleged violation of Rule 2.

These two transformations can be iterated until the resulting tree is an eager tree decomposition. From these observations, this tree has the same width and vertex order as T. □

We can now prove the correctness of our oracle. We say that a cache transition parser running in oracle mode **accepts** its input if it reaches a final configuration.

Theorem 3
Let G be some graph and let π be some ordering of its vertices. Assume that the relative treewidth of G with respect to π is $m - 1$. Then a cache transition parser with cache size m running in oracle mode on input G and π will accept.

Proof. By Lemma 5, there exists an eager tree decomposition T for G of width $m - 1$ such that $\pi(T) = \pi$. By definition of eager tree decomposition, a cache transition parser with cache size m can run in oracle mode on input G and π, without any violation of Rules 1 to 3. The parser will then accept. □

We conclude this section with a computational analysis of the cache transition parser running in oracle mode. Let G and π be the input to the parser, and assume the cache size is m. Each pop transition can be carried out in constant time. Each push transition involves the processing of $m - 1$ vertices from the cache, testing their connection in G to the vertex shifted into the cache. This can be easily carried out in total time $\mathcal{O}(m)$.

We now consider the computation of Rules 1 to 3 of the oracle at each step of the parser. We preprocess G in such a way that, for each vertex v, we have an adjacency list $a(v)$ with all of v's neighbors, sorted according to the left-to-right order in which these vertices appear in π. All of the adjacency lists together can be computed in time $\mathcal{O}(|G| \log(d))$, where d is the maximum degree of a vertex of G. The main idea, explained in more detail subsequently, is to remove from each $a(v)$ the vertices as soon as the associated edges are processed. In this way, at each timestep, each $a(v)$ is an ordered list of the unprocessed neighbors of v. These vertices must necessarily appear in the buffer.

Assume the current configuration has cache $[v_1, \ldots, v_m]$. The computation of the oracle rules can be carried out as follows.

- To compute Rule 1, it suffices to check whether $a(v_m)$ is empty, because any unprocessed neighbor of v_m must necessarily be located in the buffer. This takes time $\mathcal{O}(1)$.
- To compute Rule 2, we consider each vertex v_i in the cache. If $a(v_i)$ is empty, we assign to v_i a score of $+\infty$. Otherwise, let v be the first vertex in

$a(v_i)$, and let i_v be the index of v in π. We then assign to v_i a score of i_v. According to Equation (1), we can now compute index i^* by finding the vertex in the cache with the maximum score, arbitrarily solving any tie. This can be done in time $\mathcal{O}(m)$.
Next, we need to compute set C as defined in Equation (2). Let v be the first vertex in the buffer. We check that the backward neighbors in $a(v)$ are all in the cache and do not include vertex v_{i^*}, as required by Rule 2. This again can be done in time $\mathcal{O}(m)$.

- Finally, Rule 3 trivially takes time $\mathcal{O}(1)$.

To conclude our analysis, we need to consider the amount of time spent in the updating of the adjacency lists. This is done right after each push transition, when a vertex v is shifted from the buffer into the cache. We observe that, at this time, all of the backward neighbors in $a(v)$ must be in the cache, otherwise the push transition would not be possible and the computation would fail by Rule 3. We can then remove these backward neighbors from $a(v)$ in time $\mathcal{O}(m)$. Symmetrically, for each vertex v' that is removed from $a(v)$, we also remove v from $a(v')$. Note that v is always the first element of $a(v')$, and can thus be removed in time $\mathcal{O}(1)$.

To summarize, at each step in the parsing process, we check Rules 1 to 3 of the oracle, we perform the required transition, and we update all of the adjacency lists in total time $\mathcal{O}(m)$. The parser makes exactly one push and one pop transition for each arc of the eager tree decomposition of G given vertex order π. Because the number of arcs is $|\pi|$, the processing time (excluding the initialization of the adjacency lists) is $\mathcal{O}(|\pi|m)$.

Combining the initialization and the processing time, we have the following result.

Theorem 4
Let graph G and vertex ordering π be the input to a cache transition parser with cache size m, running in oracle mode. Let also d be the maximum degree of a vertex of G. A run of the parser takes time $\mathcal{O}(|G| \log(d) + |\pi|m)$.

As already discussed, this computational result refers to the training phase, where we use the oracle to map gold graphs and orderings into canonical transition sequences for training a classifier that would choose the optimal transition when decoding strings into graphs. As for the decoder itself, because there is no need to compute Rules 1 to 3 of the oracle or to initialize the adjacency lists, the running time will be $\mathcal{O}(|\pi|m)$ plus some function that accounts for the time for the computation of the classifier, which we do not deal with here.

As a second remark, in case we have a small value for the cache size m, the decoding time $\mathcal{O}(|\pi|m)$ is very close to the linear time of a transition-based system for dependency tree parsing. More precisely, when $m = 2$ our parser will only be able to build trees (see discussion in Section 7.2). On the other extreme, when $m = |\pi|$, the parser will become the transition-based implementation of the Covington algorithm (Nivre 2008) generalized to graphs. This algorithm is able to parse arbitrary graphs, and will run in quadratic time in the length of the input. In the next section we discuss an algorithm that computes the minimal value of m for an input set of data. As we will

see in Section 6, on real data for English we obtain values of m that are very small. This suggests that the graphs of interest for semantic representation of English sentences can be processed almost as efficiently as their syntactic dependency tree counterpart, when the vertices are provided according to the English order.

5. Computing Minimal Cache Size

We now examine the problem of computing the relative treewidth of a graph G with respect to an order π. As already seen in Theorem 1, this provides the smallest cache size needed by our parser in order to process π and produce G. This problem is also central in parsing applications: Its solution will allow us to compute in Section 6 the minimal cache size that guarantees a complete coverage of a given data set.

Let T_1 and T_2 be two smooth tree decompositions for the same graph G. We say that T_1 and T_2 are **m-equivalent** if the following conditions both hold.

- T_1 and T_2 have the same branching structure, that is, T_1 and T_2 are the same if we ignore the content of the bags at their nodes.
- Corresponding nodes of T_1 and T_2 introduce the same vertex of G.

Intuitively, T_1 and T_2 are m-equivalent if they differ only in the choice of the vertices of G that are dropped off at corresponding bags. As a direct consequence of the definition, we have that if T_1 and T_2 are m-equivalent, then $\pi(T_1) = \pi(T_2)$.

Lemma 6
Let G be a graph and let π be a vertex order for G. When running in oracle mode on G and π, transition parsers with different cache sizes have associated eager tree decompositions that are m-equivalent.

Proof. We start by showing that, regardless of the size of the cache, when parsing in oracle mode, the sequence of push and pop transitions is always the same, and at corresponding timesteps of parsers with different cache size, the vertex in the rightmost position of the cache is always the same. To do this, we use induction on the number of moves, and we take advantage of the fact that, when parsing in oracle mode, the sequence of push and pop transitions depends only on the rightmost vertex in the cache and on the current position in the input buffer (see Rule 1 of our oracle).

Suppose that the first $h-1$ moves for parsers of two different cache sizes, both producing G, consist of the same sequence of push and pop transitions (although the vertices chosen to be removed from the cache and pushed onto the stack may differ). Assume also that the rightmost vertex in the cache is the same for each of the first $h-1$ moves for both parsers. Because both parsers have pushed the same number of times, both parsers will be at the same location in the buffer. Because the choice of push or pop depends only on the rightmost vertex in the cache and the position in the buffer, the choice of push or pop at step h will be the same for both parsers. If both parsers push, they will both shift the same vertex from the buffer, and will place the same vertex in the rightmost position of the cache. If both parsers pop, they will both return at timestep h to

the cache configuration that they had at some previous timestep $i < h$. By the induction hypothesis, the cache configuration will have the same rightmost vertex.

Because parsers of any cache size running in oracle mode on G and π follow the same sequence of push and pop transitions, for all these runs the associated derivation trees and eager tree decompositions have the same branching structure. Furthermore, because corresponding nodes in these derivation trees have cache configurations with the same rightmost vertex, corresponding nodes of the tree decompositions introduce the same vertex of G. We thus conclude that the associated tree decompositions are all m-equivalent. □

The next result shows an easy lower bound on the width of a smooth tree decomposition.

Lemma 7
Let τ be a subtree of a smooth tree decomposition T of graph G, and let h be the number of vertices of G introduced outside τ that are adjacent in G to vertices introduced inside τ. Then the width of T is at least h.

Proof. Each vertex is introduced in the topmost node of T in which it appears, so vertices introduced in τ appear only in τ. Each edge e of G incident on a node introduced inside τ must be assigned to a bag of T inside τ. If the other endpoint of e is introduced outside of τ, then the other endpoint must occur both inside and outside τ, and, by the running intersection property of tree decompositions, must occur in the bag B at the root of τ. If there are h such distinct endpoints, B must contain these h vertices and the vertex introduced at B, for a total size of $h + 1$ vertices. Therefore, the width of T is at least h. □

The combination of Lemmas 6 and 7 leads to an efficient algorithm for finding the relative treewidth of G with respect to π, reported below. In the algorithm we use the following property. Let T be a smooth tree decomposition of G, and let τ be a subtree of T. Let also v be a vertex of G that is introduced outside of τ and that is adjacent in G to some vertex introduced inside τ. Then v must be introduced at some node of T that dominates the root of τ. To see this, consider that v is introduced at the topmost node of T in which it appears, since T is smooth. Furthermore, by the running intersection property this node must dominate the root of τ.

Algorithm 1 Procedure for determining the relative treewidth of G with respect to π

```
1: procedure OrderedTreewidth(G = (V_G, E_G), π)
2:     Run in oracle mode on G and π a cache transition parser with cache size |V_G|
3:     Let T be the resulting tree decomposition, with I its node set
4:     return max_{i∈I} |{u | (u, v) ∈ E_G, u introduced above i,
           v introduced at i or below i}|
```

The next result proves the correctness of Algorithm 1.

Theorem 5
Let graph G and vertex order π be the input to Algorithm 1. Then the algorithm returns the relative treewidth of G with respect to π.

Proof. Assume that Algorithm 1 returns integer k. We start by showing that there exists an eager tree decomposition T of G such that $\pi(T) = \pi$ and the width of T is k. Let T_a be the eager tree decomposition produced at Step 3 of Algorithm 1. We construct a tree decomposition T'_a by copying the branching structure of T_a and by editing each of the bags of T_a as described in what follows.

Let i be a node of T_a and let X_i be the associated bag, introducing vertex v_i of G. We replace X_i with the bag

$$X'_i = \{v_i\} \cup \{u \mid (u, v) \in E_G,\ u \text{ introduced above } i,\ v \text{ introduced at } i \text{ or below } i\} .$$

It is not difficult to see that for each $i \in I$ we have $X'_i \subseteq X_i$.

We now argue that T'_a is a valid tree decomposition of G. First, note that every vertex of G is introduced at some bag of T'_a. More precisely, if v is introduced at bag X_i of T_a, for some $i \in I$, then v is introduced at the corresponding bag X'_i of T'_a. Furthermore, each vertex v of G appears in a connected subtree of T'_a. To see this observe that if $v \in X_i$ is dropped from X'_i, for some $i \in I$, then v will not appear in any of the bags X'_j for nodes j that are dominated by i. Finally, each edge of G can be assigned to the bag that introduces the lower of its two endpoints (as noted above, the node introducing one endpoint must be an ancestor of the node introducing the other).

Because T_a and T'_a have the same branching structure, and because vertices of G are introduced at corresponding nodes in T_a and T'_a, we have that $\pi(T'_a) = \pi(T_a) = \pi$. Note that each X'_i is constructed following essentially the same condition at Step 4 of Algorithm 1, which provides value k. Hence the largest bag of T'_a has size $k + 1$ and T'_a has width k. By Lemma 1, T'_a can be transformed into a smooth tree decomposition of width k, preserving the order, and by Lemma 5 this smooth tree decomposition can in turn be transformed into an eager tree decomposition of width k, again preserving the order.

Let us now assume that the relative treewidth of G with respect to π is $k' < k$. From Theorem 3, we have that a cache transition parser with cache size $k' + 1$ running in oracle mode on G and π will accept. Let T' be the eager tree decomposition associated with the run of the parser, and let T be the eager tree decomposition at Step 3 of Algorithm 1. By Lemma 6, T and T' are m-equivalent.

Because corresponding nodes of T and T' introduce the same vertex of G, we have that Step 4 of Algorithm 1 would return the same value k when running on T'. We can then apply Lemma 7 to T', and conclude that T' has width at least k. However, by Lemma 3, T' will have width at most $k' < k$. Since this is a contradiction, it cannot be possible for a tree decomposition of G to have order π and to have width smaller than k. □

We conclude this section with a computational analysis of Algorithm 1. Following Theorem 4, Step 2 of the algorithm takes time $\mathcal{O}(|G| \log(d) + |V_G|^2)$, where V_G is the

set of G's vertices (with $|V_G| = |\pi|$). To compute Step 4, let I be the set of nodes of T. For each $i \in I$ we maintain a list of vertices introduced above i that are connected to nodes introduced below i. This list can be computed in time $\mathcal{O}(|V_G|)$ using the lists at the children of i. The whole step then takes time $\mathcal{O}(|V_G|^2)$. We have thus shown the following result.

Theorem 6
Let graph G and vertex order π be the input to Algorithm 1. Let also d be the maximum degree of a vertex in G. Algorithm 1 can be implemented to run in time $\mathcal{O}(|V_G|^2 \log(d))$.

6. Experiments

In this section we consider several families of graph-based representations of semantic structures for natural language that are commonly used nowadays. We run experiments on graph data sets for these representations, with the aim to assess the coverage that our cache parser provides with different cache sizes.

We first evaluate our algorithm on Abstract Meaning Representation (AMR) (Banarescu et al. 2013). AMR is a semantic formalism where the meaning of a sentence is encoded as a rooted, directed graph. Figure 8 shows an example of an AMR graph in which the nodes represent the AMR concepts and the edges represent the relations between the concepts they connect. AMR concepts consist of predicate senses, named entity annotations, and in some cases, simply lemmas of English words. AMR relations consist of core semantic roles drawn from the Propbank (Palmer, Gildea, and Kingsbury 2005) as well as very fine-grained semantic relations defined specifically for AMR. We use the training set of LDC2015E86 for SemEval 2016 task 8 on meaning representation parsing (May 2016), which contains 16,833 sentences. This data set covers various domains including newswire and Web discussion forums.

For each graph, we derive a vertex order corresponding to the English word order by using the automatically generated alignments provided with the data set, which

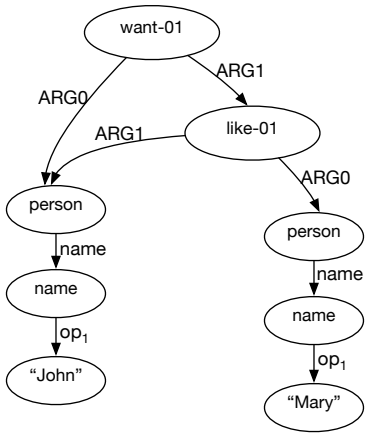


Figure 8
An example AMR graph for the sentence "John wants Mary to like him."

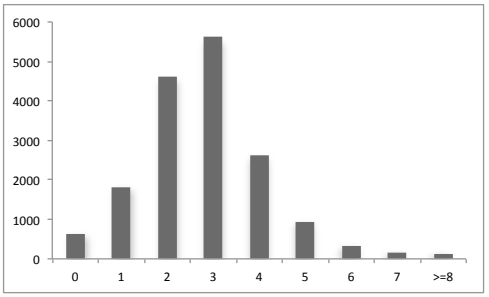


Figure 9
The distribution of AMR relative treewidth.

align tokens in the string to concepts or edges in the graph. We first collapse subgraphs of named entities and dates to a single node on the graph side. For example, the subgraph corresponding to "John" is collapsed to a single node "person+John," and the same goes for the subgraph for person name "Mary." There are also some vertices in the graph that are not aligned to any token. We want to linearize all vertices (concepts) in the graph in such a way that the string side order is kept as much as possible (we call it string order). We first sort the aligned vertices according to the position of their token side. We use the first position in case a vertex is aligned to multiple positions. If an unaligned vertex is the parent of an aligned vertex, we insert it right before the aligned vertex in the sequence. Otherwise, for simplicity, we append the unaligned vertex to the end of the vertex sequence, according to its relative order in the depth-first traversal of the graph.

After we have constructed the input vertices with vertex order π, we run Algorithm 1 to determine the relative treewidth of each AMR graph with respect to the vertex order π. Figure 9 shows the distribution of relative treewidth of AMR graphs in the data set. We can see that over 99% of the AMR graphs can be built using a cache size of 8. As shown in Table 1, the average relative treewidth with respect to the string order is 2.80. The average treewidth of this data set (i.e., the average of minimum relative treewidth of each AMR graph with respect to any vertex order) is 1.52. This shows that using the string order as a constraint does not significantly increase the treewidth statistics of the data set.

Table 1
Relative treewidth statistics with respect to different vertex orders: "real" means the real treewidth of the data, "string" means using the string order constraint, "gold" means using the gold alignment.

data	real	string (gold)	string	reversed string (gold)	reversed string	random
LDC2015E86	1.52	-	2.80	-	3.08	4.84
hand aligned	1.43	2.61	2.68	2.79	2.90	4.81

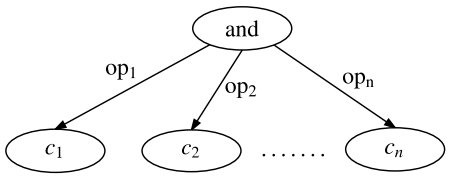


Figure 10
The AMR subgraph representation for the substring $c_1, c_2, \ldots$ and c_n.

In the worst case, the maximum relative treewidth of a graph can be 16, while the maximum treewidth of the data is 4. This is because when using the string order as a constraint for the vertex order, the string-to-vertex alignment does not always follow the preorder traversal of vertices that is desirable in the width computation. The most problematic case is the traversal of a node with many branches, with op's structure most significant in the AMR data. For example, in Figure 10, n different concepts are connected to the parent concept "and" with the op_k ($k = 1, \cdots, n$) relation. This structure does not introduce high treewidth because we can put "and" and each c_i into a separate bag, forming a chain of bags of width 1. However, when we use the string order as a constraint, we first introduce vertices $c_1, c_2, \ldots, c_{n-1}$, then we further introduce "and" and c_n. This would result in a chain structure of length $(n + 1)$ in the tree decomposition, where "and" is introduced at the n-th bag in the chain. According to Algorithm 1, because $c_1, c_2, \ldots, c_{n-1}$ are all introduced above the bag that introduces "and" and all connect to "and," the relative treewidth is at least $n - 1$. In general, a high-branching structure with most children introduced before the parent would result in larger relative treewidth and distort from the real treewidth of the graph.

Another reason for high relative treewidth is the alignment errors from the automatic alignments. When multiple instances of the same word align to multiple vertices with the same concept labels, the automatic alignment usually cannot distinguish them and often creates a many-to-many alignment between instances of the word in the string and instances of the concept in the graph. This results in a wrong traversal order of the multiple vertices and a larger relative treewidth. Our worst-case sentence, with relative treewidth of 16, is due to this type of error in the automatically generated alignments.

We additionally experiment on a smaller data set of 200 hand-aligned AMR/English sentence pairs by Pourdamghani et al. (2014). From Table 1, we can see that the average relative treewidth of these AMR graphs with respect to the string order is 2.61 when using the gold alignment, and the average treewidth is 1.43. If we use the automatic alignment for these AMRs, the relative treewidth becomes 2.68. The maximum relative treewidth for both cases are 6. This number is much lower than the maximum relative treewidth of the LDC2015E86 training data because the maximum sentence length of the smaller data set is 54, whereas for the latter data set the maximum sentence length can be as large as 225. By comparison, we can also see that alignment errors can result in higher relative treewidth, though not significantly.

We also evaluate the impact of different vertex order on the relative treewidth. We can see from the table that if we reverse the vertex order (reversed string order), the

relative treewidth is 3.08. This number is slightly larger than using the string order. The reason might be that English is more likely to have relation arcs going from left to right. If we randomize the vertex order, the relative treewidth becomes 4.84.

We also evaluate the coverage of our algorithm on semantic graph-based representations other than AMR. We consider the set of semantic graphs in the Broad-Coverage Semantic Dependency Parsing task of SemEval 2015 (Oepen et al. 2015), which uses three distinct graph representations for English semantic dependencies.

- **DELPH-IN MRS-Derived Bi-Lexical Dependencies (DM)**: These semantic dependencies are derived from the annotation of Sections 00-21 of the WSJ Corpus with gold-standard HPSG analyses provided by the LinGO English Resource Grammar (Flickinger 2000; Flickinger, Zhang, and Kordoni 2012). Among other layers of linguistic analysis, this representation also includes logical-form meaning representations in the framework of Minimal Recursion Semantics (MRS) (Copestake et al. 2005).
- **Enju Predicate-Argument Structures (PAS)**: This data set comes from the HPSG-based annotation of Penn Treebank, which is used for training the wide-coverage HPSG parser Enju (Miyao 2006). Enju can effectively analyze syntactic/semantic structures of English sentences and output phrase structures and predicate-argument structures with a wide-coverage grammar and a probabilistic model trained on this data.
- **Prague Semantic Dependencies (PSD)**: The Prague Czech-English Dependency Treebank (Hajic et al. 2012) is a set of parallel dependency trees over the WSJ texts from the Penn Treebank, and their Czech translations. The PSD bi-lexical dependencies have been extracted from what is called the **tectogrammatical** annotation layer (t-trees). We experiment on the English part of the treebank in this article.

Differently from the case of AMR, in all of these graph representations the vertices are exactly the tokens from the input string. From Table 2 we can see that using English word order as the vertex order, the average relative treewidth for the set of DM graphs is 2.95, while the average real treewidth of the DM graphs is 1.30. The PSD and PAS graphs have larger real treewidth in comparison with the DM graphs, and the relative treewidth is also larger. We also observe that even though DM has smaller average

Table 2
Relative treewidth statistics for SemEval 2015 semantic dependency graphs.

data	real	string
AMR	1.52	2.80
DM	1.30	2.95
PSD	1.61	3.01
PAS	1.72	3.84

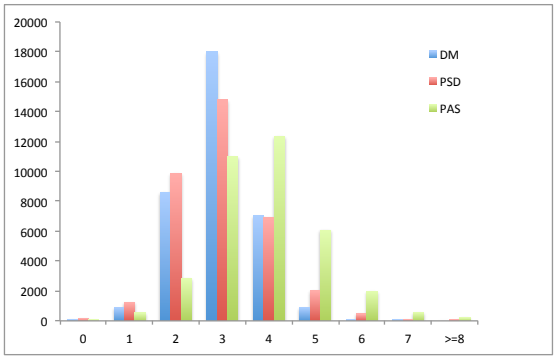


Figure 11
The distribution of relative treewidth for SemEval 2015 semantic dependency graphs.

real treewidth in comparison with AMR, the relative treewidth is slightly larger. This is because articles, prepositions, and other non-content words appear as vertices in DM graphs, whereas in AMR these words appear either as edges or not at all. Articles in particular increase the relative treewidth, because the article appears as a left neighbor of the head noun in the semantic dependency graphs, and the number of left neighbors of a vertex is a lower bound on relative treewidth, as discussed earlier in relation to conjunctions in AMR. We find that whereas most AMR graphs have relative treewidth of 2 or 3 (shown in Figure 9), most semantic graphs in the Semantic Dependency Parsing data sets have slightly larger relative treewidth (3 for DM and PSD, and 3 or 4 for PAS, shown in Figure 11). With a cache size of 8, the coverage rate is 100% for DM and over 99% for PSD and PAS. In practice, we can tune the cache size to optimize the trade-off between coverage and the ease of prediction.

To summarize, we find that the relative treewidth of the analyzed semantic graph-based structures, with respect to the English word order, is low enough to make efficient parsing possible. Furthermore, the fact that the real word order results in lower relative treewidth than random orders, or even the reverse order, indicates that the real English word order provides valuable information that our parsing framework can exploit.

7. Comparison with Other Formalisms

In this section we compare our cache transition parser with existing formalisms that have been used for graph-based parsing, as well as to similar transition-based systems for dependency tree parsing.

7.1 Connection to Hyperedge Replacement Grammars

Hyperedge Replacement Grammars (HRGs) are a general graph rewriting formalism (Drewes, Kreowski, and Habel 1997) that has been applied by a number of authors to semantic graphs such as AMRs (Jones et al. 2012; Jones, Goldwater, and Johnson 2013; Peng, Song, and Gildea 2015). Our parsing formalism can be related to HRG through the concept of tree decomposition.

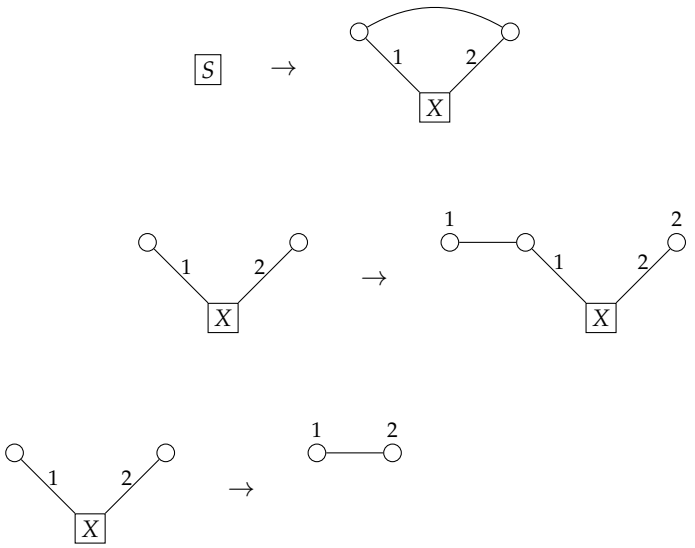


Figure 12
An HRG that generates cycles of any size. Squares indicate grammar nonterminals with numbered ports. Number above vertices in a rule's right-hand side correspond to the ports of the lefthand side nonterminal. An example derivation is shown in Figure 13.

HRGs contain rules that rewrite a nonterminal hyperedge into a graph fragment consisting of a number of new nonterminal hyperedges and terminal edges (see the example grammar in Figure 12). The vertices that a nonterminal hyperedge connects to, known as its **ports**, are also the points at which the rule's right-hand side graph fragment is attached to the rest of the graph after the nonterminal is rewritten. An HRG derivation of a graph can be viewed as a derivation tree, with a grammar rule at each node, where the rule expanding a nonterminal hyperedge is a child of the rule that introduced the nonterminal in its righthand side. This derivation tree provides a tree decomposition of the derived graph. More precisely, the tree decomposition has the same nodes and the same branching structure as the derivation tree, and each bag in the tree decomposition contains all the vertices that appear in the right-hand side of the rule.

In the other direction, it is possible to extract HRG rules from a tree decomposition by treating each bag as a rule. Nonterminals in the extracted rules correspond to the arcs of the tree decomposition. For an arc (i, j) of the tree decomposition, let X_i and X_j be the bags at nodes i and j, respectively. The set of vertices $X_i \cap X_j$, known as the arc's separator, are the vertices to which the corresponding HRG nonterminal was connected before being rewritten.

Example 5
An HRG generating graphs consisting of a single cycle of any size is shown in Figure 12. An example derivation of this grammar along with the corresponding tree decomposition is shown in Figure 13. Cycles have treewidth 2, as seen from the fact that the largest bag in the figure has size 3, and the grammar rules involve at most three vertices.

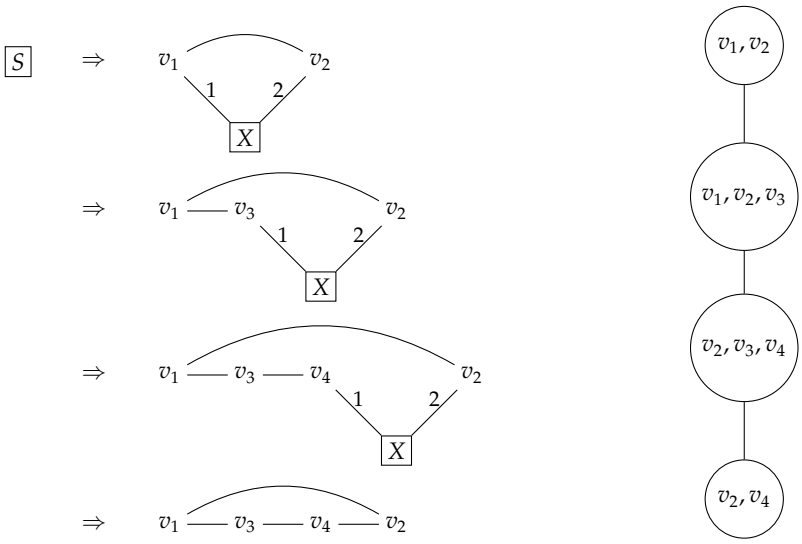


Figure 13
An example derivation of the HRG from Figure 12 (left), along with the corresponding tree decomposition of the derived graph (right). Each bag of the tree decomposition corresponds to one rule application in the derivation.

Because each run of our parser corresponds to a smooth tree decomposition, it is possible to describe a corresponding HRG. This HRG will have a number of specific properties. Because our tree decompositions are smooth, the HRG rules will always introduce exactly one new vertex in their right-hand side. Because the separators of a smooth tree decomposition of treewidth k all contain k vertices, the nonterminals of the derived HRG all have exactly k ports. The branching factor of a node in our tree decomposition corresponds to the number of right-hand side nonterminals in the corresponding HRG rule, and is potentially unlimited.

In general, the branching factor of the tree decompositions produced by our parser is not bounded by a constant, while any fixed HRG has a maximum number of right-hand side nonterminals in its rules. This implies that, although it is possible to extract an HRG corresponding to one run of our parser, it is not always possible to produce an HRG whose derivations correspond to all possible runs of a parser. We emphasize that these observations apply to the derivations of the HRG rather than to the language of graphs produced. It is an open question whether all the graphs of fixed relative treewidth with respect to a vertex order can be generated by a fixed HRG.

7.2 Connection to Existing Transition-Based Systems

As already mentioned in the Introduction, transition systems using a stack data structure have been very successful in dependency tree parsing, and several proposals can be found in the literature. In this section we compare some of these systems with our cache transition parser.

We have already mentioned that when we use a cache with size 2, we can only construct graphs that are trees. This follows from the fact that any graph with at least one cycle has relative treewidth larger than one, and thus cannot be parsed with cache size 2, by Theorem 1. When the cache size is bounded by two, the edge construction operations that are available to the parser resemble the left-arc and the right-arc transitions of the arc-standard parser described by Nivre (2008), if we disregard the fact that these transitions remove the dependent vertex from the stack. However, this similarity is only superficial and the two parsers are incomparable, as explained subsequently.

The arc-standard parser can construct trees in which the root is at the rightmost position of the input string, and all of the remaining tokens are left children of the root. As already discussed in Section 6, such trees have relative treewidth proportional to the length of the string. By Theorem 1, these trees cannot be parsed with cache size of 2.

In the opposite direction, when using cache size of 2 we can construct non-projective trees, something that is not possible with an arc-standard parser. As a simple example, consider the non-projective dependency tree G shown in the left part of Figure 14, where we have disregarded the edge directions. The derivation tree in the right part of Figure 14 displays a run of a parser with cache size 2 constructing G. When vertices v_1 and v_2 are in the cache for the first time, the parser constructs the edge (v_1, v_2). It then pushes v_3 into the cache, moving v_2 out of the cache and into the stack and constructing the edge (v_1, v_3). It then pops v_3 from the cache, moving back to the configuration with cache content v_1, v_2. Next, the parser pushes v_4 into the cache, moving v_1 into the stack and constructing the edge (v_2, v_4). Afterward, it pops v_4 from the cache, again moving back to the cache content v_1, v_2. Two more pops conclude the computation.

The fact that we can build non-projective trees, even with the minimum cache size of 2, is explained by the capability of our parser to use the cache to "reorder" vertices with respect to their original ordering in the input. In our example, for instance, at some step in the computation we have v_2 in the stack and v_1, v_3 in the cache in the given order. This reordering of the input tokens is somehow reminiscent of the parsing model proposed by Nivre (2009), where a special transition called swap is used to reorder the input tokens and to construct non-projective dependency trees.

An alternative model for parsing non-projective dependency trees has been proposed by Attardi (2006). This parser constructs edges by connecting vertices in the stack that are not at adjacent positions, using special transitions called left-arc$_k$ and right-arc$_k$

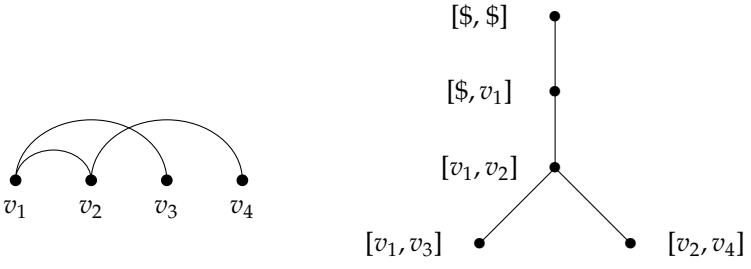


Figure 14
Non-projective tree G (left), and derivation tree of a parser with cache size two constructing G (right).

for arbitrarily large values of integer $k > 0$. More precisely, left-arc$_k$ and right-arc$_k$ create an edge between the topmost vertex in the stack and the vertex at position $k + 1$. (In the original formulation of the parser, the two vertices involved are the first vertex in the buffer and the vertex at position k in the stack. If we consider the first element of the buffer as an additional stack element sitting on the top of the top-most stack symbol, the two formulations are equivalent.) The left-arc$_k$ and right-arc$_k$ transitions are superficially similar to the operations for edge construction that we exploit in our cache, since both operations connect vertices that are not at adjacent positions. However, the two models present two substantial differences, as discussed below.

First, the left-arc$_k$ and right-arc$_k$ transitions remove the dependent vertex from the stack, while vertices in the cache are retained after edge construction. This feature allows us to create loops in the produced graphs, or edge re-entrancies in case we produce directed graphs. Second, and most important, in the cache parser there is an interplay between the stack and the cache that allows us to reorder vertices in the stack with respect to their original ordering in the input, as already observed. This is not possible in Attardi's parser. In different words, the cache should not be regarded as a finite size window at the top of the stack, as the case of Attardi's parser might suggest: In the cache parser we can pick up any vertex from the cache and move it into the stack. Hence the stack is effectively used to "delay" the processing of vertices, if these vertices are not needed in the current branch of the computation.

The cache transition parsing proposed in this article can also be related to the two-register transition system of Pitler and McDonald (2015) for non-projective dependency parsing. In addition to the buffer and the stack, a two-register transition system uses two registers to store input vertices and to create edges involving these vertices and vertices in the stack. Because the stack and the registers are manipulated independently one of the other, this technique basically alters the input order of the tokens, making it possible to produce non-projective trees. The cache parser can then be viewed as a generalization of the two-register transition systems. This is because in a cache parser one can move tokens in and out of the cache repeatedly, as already discussed. This is not possible in a register transition system. It would be interesting then to explore the use of our cache parsers for non-projective dependency grammars.

We conclude this section with a discussion of other transition-based systems explicitly designed for graph parsing, as opposed to tree parsing. Sagae and Tsujii (2008) have possibly been the first authors to extend the stack-based transition framework for dependency tree parsing to directed acyclic graphs, with the motivation of representing semantically motivated predicate-argument relations and anaphoric references. This is done by dropping the constraint of a single head per word, and by using post-processing transformations that introduce non-projectivity. Titov et al. (2009) and Henderson et al. (2013) present a transition system for synchronous syntactic-semantic parsing, with the motivation of modeling the syntax/semantic interface. On the semantic side, their system mainly captures the predicate-argument structure and semantic role labeling. Their model has then been adapted by Du et al. (2014) for semantic-only parsing.

Later, Wang, Xue, and Pradhan (2015) proposed a transition system for AMR parsing. Unlike traditional stack-based transition parsers that process input strings,

this system takes as input a dependency tree and processes its edges using a stack, applying tree-to-graph transformations that produce a directed acyclic graph. Similarly to Sagae and Tsujii (2008), the system presented by Damonte, Cohen, and Satta (2017) extends standard approaches for transition-based dependency parsing to AMR parsing, allowing re-entrancies. Similar extensions of transition-based systems to AMR parsing also appear in Zhou et al. (2016) and Ribeyre, de La Clergerie, and Seddah (2015).

All of these approaches are based on the idea of extending the transition inventory of standard transition-based dependency parsing systems in order to produce graph representations. On a theoretical perspective, what is missing from these proposals is a mathematical characterizaton of the set of graphs that can be produced and, with few exceptions, a precise description of the oracle algorithms that are used to produce training data from the gold graphs. Furthermore, all of these proposals still retain the stack and buffer architecture of the transition-based dependency parsing system they extend. In contrast, the proposal in this article introduces the novel idea of using a cache component in stack-based transition systems. As we have already discussed, the specific interplay between the stack and the cache allows the system to split the computation into different branches, and for each branch to reorder the input tokens in a way that allows edge processing locally to the cache, even in cases where the involved vertices are at a long distance in the input sequence. We have also provided a mathematical characterization of the graphs that can be constructed in this way, in terms of the novel notion of relative treewidth, and we have specified and analyzed an oracle algorithm to produce training data from the gold graphs.

8. Concluding Remarks

Our transition system is motivated by the task of semantic parsing of natural language sentences, and we now proceed to discuss some of the issues that still need to be addressed in developing a practical system based on our framework. The primary task is to develop a machine learning system for predicting the parser's next action at each step. The optimal cache size will need to be determined empirically, as it may be beneficial to trade off coverage of the small number of sentences requiring large cache size in order to make the prediction of parser actions more accurate. We speculate that it will be desirable to decompose the push action into steps that first make the decision of whether to push or pop, and then whether to build each of the potential arcs within the cache individually, in order to reduce the space of predictions at each step. In the literature on dependency grammar parsing, models of this type are called **arc-factored models** and are frequently used. Further experimentation will be required to determine the best set of features and the best architecture for the machine learning component.

A possible extension of our framework is the development of a dynamic programming algorithm to allow efficient exploration of the space of possible runs of a parser on an input string. Intuitively, different runs on the same string might share common subparts. These subparts can be computed only once, and then "shared" among different runs using dynamic programming techniques. Dynamic programming algorithms for transition-based dependency parsing have been proposed by Huang and Sagae (2010) and Kuhlmann, Gómez-Rodríguez, and Satta (2011). These algorithms

could be extended to our system, which is also fundamentally stack-based. Dynamic programming algorithms simulating transition-based parsers have proven useful in the realization of so-called dynamic oracles (Goldberg, Sartorio, and Satta 2014) for transition-based parsers, improving parsing performance with respect to static oracles, that is, oracles of the type discussed in Section 4. Furthermore, dynamic programming algorithms are at the basis of the development of methods for unsupervised learning, as for example the inside-outside algorithm (Charniak 1993).

Although we have treated the input buffer as an ordering of the vertices of the final graph, this is a simplification of the problem setting of semantic parsing for NLP. Given as input a sequence of English words, the parser must also predict which words correspond to zero, one, or more vertices of the final graph, and possibly insert vertices not corresponding to any English word. This could be accomplished either by preprocessing the input string with a separate concept identification phase (Flanigan et al. 2014), or by extending the actions of the transition system to include moves inserting new vertices into the graph. We have not included moves inserting new vertices, in order to simplify our exposition, but such moves would not fundamentally alter the correspondence between parsing runs and tree decompositions described in this article.

The correspondence between runs of our parser and tree decompositions of the output graph allows for a precise characterization of the class of graphs covered, as well as simple and efficient algorithms for providing an oracle sequence of parser moves, and for determining the minimum cache size required to cover a data set. We find experimentally that semantic graphs have low relative treewidth with respect to English word order, indicating that our parsing approach provides a practical method of exploiting the word order in semantic parsing. Our concept of relative treewidth with respect to a vertex order appears to be new in the graph theory literature, and may have applications outside of natural language processing. Our transition system was primarily motivated by these theoretical considerations, and many other definitions are possible. In particular, our decision that vertices can only be popped from the rightmost position in the cache simplifies our analysis. Theoretical characterization of, and experimentation with, the set of other possible transition systems for building graphs is a promising area for future research.

References

Arnborg, Stefan, Jens Lagergren, and Detlef Seese. 1991. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340.

Arnborg, Stefen, Derek G. Corneil, and Andrzej Proskurowski. 1987. Complexity of finding embeddings in a *k*-tree. *SIAM Journal of Algebraic and Discrete Methods*, 8:277–284.

Attardi, Giuseppe. 2006. Experiments with a multilanguage non-projective dependency parser. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pages 166–170, New York, NY.

Banarescu, Laura, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia.

Bodlaender, H. L. 1996. A linear time algorithm for finding tree decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317.

Bodlaender, Hans L. 2006, Treewidth: Characterizations, applications, and computations. In Fedor V. Fomin, editor, *Graph-Theoretic Concepts in Computer Science: 32nd International Workshop (WG 2006)*, Springer, Bergen, Norway, pages 1–14.

Charniak, E. 1993. *Statistical Language Learning*. MIT Press.

Choi, Jinho D. and Andrew McCallum. 2013. Transition-based dependency parsing with selectional branching. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL-13)*, pages 1052–1062, Sofia.

Copestake, Ann, Dan Flickinger, Carl Pollard, and Ivan A Sag. 2005. Minimal recursion semantics: An introduction. *Research on Language and Computation*, 3(2-3):281–332.

Courcelle, Bruno. 1990. The monadic second-order logic of graphs. I. recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75.

Covington, Michael A. 2001. A fundamental algorithm for dependency parsing. In *Proceedings of the 39th Annual ACM Southeast Conference*, pages 95–102, Athens, GA.

Damonte, Marco, Shay B. Cohen, and Giorgio Satta. 2017. An incremental parser for abstract meaning representation. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics (EACL)*, pages 536–546, Valencia.

Drewes, Frank, Hans-Jörg Kreowski, and Annegret Habel. 1997. Hyperedge replacement, graph grammars. In *Handbook of Graph Grammars*, volume 1. World Scientific, Singapore, pages 95–162.

Du, Yantao, Fan Zhang, Weiwei Sun, and Xiaojun Wan. 2014. Peking: Profiling syntactic tree parsing techniques for semantic graph parsing. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval-2014)*, pages 459–464, Dublin.

Flanigan, Jeffrey, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A. Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 1426–1436, Baltimore, MD.

Flickinger, Dan. 2000. On building a more effcient grammar by exploiting types. *Natural Language Engineering*, 6(1):15–28.

Flickinger, Dan, Yi Zhang, and Valia Kordoni. 2012. Deepbank: A dynamically annotated treebank of the Wall Street Journal. In *Proceedings of the 11th International Workshop on Treebanks and Linguistic Theories*, pages 85–96, Lisbon.

Goldberg, Yoav, Francesco Sartorio, and Giorgio Satta. 2014. A tabular method for dynamic oracles in transition-based parsing. *Transactions of the Association for Computational Linguistics*, 2:116–130.

Gómez-Rodríguez, Carlos and Joakim Nivre. 2013. Divisible transition systems and multiplanar dependency parsing. *Computational Linguistics*, 39:799–846.

Hajic, Jan, Eva Hajicová, Jarmila Panevová, and Petr Sgall. 2012. Announcing Prague Czech-English dependency treebank 2.0. In *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*, pages 3153–3160, Istanbul.

Henderson, James, Paola Merlo, Ivan Titov, and Gabriele Musillo. 2013. Multilingual joint parsing of syntactic and semantic dependencies with a latent variable model. *Computational Linguistics*, 39(4):949–998.

Huang, Liang and Kenji Sagae. 2010. Dynamic programming for linear-time incremental parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL-10)*, pages 1077–1086, Uppsala.

Jones, Bevan, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, and Kevin Knight. 2012. Semantics-based machine translation with hyperedge replacement grammars. In *Proceedings of the 24th International Conference on Computational Linguistics (COLING-12)*, pages 1359–1376, Mumbai.

Jones, Bevan K., Sharon Goldwater, and Mark Johnson. 2013. Modeling graph languages with grammars extracted via tree decompositions. In *Proceedings of the 11th International Conference on Finite-State Methods and Natural Language Processing (FSMNLP2013)*, pages 54–62, St. Andrews.

Joshi, A. K. and Y. Schabes. 1997. Tree-adjoining grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of*

Formal Languages, volume 3. Springer, Berlin, pages 69–124.

Kuhlmann, Marco, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 673–682, Portland, OR.

Kuhlmann, Marco and Stephan Oepen. 2016. Towards a catalogue of linguistic graph banks. *Computational Linguistics*, 42(4):819–827.

May, Jonathan. 2016. Semeval-2016 task 8: Meaning representation parsing. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1063–1073, San Diego, CA.

Miyao, Yusuke. 2006. *From Linguistic Theory to Syntactic Analysis: Corpus-Oriented Grammar Development and Feature Forest Model*. Ph.D. thesis, University of Tokyo, Tokyo, Japan.

Nivre, Joakim. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513–553.

Nivre, Joakim. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359, Singapore.

Oepen, Stephan, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinkova, Dan Flickinger, Jan Hajic, and Zdenka Uresova. 2015. Semeval 2015 task 18: Broad-coverage semantic dependency parsing. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 915–926, Denver, CO.

Palmer, Martha, Daniel Gildea, and Paul Kingsbury. 2005. The Proposition Bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1):71–106.

Peng, Xiaochang, Linfeng Song, and Daniel Gildea. 2015. A synchronous hyperedge replacement grammar based approach for AMR parsing. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning (CoNLL-15)*, pages 731–739, Beijing.

Pitler, Emily and Ryan McDonald. 2015. A linear-time transition system for crossing interval trees. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 662–671, Denver, CO.

Pourdamghani, Nima, Yang Gao, Ulf Hermjakob, and Kevin Knight. 2014. Aligning English strings with abstract meaning representation graphs. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 425–429, Doha.

Ribeyre, Corentin, Éric Villemonte de La Clergerie, and Djamé Seddah. 2015. Because syntax does matter: Improving predicate-argument structures parsing using syntactic features. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-15)*, pages 64–74, Denver, CO.

Sagae, Kenji and Jun'ichi Tsujii. 2008. Shift-reduce dependency DAG parsing. In *Proceedings of the 22nd International Conference on Computational Linguistics (COLING-08)*, pages 753–760, Manchester.

Titov, Ivan, James Henderson, Paola Merlo, and Gabriele Musillo. 2009. Online graph planarisation for synchronous parsing of semantic and syntactic dependencies. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 1562–1567, Pasadena, CA.

Wang, Chuan, Nianwen Xue, and Sameer Pradhan. 2015. A transition-based algorithm for AMR parsing. In *Proceedings of the 2015 Meeting of the North American Chapter of the Association for Computational Linguistics (NAACL-15)*, pages 366–375, Denver, CO.

Zhou, Junsheng, Feiyu Xu, Hans Uszkoreit, Weiguang Qu, Ran Li, and Yanhui Gu. 2016. AMR parsing with an incremental joint model. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 680–689, Austin, TX.