Synchronous Context-Free Grammars and Optimal Parsing Strategies

Daniel Gildea[*]
University of Rochester

Giorgio Satta[**]
Università di Padova

The complexity of parsing with synchronous context-free grammars is polynomial in the sentence length for a fixed grammar, but the degree of the polynomial depends on the grammar. Specifically, the degree depends on the length of rules, the permutations represented by the rules, and the parsing strategy adopted to decompose the recognition of a rule into smaller steps. We address the problem of finding the best parsing strategy for a rule, in terms of space and time complexity. We show that it is NP-hard to find the binary strategy with the lowest space complexity. We also show that any algorithm for finding the strategy with the lowest time complexity would imply improved approximation algorithms for finding the treewidth of general graphs.

1. Introduction

Synchronous context-free grammars (SCFGs) generalize context-free grammars (CFGs) to generate two strings simultaneously. The formalism dates from the early days of automata theory; it was developed under the name syntax-direct translation schemata to model compilers for programming languages (Lewis and Stearns 1968; Aho and Ullman 1969). SCFGs are widely used today to model the patterns of re-ordering between natural languages, and they form the basis of many state-of-the-art statistical machine translation systems (Chiang 2007).

Despite the fact that SCFGs are a very natural extension of CFGs, and that the parsing problem for CFGs is rather well understood nowadays, our knowledge of the parsing problem for SCFGs is quite limited, with many questions still left unanswered. In this article we tackle one of these open problems.

Unlike CFGs, SCFGs do not admit any canonical form in which rules are bounded in length (Aho and Ullman 1972), as for instance in the well-known Chomsky normal form for CFGs. A consequence of this fact is that the computational complexity of parsing with SCFG depends on the grammar. More precisely, for a fixed SCFG, parsing is polynomial in the length of the string. The degree of the polynomial depends on the

---

[*] 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.

length of the grammar's rules, the re-ordering patterns represented by these rules, as well as the strategy used to parse each rule. The complexity of finding the best parsing strategy for a fixed SCFG rule has remained open. This article shows that it is NP-hard to find the binary parsing strategy having the lowest space complexity. We also consider the problem of finding the parsing strategy with the lowest time complexity, and we show that it would require progress on long-standing open problems in graph theory either to find a polynomial algorithm or to show NP-hardness.

The parsing complexity of SCFG rules increases with the increase of the number of nonterminals in the rule itself. Practical machine translation systems usually confine themselves to binary rules, that is, rules having no more than two right-hand side nonterminals, because of the complexity issues and because binary rules seem to be adequate empirically (Huang et al. 2009). Longer rules are of theoretical interest because of the naturalness and generality of the SCFG formalism. Longer rules may also be of practical interest as machine translation systems improve.

For a fixed SCFG, complexity can be reduced by **factoring** the parsing of a grammar rule into a sequence of smaller steps, which we refer to as a **parsing strategy**. Each step of a parsing strategy collects nonterminals from the right-hand side of an SCFG rule into a subset, indicating that a portion of the SCFG rule has been matched to a subsequence of the two input strings, as we explain precisely in Section 2. A parsing strategy is binary if it combines two subsets of nonterminals at each step to create a new subset. We consider the two problems of finding the binary parsing strategy with optimal time and space complexity. For time complexity, there is no benefit to considering parsing strategies that combine more than two subsets at a time. For space complexity, the lowest complexity can be achieved with no factorization whatsoever; however, considering binary parsing strategies can provide an important trade-off between time and space complexity. Our results generalize those of Crescenzi et al. (2015), who show NP-completeness for decision problems related to both time and space complexity for **linear** parsing strategies, which are defined to be strategies that add one nonterminal at a time.

Our approach constructs a graph from the permutation of nonterminals in a given SCFG rule, and relates the parsing problem to properties of the graph. Our results for space complexity are based on the graph theoretic concept of **carving width**, whose decision problem is NP-complete for general graphs. Section 3 relates the space complexity of the SCFG parsing problem to the carving width of a graph constructed from the SCFG rule. In Section 4, we show that any polynomial time algorithm for optimizing the space complexity of binary SCFG parsing strategies would imply a polynomial time algorithm for carving width of general graphs. Our results for time complexity are based on the graph theoretic concept of **treewidth**. In Section 5, we show that any polynomial time algorithm for the decision problem of the time complexity of binary SCFG strategies would imply a polynomial time constant–factor approximation algorithm for the treewidth of general graphs. In the other direction, NP-completeness of the decision problem of the time complexity for SCFG would imply the NP-completeness of treewidth of general graphs of degree six. These are both long-standing open problems in graph theory.

2. Synchronous Context-Free Grammars and Parsing Strategies

In this section we informally introduce the notion of synchronous context-free grammar and define the computational problem that we investigate in this article. We assume the reader to be familiar with the notion of context-free grammar, and only briefly

summarize the adopted notation. Throughout this article, for any positive integer n we write $[n]$ to denote the set $\{1, \ldots, n\}$.

2.1 Synchronous Context-Free Grammars

As usual, a CFG has a finite set of rules having the general form $A \to \alpha$, where A is a nonterminal symbol to be rewritten and α is a string of nonterminal and terminal symbols. A **synchronous context-free grammar** is a rewriting system based on a finite set of so-called synchronous rules. Each synchronous rule has the general form $[A_1 \to \alpha_1, \; A_2 \to \alpha_2]$, where $A_1 \to \alpha_1$ and $A_2 \to \alpha_2$ are CFG rules. By convention, we refer to $A_1 \to \alpha_1$ and $A_2 \to \alpha_2$ as the Chinese and English components of the synchronous rule, respectively. Furthermore, α_1, α_2 must be synchronous strings. This means that there exists a bijection between the occurrences of nonterminals in α_1 and the occurrences of nonterminals in α_2, and that this bijection is explicitly provided by the synchronous rule. Nonterminal occurrences that correspond under the given bijection are called **linked** nonterminal occurrences, or just linked nonterminals when no ambiguity arises. We assume that linked nonterminals always have the same label.

As a simple example, consider the synchronous rule

$$[A \to aA^{\boxed{1}}bB^{\boxed{2}}d, \; A \to bB^{\boxed{2}}cA^{\boxed{1}}]$$

where A, B are nonterminal symbols and a, b, c, d are terminal symbols. We have indicated the bijection associated with the synchronous rule by annotating the nonterminal occurrences with natural numbers, with the intended meaning that linked nonterminals get the same number. We refer to this number as a nonterminal's **index**.

The bijection associated with a synchronous rule plays a major role in the derivation of a pair of strings by the SCFG. In fact, in an SCFG we can only apply a synchronous rule to linked nonterminals. To illustrate this, we use our running example and consider the synchronous strings $[A^{\boxed{1}}, \; A^{\boxed{1}}]$. We can construct a derivation by applying our synchronous rule to the linked nonterminals, written

$$[A^{\boxed{1}}, \; A^{\boxed{1}}] \Rightarrow [aA^{\boxed{2}}bB^{\boxed{3}}d, \; bB^{\boxed{3}}cA^{\boxed{2}}]$$

Note that we have renamed the indices in the synchronous rule to make them disjoint from the indices already in use in the synchronous string to be rewritten. Although this is unnecessary in this first derivation step, this strategy will always be adopted to avoid conflicts in more complex derivations. We can move on with our derivation by applying once more our synchronous rule to rewrite the linked A nonterminals, obtaining

$$[A^{\boxed{1}}, \; A^{\boxed{1}}] \Rightarrow [aA^{\boxed{2}}bB^{\boxed{3}}d, \; bB^{\boxed{3}}cA^{\boxed{2}}] \Rightarrow [aaA^{\boxed{4}}bB^{\boxed{5}}dbB^{\boxed{3}}d, \; bB^{\boxed{3}}cbB^{\boxed{5}}cA^{\boxed{4}}]$$

In this derivation, the renaming of the indices is crucial to avoid conflicts.

Let S be a special nonterminal of our SCFG, which we call the starting nonterminal. Using our notion of derivation, one can start from $[S^{\boxed{1}}, \; S^{\boxed{1}}]$ and attempt to derive a pair of strings $[u, v]$ entirely composed of terminal symbols. Whenever this is possible, we say that $[u, v]$ is in the translation generated by the SCFG, meaning that v is one of the possible translations of u.

Example 1
Consider the following list of synchronous rules, where symbols s_i are rule labels to be used later as references

$$s_1 : [S \to A^{\boxed{1}} \ B^{\boxed{2}}, \ S \to B^{\boxed{2}} \ A^{\boxed{1}}]$$
$$s_2 : [A \to aA^{\boxed{1}}b, \ A \to bA^{\boxed{1}}a]$$
$$s_3 : [A \to ab, \ A \to ba] \ ,$$
$$s_4 : [B \to cB^{\boxed{1}}d, \ B \to dB^{\boxed{1}}c]$$
$$s_5 : [B \to cd, \ B \to dc]$$

What follows is a valid derivation of the SCFG, obtained by rewriting at each step the linked nonterminals with the Chinese component occurring at the left-most position in the sentential form

$$[S^{\boxed{1}}, \ S^{\boxed{1}}] \Rightarrow^{s_1} [A^{\boxed{2}}B^{\boxed{3}}, B^{\boxed{3}}A^{\boxed{2}}]$$
$$\Rightarrow^{s_2} [aA^{\boxed{4}}bB^{\boxed{3}}, B^{\boxed{3}}bA^{\boxed{4}}a]$$
$$\Rightarrow^{s_2} [aaA^{\boxed{5}}bbB^{\boxed{3}}, B^{\boxed{3}}bbA^{\boxed{5}}aa]$$
$$\Rightarrow^{s_3} [aaabbbB^{\boxed{3}}, B^{\boxed{3}}bbbaaa]$$
$$\Rightarrow^{s_4} [aaabbbcB^{\boxed{6}}d, dB^{\boxed{6}}cbbbaaa]$$
$$\Rightarrow^{s_5} [aaabbbccdd, ddccbbbaaa]$$

It is not difficult to see that the given SCFG provides the translation $\{[a^pb^pc^qd^q, d^qc^qb^pa^p] \mid p, q \geq 1\}$.

One can also associate SCFG derivations with parse trees, in a way very similar to what we do with CFG derivations. More precisely, an SCFG derivation is represented as a pair of parse trees, each generating one component in the derived string pair. Furthermore, linked nonterminals in the SCFG derivation are annotated in the parse trees to indicate the place of application of some synchronous rule.

Example 2
The SCFG derivation for string pair [*aaabbbccdd*, *ddccbbbaaa*] in Example (1) is associated with the following tree pair:
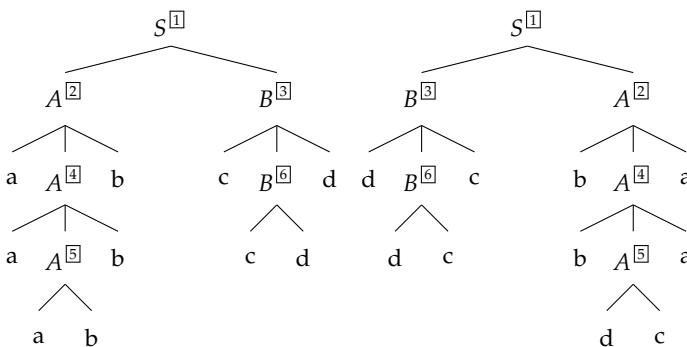
One common view of SCFGs is that each synchronous rule implements a permutation of the (occurrences of the) nonterminal symbols in the two rule components. More precisely, the nonterminals in the right-hand side of the Chinese rule component are re-used in the right-hand side of the English component, but with a different ordering, as defined by the bijection associated with the rule. This reordering is at the core of the translation schema implemented by the SCFG. We will often use the permutation view of a synchronous rule in later sections.

As already mentioned, we assume that linked nonterminals in a synchronous rule have the same label. In natural language processing applications, this is by far the most common assumption, but one might also drop this requirement. The results presented in this article do not rest on such a restriction.

2.2 Parsing Strategies

The recognition and the parsing problems for SCFGs are natural generalizations of the same problems defined for CFGs. Given an SCFG of interest, in the recognition problem one has to decide whether an input pair $[u, v]$ can be generated. In addition, for the parsing problem one has to construct a parse forest, in some compact representation, with all tree pairs that the SCFG produces when generating $[u, v]$. When dynamic programming techniques are used, it turns out that recognition and parsing algorithms are closely related, since the elementary steps that are performed during recognition can be memoized and later used to construct the desired parse forest.

Despite the similarity between CFGs and SCFGs, it turns out that there is a considerable gap in the space and time complexity for recognition and parsing based on these two classes. More specifically, for a fixed CFG, we can solve the recognition problem in time cubic in the length of the input string, using standard dynamic programming techniques. In contrast, for a fixed SCFG we can still recognize the input pair $[u, v]$ in polynomial time, but the degree of the polynomial depends on the specific structure of the synchronous rules of the grammar, and can be much larger than in the CFG case. A similar scenario also holds for the space complexity. The reason for this gap is informally explained in what follows.

In the investigation of the recognition problem for CFGs and SCFGs, a crucial notion is the one of parsing strategy. Recognition algorithms can be seen as sequences of elementary computational steps in which two parse trees that generate non-overlapping portions of the input are combined into a larger parse tree, according to the rules of the grammar. We call parsing strategy any general prescription indicating the exact order in which these elementary steps should be carried out. Generally speaking, parsing strategies represent the order in which each parsing tree of the input is constructed.

For the CFG case, let us consider the standard top–down chart parsing algorithm of Kay (1980). The algorithm adopts a parsing strategy that visits the nonterminals in the right-hand side of a rule one at a time and in a left-to-right order, combining the associated parse trees accordingly. Parse trees are then compacted into so-called chart edges, which record the two endpoints of the generated substring along with other grammar information. Chart parsing can be generalized to SCFGs. However, in the latter case parse trees no longer generate single substrings: They rather generate tuples with several substrings. This can be ascribed to the added component in synchronous rules and to the re-ordering of the nonterminals across the two components.

Let us call fan-out the number of substrings generated by a parse tree (this notion will be formally defined later). It is well-known among parsing practitioners that the

fan-out affects the number of stored edges for a given input string, and is directly connected to the space and time performance of the algorithm. A binary parsing strategy of fan-out φ has space complexity $\mathcal{O}(n^{2\varphi})$ and time complexity at most $\mathcal{O}(n^{3\varphi})$ where n is the sentence length (Seki et al. 1991; Gildea 2010). If we adopt the appropriate parsing strategy, we can reduce the fan-out, resulting in asymptotic improvement in the space and time complexity of our algorithm. To illustrate this claim we discuss a simple example.

Example 3
Consider the synchronous rule

$$s : [A \rightarrow B^{\boxed{1}} B^{\boxed{2}} B^{\boxed{3}} B^{\boxed{4}} B^{\boxed{5}} B^{\boxed{6}} B^{\boxed{7}} B^{\boxed{8}}$$
$$A \rightarrow B^{\boxed{5}} B^{\boxed{7}} B^{\boxed{3}} B^{\boxed{1}} B^{\boxed{8}} B^{\boxed{6}} B^{\boxed{2}} B^{\boxed{4}}] \tag{1}$$

The permutation associated with this rule is schematically visualized in Figure 1.

A naive parsing strategy for rule s would be to collect the nonterminals $B^{\boxed{k}}$ one at a time and in ascending order of k. For instance, at the first step we combine $B^{\boxed{1}}$ and $B^{\boxed{2}}$, constructing a parse tree with fan-out 3, as seen from Figure 1. The worst case is attested when we construct the parse tree consisting of occurrences $B^{\boxed{1}}, \ldots, B^{\boxed{5}}$, which has a fan-out of 4.

Alternatively, we could explore more flexible parsing strategies. An example is depicted in Figure 2, where each elementary step is represented as an internal node of a tree. This time, at the first step (node n_1) we combine $B^{\boxed{3}}$ and $B^{\boxed{4}}$, as depicted in Figure 3a. At the second step (node n_3), we combine the result of node n_1 and $B^{\boxed{2}}$, again constructing a parse tree with fan-out three, as depicted in Figure 3b, and so on. This strategy is non-linear, because it might combine parses containing more than one nonterminal occurrence each; see Figure 3c. From Figure 1 it is not difficult to check that at each node n_k the constructed parse has fan-out bounded by 3. We therefore conclude that our second strategy is more efficient than the left-to-right one.
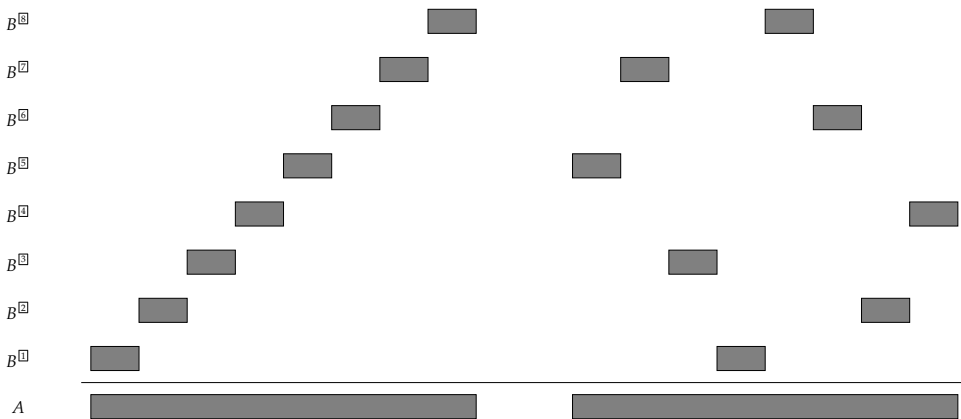

Figure 1
Graphical representation of the permutation associated with synchronous rule s of Equation (1).
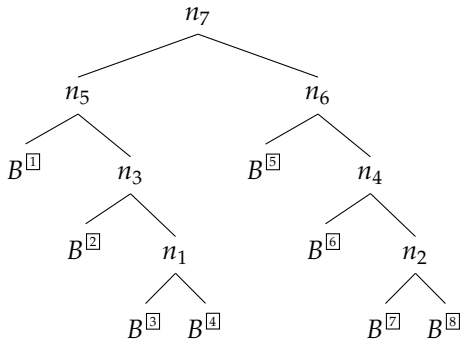
Figure 2
A bidirectional parsing strategy for rule s of Equation (1).

This example shows that the maximum fan-out needed to parse a synchronous rule depends on the adopted parsing strategy. In order to optimize the computational resources of a parsing algorithm, we need to search for a parsing strategy that minimizes the maximum fan-out of the intermediate parse trees. The problem that we investigate in this article is therefore the one of finding optimal parsing strategies for synchronous rules, where in this context optimal means with a value of the maximal fan-out as small as possible.

2.3 Fan-out and Parsing Optimization

In this section we provide a mathematical definition for the concepts that we have informally introduced in the previous section. We need to do so in order to be able to precisely define the computational problem that is investigated in this article.
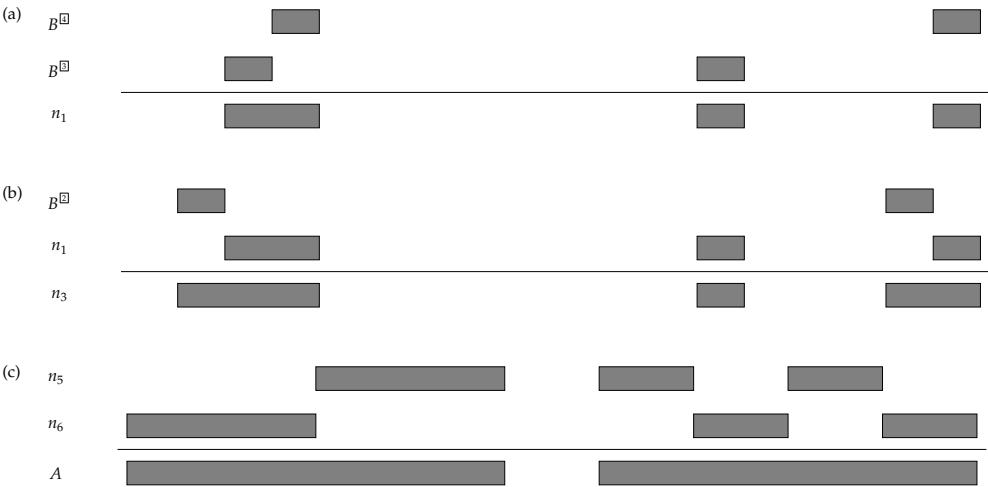

Figure 3
First step (a), second step (b), and last step (c) of the parsing strategy in Figure 2.

Assume a synchronous context-free rule s with $r \geq 2$ linked nonterminals. We need to address occurrences of nonterminal symbols within s. We write $\langle 1, i \rangle$ to represent the ith occurrence (from left to right) in the right-hand side of the Chinese component of s. Similarly, $\langle 2, i \rangle$ represents the ith occurrence in the right-hand side of the English component of s. Each pair $\langle h, i \rangle$, $h \in [2]$ and $i \in [r]$, is called an **occurrence**.

We assume without loss of generality that the nonterminals of the Chinese component are indexed sequentially. That is, the index of $\langle 1, i \rangle$ is i. Let π be the permutation over set $[r]$ implemented by s, meaning that $\langle 2, i \rangle$ is annotated with index $\pi(i)$, and therefore is co-indexed with $\langle 1, \pi(i) \rangle$. As an example, in rule (1) we have $r = 8$ and $\pi(1) = 5$, $\pi(2) = 7$, $\pi(3) = 3$, etc. Under this convention, each pair $(\langle 1, \pi(i) \rangle, \langle 2, i \rangle)$ and, equivalently, each pair $(\langle 1, i \rangle, \langle 2, \pi^{-1}(i) \rangle)$, $i \in [r]$ is called a **linked pair**.

A **parsing strategy** for s is a rooted, binary tree τ_s with r leaves, where each leaf is a linked pair $(\langle 1, \pi(i) \rangle, \langle 2, i \rangle)$. As already explained, the intended meaning is that each internal node of τ_s represents the operation of combining the linked pairs below the left node with the linked pairs below the right node. These operations must be performed in a bottom–up fashion.

Let n be an internal node of τ_s, and let τ_n be the subtree of τ_s rooted at n. We write $y(n)$ to denote the set of all occurrences $\langle h, i \rangle$ appearing in the linked pair of some leaf of τ_n. We say that occurrence $\langle h, i \rangle \in y(n)$ is a **right boundary** of n if $i = r$ or if $i < r$ and $\langle h, i+1 \rangle \notin y(n)$. Symmetrically, we say that $\langle h, i \rangle \in y(n)$ is a **left boundary** if $i = 1$ or if $i > 1$ and $\langle h, i-1 \rangle \notin y(n)$. Note that the occurrences $\langle 1, 1 \rangle$ and $\langle 2, 1 \rangle$ are always left boundaries, and $\langle 1, r \rangle$ and $\langle 2, r \rangle$ are always right boundaries. We let $\mathbf{bd}(n)$ be the total number of right and left boundaries in $y(n)$.

Intuitively, the number of boundaries in $y(n)$ provides the number of endpoints of the substrings in the rule components of s that are spanned by the occurrences in $y(n)$. Dividing this total number by two provides the number of substrings spanned by the occurrences in $y(n)$. We therefore define the **fan-out** at node n as

$$\mathbf{fo}(n) = \frac{1}{2}\mathbf{bd}(n) \tag{2}$$

As discussed in Section 2.2, the largest fan-out over all internal nodes of a parsing strategy provides space and time bounds for a dynamic programming parsing algorithm adopting that strategy. Given an input synchronous rule s, we wish to find the parsing strategy that minimizes quantity

$$\min_{\tau} \max_{n} \mathbf{fo}(n) \tag{3}$$

where τ ranges over all possible parsing strategies for s, and n ranges over all possible nodes of τ. One of the two main results in this article is that this optimization problem is NP-hard.

3. Cyclic Permutation Multigraphs and Carving Width

In this section, we relate the fan-out of parsing strategies for an SCFG rule to the properties of a specific graph derived from the rule's permutation.

We begin by introducing some basic terminology. A multiset is a set where there can be several occurrences of a given element. We use $\uplus$ as the merge operation defined for multisets: This operation preserves the number of occurrences of the merged multisets. As usual, we denote an undirected graph as a pair $G = (V, E)$ where V is a finite

set of vertices and E is a set of edges, with each edge consisting of an unordered pair of vertices. A multigraph is a graph that uses a multiset of edges. This means that a multigraph can have several occurrences of an edge impinging on a pair of vertices.

A **cyclic permutation multigraph** is a multigraph $G = (V, A \uplus B)$ such that both $P_A = (V, A)$ and $P_B = (V, B)$ are Hamiltonian cycles, that is, cycles that visit all the vertices in V exactly once. In the following, the edges in A will be called **red** and the edges in B will be called **green**. Our definition is based on the permutation multigraphs of Crescenzi et al. (2015), which differ in that they consist of two acyclic Hamiltonian paths.

In this article, we use cyclic permutation multigraphs to encode synchronous rules. Let s be a synchronous rule with $r \geq 2$ linked pairs. Let also $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$ be a special linked pair representing the left-hand side nonterminal of s. We construct the cyclic permutation multigraph M_s, representing s, as follows. The linked pairs $(\langle 1, \pi(i) \rangle, \langle 2, i \rangle)$, $i \in [r]$, along with $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$, form the vertices of M_s. The red cycle of M_s begins with $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$, then follows the order in which nonterminals occur in the Chinese component of s, and finally returns to $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$. Similarly, the green cycle of M_s begins and ends with $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$, and follows the order in which nonterminals occur in the English component.

In the other direction, given any cyclic permutation multigraph, we can derive a corresponding SCFG rule by, first, choosing an arbitrary vertex as representing the left-hand side nonterminal, and then following the red cycle to obtain the sequence of Chinese nonterminals, and finally following the green cycle to obtain the sequence of English nonterminals.

Example 4
Consider the SCFG rule s of Equation (1) in Example (3). Figure 4 shows the cyclic permutation multigraph associated with s. The red path starts at $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$, followed by the vertices in the order of the Chinese nonterminals of the rule, that is, $(\langle 1, 1 \rangle, \langle 2, 4 \rangle)$, $(\langle 1, 2 \rangle, \langle 2, 7 \rangle)$, $(\langle 1, 3 \rangle, \langle 2, 3 \rangle)$, ..., $(\langle 1, 8 \rangle, \langle 2, 5 \rangle)$. The green path starts at $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$, followed by the vertices in the order of the English nonterminals, that is, $(\langle 1, 5 \rangle, \langle 2, 1 \rangle)$, $(\langle 1, 7 \rangle, \langle 2, 2 \rangle)$, $(\langle 1, 3 \rangle, \langle 2, 3 \rangle)$, ..., $(\langle 1, 4 \rangle, \langle 2, 8 \rangle)$.

An important property of M_s is that every edge connects two linked pairs that share a boundary in either the Chinese component of the rule (red edge) or in the English component (green edge). Note that, as a special case, we consider the linked pair $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$ as sharing two left boundaries with the linked pairs $(\langle 1, 1 \rangle, \langle 2, \pi^{-1}(1) \rangle)$
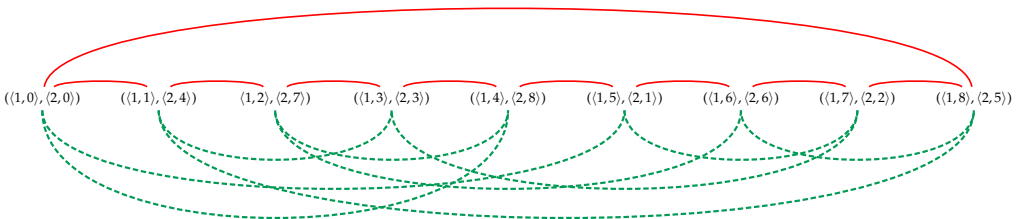

Figure 4
The cyclic permutation multigraph corresponding to the SCFG rule s of Equation (1). In this figure and all subsequent figures, green edges are shown with dashed lines.

and $(\langle 1, \pi(1)\rangle, \langle 2, 1\rangle)$, and as sharing two right boundaries with the linked pairs $(\langle 1, r\rangle, \langle 2, \pi^{-1}(r)\rangle)$ and $(\langle 1, \pi(r)\rangle, \langle 2, r\rangle)$.

Consider any set S of linked pairs from M_s such that $(\langle 1, 0\rangle, \langle 2, 0\rangle) \notin S$. Let $\overline{S}$ be the complement set of S, that is, $\overline{S}$ is the set of linked pairs from M_s not in S. It is not difficult to see that the multiset of edges connecting S and $\overline{S}$ corresponds to the set of boundaries of any parse tree associated with the linked pairs in S. We can apply this property to parsing strategies, which we have previously defined as rooted binary trees, in order to count the boundaries that are open after completing some parsing step represented by an internal node n, which we have defined as $\mathbf{bd}(n)$. Consider for instance the parsing strategy shown in Figure 2, and consider the internal node n_3. At this node we have collected nonterminals $B^{[2]}$, $B^{[3]}$, and $B^{[4]}$, and we have $\mathbf{bd}(n_3) = 6$. If we consider the set S with the corresponding linked pairs $(\langle 1, 2\rangle, \langle 2, 7\rangle)$, $(\langle 1, 3\rangle, \langle 2, 3\rangle)$, and $(\langle 1, 4\rangle, \langle 2, 8\rangle)$, we can see that the number of edges of the multigraph that connect S with $\overline{S}$ is six, that is, exactly the value of $\mathbf{bd}(n_3)$. In order to express these observations in a mathematical way, we need to introduce the notions of tree layout, width, and carving width, which we borrow from graph theory.

A **tree layout** T of a graph G is an undirected binary branching tree having one vertex of G at each leaf. To avoid confusion, in what follows we use the term **node** when referring to vertices of layout trees and we use the term **arc** when referring to edges of layout trees. Edges of G are routed along the arcs of T. A simple example is provided in Figure 5, showing a graph G and a tree layout T of G.

Informally, the width of an arc of T is the number of edges from G that are routed through that arc. We define this notion more precisely in what follows. We start with some auxiliary notation. Let V be the set of vertices of G and let S be any subset of V. The **edge boundary** of S in V, written $\partial_V(S)$, is the set of edges of G that connect vertices in S and vertices in the complement set $\overline{S} = V \setminus S$. Let a be an arc in a tree layout T of G, and let T_1 and T_2 be the two components of the graph obtained by removing a from T. Let also $L(T_1)$ be the subset of V appearing at the leaves of T_1. The **width** of a in T is the number of edges in G that cross between T_1 and T_2:

$$\mathbf{wd}_{G,T}(a) = |\partial_V(L(T_1))| \qquad (4)$$

The maximum width among all of the arcs of T is the **carving width** of T. The carving width of T in Figure 5 is 3. The carving width of G is the minimum carving width over all possible tree layouts of G:

$$\mathbf{wd}(G) = \min_T \max_a \mathbf{wd}_{G,T}(a) \qquad (5)$$
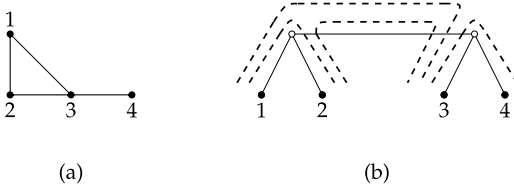

(a)   (b)

Figure 5
(a) Graph G. (b) Tree layout T of G; G's edges are routed along T's arcs and are represented as dashed lines.

Deciding whether the carving width of arbitrary graphs is less than or equal to a given integer is an NP-complete problem (Seymour and Thomas 1994). Throughout this article, we extend to multigraphs all of the discussed notions of tree layout, width, and carving width, in the obvious way. Because graphs are a subset of multigraphs, carving width of multigraphs is also an NP-complete problem.

We can now apply the notion of tree layout to parsing strategies for SCFG rules, and show an important property that relates the notions of fan-out (equivalently, boundary count) and carving width. Let s be an SCFG rule with r linked pairs. Recall that a parsing strategy for s is a rooted binary tree where each leaf node is a linked pair representing some nonterminal from the right-hand side of s. We attach to the root of our parsing strategy an additional leaf node, representing the special linked pair $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$. The resulting tree has $r + 1$ leaves, and can therefore be used as a tree layout for the cyclic permutation multigraph M_s encoding s. Furthermore, each internal node of the tree layout is associated with a parsing step.

Example 5

Consider the SCFG rule s in Equation (1) and the associated cyclic permutation multigraph M_s shown in Figure 4. Consider also the parsing strategy τ_s for s shown in Figure 2. By attaching the linked pair $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$ to the root node of τ_s, we can derive a tree layout T for M_s, which is shown in Figure 6. Observe that every linked pair (vertex) of M_s, including the special linked pair $(\langle 1, 0 \rangle, \langle 2, 0 \rangle)$, is placed at some leaf node of T, and the edges of M_s are routed along arcs of T. Let a be an arc of T and let n be the node below a, with respect to the root node. Observe how edges that are routed along a correspond to boundaries that are open at the associated step n of the parsing strategy τ_s.
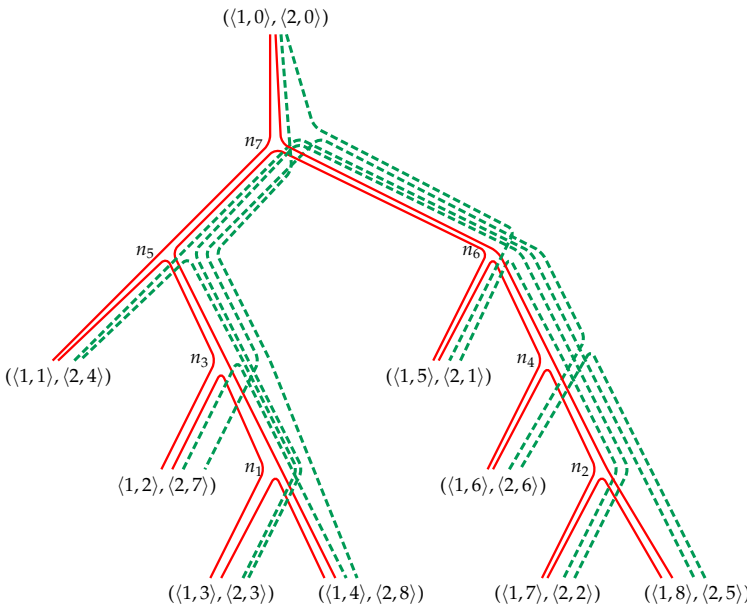

Figure 6
A tree layout of the cyclic permutation multigraph of Figure 4 obtained from the parsing strategy in Figure 2.

The next lemma summarizes the connection between carving width and fan-out.

Lemma 1
The minimum fan-out of any parsing strategy of an SCFG rule s is half of the carving width of the cyclic permutation multigraph constructed from s.

Proof
Let M_s be the cyclic permutation multigraph derived from s. Consider a parsing strategy τ_s for s. We construct a tree layout T_{M_s, τ_s} of M_s from the strategy τ_s by attaching a new leaf node for the left-hand side nonterminal to the root of τ_s. Each vertex of the cyclic permutation multigraph is placed at the corresponding leaf of the tree layout. When the two nonterminals sharing a given boundary are combined, the edge representing the boundary is routed along the two arcs below the node for this combination. The edges representing the left-most and right-most boundaries of the rule's Chinese and English components are routed through the node for the parsing strategy's root, because they must continue through the root to reach the leaf representing the left-hand side nonterminal. At each arc a of T_{M_s, τ_s}, the right-hand side of Equation (4) corresponds to the right-hand side of Equation (2), and thus half the carving width of T_{M_s, τ_s} is equal to the fan-out of τ_s.

In the other direction, let T_{M_s} be a tree layout of M_s. A parsing strategy τ_s can be constructed from T_{M_s} by removing the leaf node of T_{M_s} corresponding to the left-hand side of s, and choosing the adjacent node as the parsing strategy's root. As before, the fan-out of τ_s is half the carving width of T_{M_s}. The minimum fan-out over all strategies is also the minimum carving width over all tree layouts. ■

4. Carving Width of Cyclic Permutation Multigraphs

In this section, we reduce the problem of carving width of general graphs to the problem of carving width of cyclic permutation multigraphs, defined in Section 3. This reduction involves constructing a cyclic permutation multigraph from an input graph as outlined in Section 4.1. The details of the construction are given in Section 4.2, and the proof of NP-completeness for carving width of the resulting multigraphs is given in Section 4.3. This result is then used in Section 4.4 to show that finding a space-optimal binary parsing strategy for an SCFG rule is an NP-hard problem.

4.1 General Idea

Suppose that we are given an instance of the general carving width problem, consisting of a graph G and a positive integer k, where we have to decide whether the carving width of G is less than or equal to k. We identify the vertices of G with positive integers in $[n]$, where $n \geq 2$ is the number of vertices of G. Our reduction consists in the construction of a cyclic permutation multigraph M such that the carving width of M is equal to $4k$ if and only if the carving width of G is equal to k.

The main idea underlying our construction is that each vertex i of G is associated with a gadget in M consisting of two components. The first component is a grid-like graph X_i of $2d_i$ rows and $2d_i$ columns, where d_i is the degree of vertex i in G. The second component is a grid-like graph G_i of $4k$ rows and $4k$ columns. Besides the edges of the grids X_i and G_i, M also includes some extra edges, called **interconnection edges**. For each $i \in [n]$, there are some interconnection edges connecting G_i and X_i. Furthermore,
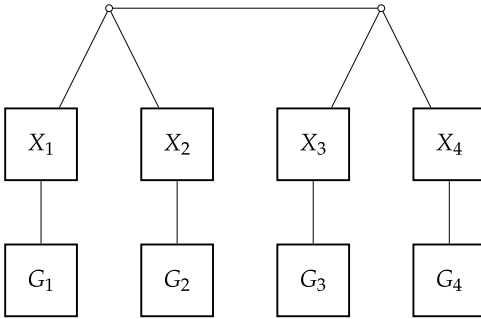
Figure 7
A tree layout T_M of cyclic permutation multigraph M (shown later in Figure 11) derived from the tree layout T of Figure 5.

for each edge (i, j) of G, there are interconnection edges connecting components X_i and X_j.

Interconnection edges serve two main purposes. First, they are used to connect the red and green paths within M, so that M satisfies the definition of cyclic permutation multigraph. Second, the connections between component pairs X_i and X_j mimic the structure of the source graph G, as will be explained in more detail later. This second condition guarantees that an optimal tree layout T_M of M can be obtained from an optimal tree layout T of G, by placing each X_i and G_i component under the leaf node of T that is associated with vertex i of G. A simple example of the construction is schematically depicted in Figure 7.

4.2 Construction

In this section, we define precisely the grid-like graphs X_i and G_i, for each vertex i of the source graph G, and outline the overall structure of the cyclic permutation multigraph M constructed from G.

4.2.1 Depth-First Traversal. To be used later in our construction, we need to define a special ordering of the edges of G. We do this here by constructing a path through G that starts at an arbitrary vertex and visits each edge exactly twice. We adapt the standard procedure for depth-first traversal of an undirected graph. More precisely, whenever we reach a vertex i, we continue our path by arbitrarily choosing an edge

---

Algorithm 1 Procedure for depth-first traversal of G starting at i

```
1: procedure DFS(i)
2:     append i to path
3:     if each edge (i, j) has already been visited then
4:         return
5:     for each edge (i, j) not already visited (in arbitrary order) do
6:         DFS(j)
7:         append i to path
8:     return
```
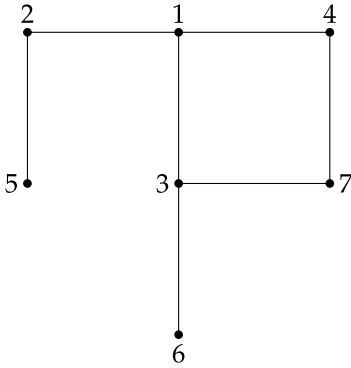
Figure 8
The path constructed by Algorithm 1 starting at vertex 1 of the displayed graph is
$\langle 1, 2, 5, 2, 1, 3, 6, 3, 7, 4, 1, 4, 7, 3, 1 \rangle$.

(i, j) that has not already been visited, and move to j. If all of the edges at i have already been visited, we backtrack to previous vertices of our path, until we reach edges that have not already been visited. The path construction stops when we reach the starting vertex and all of the edges of G have been visited. Algorithm 1 provides a recursive procedure for the incremental construction of the path, assuming that the path is a global variable initialized as the empty sequence of vertices. A simple example is also shown in Figure 8.

Let m be the number of edges of G. It is not difficult to show that the path produced by Algorithm 1 is a sequence $\langle i_1, \ldots, i_{2m+1} \rangle$ of vertices from G satisfying both of the following properties.

- For each edge (i, j) of G, there is a unique $k \in [2m]$ such that $i_k = i$ and $i_{k+1} = j$, and there is a unique $h \in [2m]$ such that $i_h = j$ and $i_{h+1} = i$.

- For each $j \in [n]$ we have $|\{k : k \leq 2m, i_k = j\}| = d_j$. (Recall that d_j is the degree of vertex j in G.)

The first bullet says that each edge of G is traversed exactly twice, the first time in one direction and the second time in the opposite direction. The second bullet is a direct consequence of the first bullet. These properties will be used in the specification of cyclic permutation multigraph M.

4.2.2 Component X_i. The graph component X_i, $i \in [n]$, embeds a square grid with $2d_i$ rows and $2d_i$ columns. In addition to the edges of the grid, X_i also contains a set of interconnection edges that connect X_i to component G_i (specified later) and that connect X_i to all components X_j such that (i, j) is an edge of G. This is schematically depicted in Figure 9. (A complete example for components X_i and G_i will be presented later.)

More precisely, let $x_{m,n}^{(i)}$ be the vertex in the mth row and nth column of X_i. Similarly, let $g_{m,n}^{(i)}$ be the vertex in the mth row and nth column of G_i.
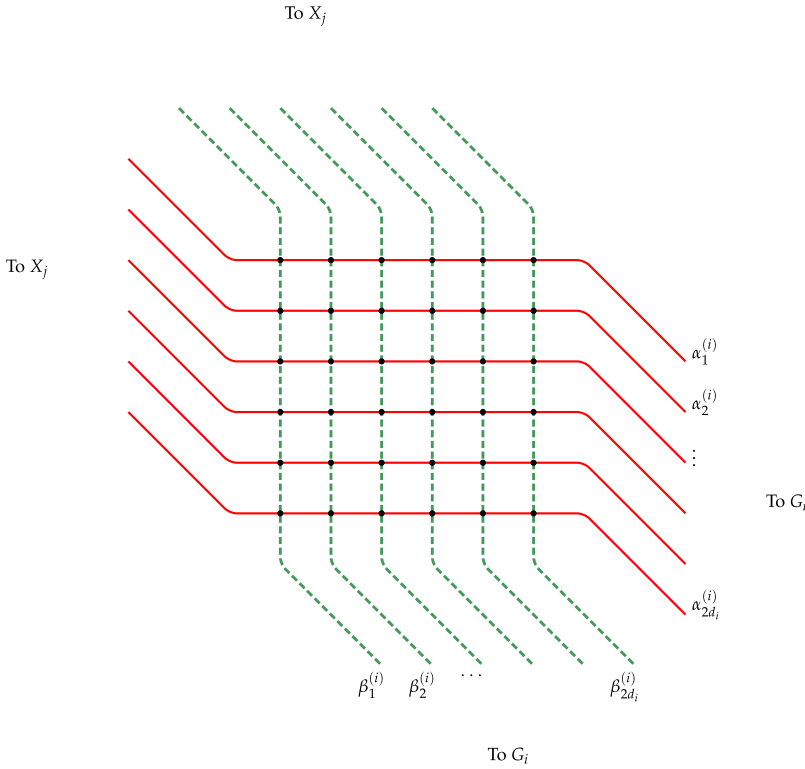
To X_j

Figure 9
Grid X_i corresponding to vertex i of G with $d_i = 3$. Interconnection edges from the left column and top row reach grids X_j for vertices j that are neighbors of i in G.

- For each $p \in [2d_i - 1]$, X_i has an interconnection edge $\alpha_p^{(i)} = (x_{p,2d_i}^{(i)}, g_{p,1}^{(i)})$. Furthermore, X_i has an interconnection edge $\alpha_{2d_i}^{(i)} = (x_{2d_i,2d_i}^{(i)}, g_{2k,1}^{(i)})$.

- Symmetrically, for each $q \in [2d_i - 1]$, X_i has an interconnection edge $\beta_q^{(i)} = (x_{2d_i,q}^{(i)}, g_{1,q}^{(i)})$. Furthermore, X_i has an interconnection edge $\beta_{2d_i}^{(i)} = (x_{2d_i,2d_i}^{(i)}, g_{1,2k}^{(i)})$.

Overall, this specification provides $4d_i$ edges connecting X_i and G_i. As will be discussed later, $2d_i$ of these edges belong to the red path and $2d_i$ belong to the green path.

For the connections between gadget X_i and other gadgets X_j, our strategy is more involved, because we need to reproduce the topology of G and, at the same time, we need to construct a red and a green path within M. For each vertex j that is a neighbor of vertex i in G, we add two red edges between the left column of X_i and the left column of X_j, and two green edges between the top row of X_i and the top row of X_j; see again Figure 9. Crucial to our construction, the order in which we visit the neighbors of i is induced from the depth-first traversal of G specified by Algorithm 1 in Section 4.2.1.

Let $\gamma = \langle i_1, \ldots, i_{2m+1} \rangle$ be the path produced by Algorithm 1 when given vertex 1 as input, so that $i_1 = i_{2m+1} = 1$. We use a function $df_\gamma(i, k)$, for $i \in [n]$ and $k \in [2m]$, that

counts the number of interconnection edges already established for the left column (equivalently, for the top row) of X_i, at step k of the depth-first traversal, that is, right before the kth edge (i_k, i_{k+1}) of γ is processed. In this way, $df_\gamma(i, k) + 1$ is the next available position in the left column (equivalently, top row) of X_i when visiting the kth edge of γ.

For technical reasons, grid X_1 needs a special treatment: the last edge (i_{2m}, i_{2m+1}) in γ is used to enter grid X_1 for the first time, and is therefore placed at vertex $x_{1,1}^{(1)}$. Therefore, all other edges of γ that are placed at X_1 need to be shifted by one position. This means that we have to increase our count $df_\gamma(i, k)$ by one unit when $i = 1$ and $k < 2m$, and we have to treat the case of $i = 1$ and $k = 2m$ in a special way.

Formally, for $i \in [n]$ and $k \in [2m]$, we define

$$df_\gamma(i, k) = \begin{cases} |\{k' : k' < k, \, i \in \{i_{k'}, i_{k'+1}\}\}|, & \text{if } i > 1 \\ |\{k' : k' < k, \, i \in \{i_{k'}, i_{k'+1}\}\}| + 1, & \text{if } i = 1, k < 2m \\ 1, & \text{if } i = 1; k = 2m \end{cases}$$

We are now ready to specify the interconnection edges for X_i. Recall from Section 4.2.1 that, for an edge (i, j) of G, there exists a unique k such that $i_k = i$ and $i_{k+1} = j$ in γ, and there exists a unique k' such that $i_{k'} = j$ and $i_{k'+1} = i$.

- For each edge (i, j) of G, let k be as above. X_i has an interconnection edge $(x_{df_\gamma(i,k)+1,1}^{(i)}, x_{df_\gamma(j,k)+1,1}^{(j)})$. Symmetrically, X_i has an interconnection edge $(x_{1,df_\gamma(i,k)+1}^{(i)}, x_{1,df_\gamma(j,k)+1}^{(j)})$.

- For each edge (i, j) of G, let k' be as above. X_i has an interconnection edge $(x_{df_\gamma(i,k')+1,1}^{(i)}, x_{df_\gamma(j,k')+1,1}^{(j)})$. Symmetrically, X_i has an interconnection edge $(x_{1,df_\gamma(i,k')+1}^{(i)}, x_{1,df_\gamma(j,k')+1}^{(j)})$.

Informally, this specification means that for each edge (i, j) of G we have four interconnection edges between X_i and X_j: two edges from when the depth-first traversal first explores the edge from i to j, and two edges from when the depth-first traversal travels back from j to i. This amounts to $4d_i$ interconnection edges for grid X_i.

4.2.3 Component G_i. We now turn to (multi)graph G_i, $i \in [n]$, which embeds a square grid with $4k$ rows and $4k$ columns; here k is the positive integer in the input instance of the carving width problem in our reduction; see Section 4.1. We have already introduced interconnection edges $\alpha_p^{(i)}$ and $\beta_q^{(i)}$ for G_i in Section 4.2.2. In addition to these edges, we need to double some occurrences of the edges internal to the grid G_i, in order to connect the red and green paths within M. We remind the reader that M is defined as a multigraph; therefore we can introduce multiple edges joining the same pair of vertices of G_i. As will be discussed in detail later, our construction always assigns different colors (red or green) to two edges joining the same pair of vertices. An overall picture of G_i is schematically presented in Figure 10 for $d_i = 3$ and $k = 5$.

We first provide the specification of the red edges of G_i.

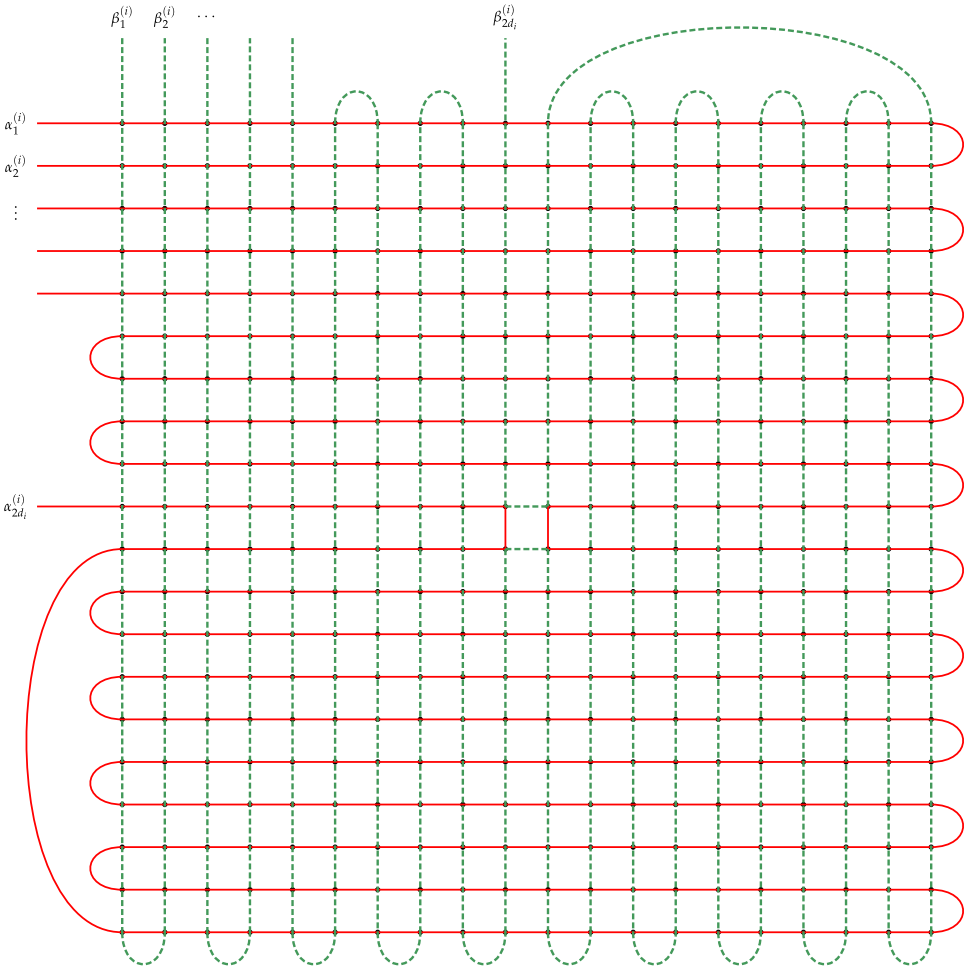- For each $p \in [2k]$, we double edge $(g_{2p-1,4k}^{(i)}, g_{2p,4k}^{(i)})$.

Figure 10
Grid gadget G_i corresponding to vertex i. We assume $d_i = 3$ and $k = 5$. The red path is drawn using a heavy red line; the green path is drawn using a dashed, heavy green line. The red path visits G_i one row at a time. The only exception is at row $2k$: after reaching the middle of that row, the red path switches to row $(2k + 1)$. The path comes back to row $2k$ only after having completed the lower half of G_i. The green path follows a symmetrical pattern in visiting G_i one column at a time, with a switch in the middle of column $2k$.

- For each p with $d_i \leq p \leq k - 1$, we double edge $(g^{(i)}_{2p,1}, g^{(i)}_{2p+1,1})$.

- For each p with $k + 1 \leq p \leq 2k - 1$, we double edge $(g^{(i)}_{2p,1}, g^{(i)}_{2p+1,1})$.

- We add edge $(g^{(i)}_{2k+1,1}, g^{(i)}_{4k,1})$.

This set of edges form the red extra edges of G_i. Symmetrically, for each red extra edge $(g^{(i)}_{p,q}, g^{(i)}_{p',q'})$ above, we also add a green extra edge $(g^{(i)}_{q,p}, g^{(i)}_{q',p'})$ to G_i, that is, the new edge has row and column reversed for both endpoints.

223

Example 6

We now provide an example that uses the specifications in this and the previous section. Consider the graph G in Figure 5, and let $k = 4$. Assume the depth-first traversal of G that produces the vertex path $\gamma = \langle 1, 2, 3, 4, 3, 1, 3, 2, 1 \rangle$. From G, k, and γ we construct the cyclic permutation multigraph M shown in Figure 11.

Let us focus on vertex 3 in G, and the associated components X_3 and G_3 in M. Because $d_3 = 3$, X_3 has size 6×6. Because vertex 3 is connected to vertices 1, 2, and 4 in G, grid X_3 has two red and two green interconnection edges to each of the components X_1, X_2, and X_4. Consider now all the edges of G impinging on vertex 3. These edges are visited twice by γ, in the order $(2, 3)$, $(3, 4)$, $(4, 3)$, $(3, 1)$, $(1, 3)$, $(3, 2)$. Accordingly, when visiting the first column of X_3 from top to bottom, we touch upon the red interconnection edges for grids X_2, X_4, X_4, X_1, X_1, and X_2. Exactly the same order is found for the green interconnection edges, when visiting the first row of X_3 from left to right.

Grid G_3 has size $4k \times 4k$, that is, 16×16. This is also the size of any other grid G_i in M. We have $2d_3 = 6$ red interconnection edges connecting G_3 and X_3; these are the $\alpha_p^{(3)}$ edges of Section 4.2.2, for $p \in [6]$. Each edge $\alpha_p^{(3)}$, $p \in [5]$, impinges on vertex $g_{p,1}^{(3)}$, and edge $\alpha_6^{(3)}$ impinges on vertex $g_{8,1}^{(3)}$. A symmetrical pattern is seen for the green interconnection edges $\beta_q^{(3)}$, $q \in [6]$, connecting G_3 and X_3.
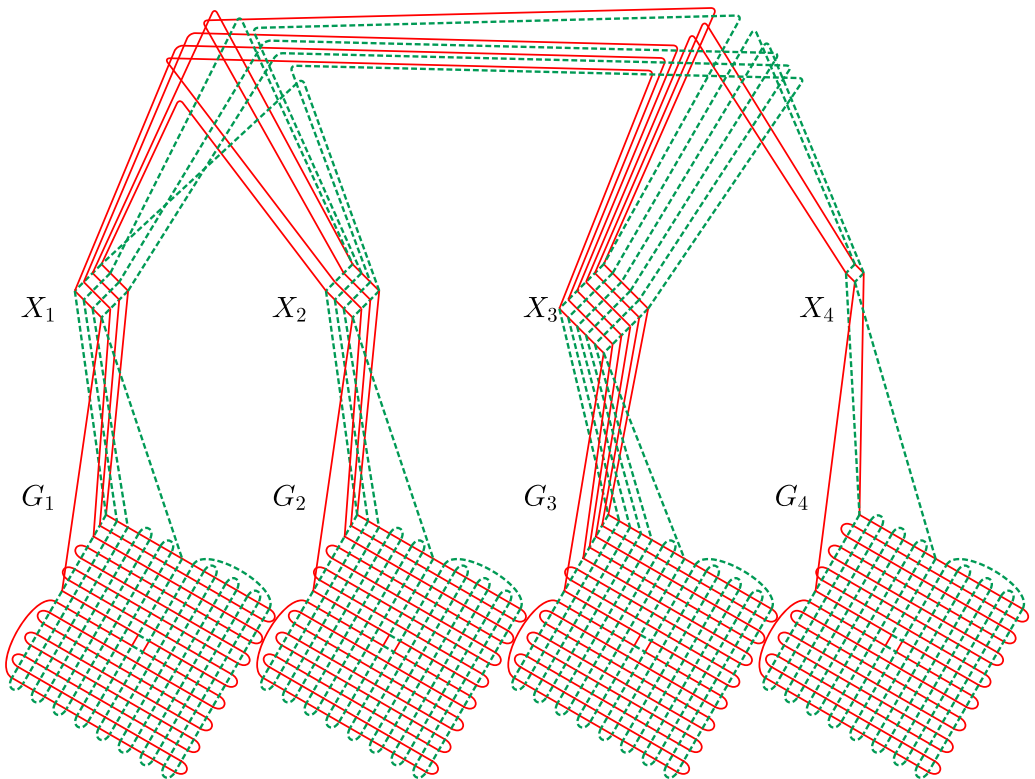

Figure 11
Cyclic permutation multigraph M constructed from the example graph G shown in Figure 5 with $k = 4$ and depth-first traversal path $\gamma = \langle 1, 2, 3, 4, 3, 1, 3, 2, 1 \rangle$. Edges are routed according to the high-level tree layout of Figure 7.

4.2.4 Cyclic Permutation Multigraph M. To summarize the previous sections, the cyclic permutation multigraph M contains components X_i and G_i for each $i \in [n]$, where n is the number of vertices of the source graph G. Besides the edges of the grid-like components X_i and G_i, M also contains some interconnection edges. More precisely, for each vertex i of G, M contains $2d_i$ red edges $\alpha_p^{(i)}$ and $2d_i$ green edges $\beta_q^{(i)}$ connecting X_i and G_i. Furthermore, for each edge (i, j) of G, M contains two red edges and two green edges connecting X_i and X_j.

We still need to show that M is a cyclic permutation multigraph, that is, all the edges mentioned above form a red Hamiltonian cycle and a green Hamiltonian cycle over the vertices of M. We start by observing that, within each component G_i, these two cycles are symmetric, in the sense that the green Hamiltonian cycle can be obtained from the red Hamiltonian cycle by switching the first and the second indices of each vertex in an edge. In other words, within G_i the green Hamiltonian cycle can be obtained from the red Hamiltonian cycle by a rotation along the axis from vertex $g_{1,1}^{(i)}$ to vertex $g_{4k,4k}^{(i)}$. This is also apparent from Figure 10. A similar observation holds for the components X_i, as apparent from Figure 9, and for the interconnection edges. For this reason, we outline in the following only the red Hamiltonian cycle of M.

The red cycle of M starts and ends at vertex $g_{1,1}^{(1)}$ in G_1. The cycle follows a depth-first traversal of G specified by Algorithm 1 in Section 4.2.1. Each time the depth-first traversal visits node i of G, the red cycle travels across one row of X_i from left to right, then passes through G_i, and then travels across the next row in X_i from right to left, before proceeding to the gadget for the next vertex of G in the depth-first traversal. Exactly how the red cycle travels through G_i differs for the first $d_i - 1$ times that G_i is visited and the final time; see Figure 10. The first $d_i - 1$ times that G_i is visited, the red cycle travels from left to right across one row, descends to the next row, and traverses it from the right to left. On the final trip into G_i, the cycle visits all the remaining rows as follows. It travels left to right across row $2(d_i - 1) + 1$ of G_i, then zig-zags across the next $2k - 2(d_i - 1) - 1$ rows in the upper half of G_i. Upon reaching vertex $g_{2k,2k+1}^{(i)}$, coming from its right and proceeding to the left, the red cycle moves to $g_{2k+1,2k+1}^{(i)}$ using an edge internal to the grid. This choice is designed to ensure, for reasons that will become clear later, that none of the extra edges added to the grid underlying G_i crosses between the grid's four quadrants.

Next, the red cycle travels from vertex $g_{2k+1,2k+1}^{(i)}$ toward the right to reach $g_{2k+1,4k}^{(i)}$, and then zig-zags across the next $2k - 1$ rows of the lower half of G_i. Upon reaching vertex $g_{4k,1}^{(i)}$, the red cycle jumps to vertex $g_{2k+1,1}^{(i)}$ using a single extra edge. It then travels toward the right to reach $g_{2k+1,2k}^{(i)}$, moves to $g_{2k,2k}^{(i)}$ using an edge internal to the grid, and finally proceeds toward the left to reach $g_{2k,1}^{(i)}$. This completes the last traversal of G_i by the red cycle.

From $g_{2k,1}^{(i)}$ the red cycle leaves G_i and enters X_i at vertex $x_{2d_i,2d_i}^{(i)}$. It then travels across the bottom row of X_i to reach vertex $x_{2d_i,1}^{(i)}$, and then proceeds to visit the gadget X_j for the next vertex j of G in the depth-first traversal, as already mentioned.

4.3 NP-Completeness

The main result in this section shows that deciding whether the carving width of a cyclic permutation multigraph is less than or equal to a given integer is an NP-complete problem. We develop this result by means of several intermediate lemmas, reducing

from the carving width decision problem for a general graph. Throughout this section, we assume that G, k, and M are defined as in the previous sections.

Lemma 2
Within a tree layout T_M of M, we can organize all vertices of G_i, for any $i \in [n]$, into a (connected) subtree T_{G_i} of T_M in such a way that there is an upper bound of $4k$ for the width of any arc internal to T_{G_i} and for the width of any arc connecting T_{G_i} to the remaining nodes of T_M.

Proof
It has been shown by Kozawa, Otachi, and Yamazaki (2010) that the carving width of an $m \times m$ grid with m even is m. We adapt here their construction to show the statement of the lemma.

The subtree T_{G_i} is built by dividing the grid G_i into four quadrants of size $2k \times 2k$, shown by black dotted lines in Figure 12. Consider first the upper left quadrant. We build a linear subtree T_{UL} by adding vertices of this quadrant one at a time in column major order. More specifically, we add columns of the quadrant from left to right, and we add vertices within each column from top to bottom, as shown Figure 12. We use symmetrical constructions for the bottom left quadrant, the bottom right quadrant,
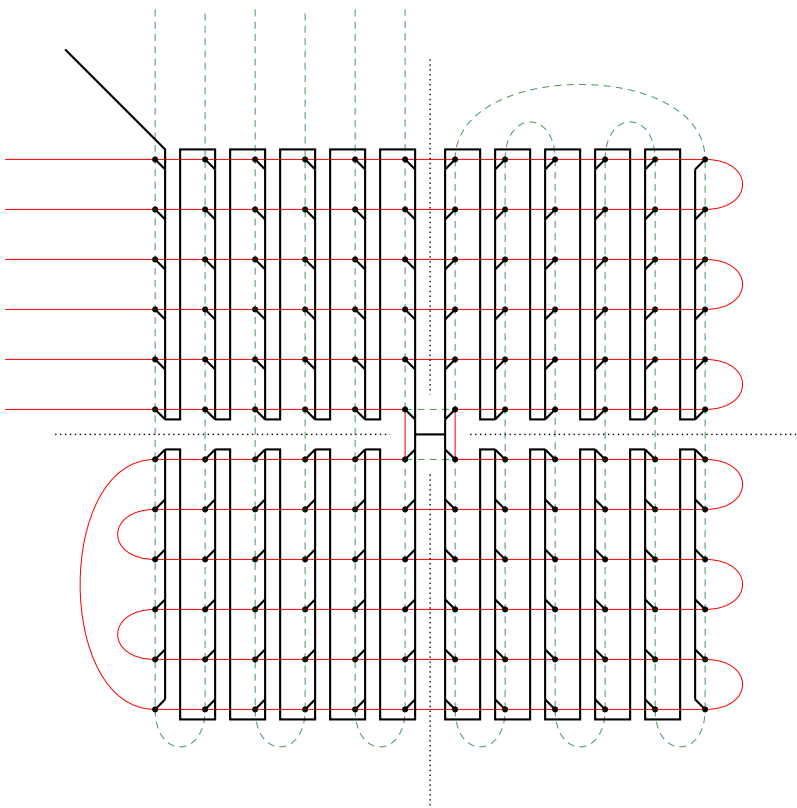

Figure 12
Subtree layout for grid G_i, assuming $d_i = k = 3$. The subtree layout is drawn using a thick black line. A dotted thin black line shows the division of grid G_i into four quadrants.

and the upper right quadrant of G_i, resulting in the linear trees T_{BL}, T_{BR}, and T_{UR}, respectively. The top-most nodes of the linear trees T_{UL} and T_{BL} are connected to a new node n_L; similarly, the top-most nodes of T_{UR} and T_{BR} are connected to a new node n_R, and n_L and n_R are connected together. This completes the specification of the tree layout T_{G_i}. Finally, an extra edge is used to connect the bottom-most internal node of T_{UL} to the remaining nodes of the tree layout T_M. See again Figure 12.

We now prove an upper bound of $4k$ for the width of any arc internal to T_{UL} and for the width of any arc connecting T_{UL} to the remaining nodes of T_M. The binary tree T_{UL} has $4k^2$ leaf nodes and $4k^2$ internal nodes. Let us name the internal nodes of T_{UL} from bottom to top, using integers in $[4k^2]$ in increasing order. Because the width of T_{UL} does not decrease at the increase of d_i, in what follows we consider the worst case of $d_i = k$.

Observe that the arc connecting internal node 1 to the remaining nodes of T_M routes $4k$ edges of G_i, namely, the interconnection edges of G_i that lead to X_i. When we move to arc $(1, 2)$ of T_{UL}, we lose the two interconnection edges impinging on vertex $g_{1,1}^{(i)}$ of G_i, but those edges are replaced by two new internal edges of G_i, namely, edges $(g_{1,1}^{(i)}, g_{2,1}^{(i)})$ and $(g_{1,1}^{(i)}, g_{1,2}^{(i)})$. Therefore, arc $(1, 2)$ of T_{UL} still routes $4k$ edges of G_i. Climbing up tree T_{UL} at arcs $(2, 3)$, $(3, 4)$, and so on, shows exactly the same pattern, with two edges impinging on the newly added vertex of G_i replaced by two new edges of G_i impinging on the same vertex. This process goes on until we reach the arc that connects node $4k^2$ (the top-most node of T_{UL}) to node n_L. This arc routes the $4k$ edges of the upper left quadrant that reach the upper right quadrant and the lower left quadrant.

To conclude our proof, we observe that the upper left quadrant embeds any of the three remaining quadrants of G_i. Because the trees T_{BL}, T_{BR}, and T_{UR} are symmetrical to T_{UL}, the former trees must also have an upper bound of $4k$ on the width of their internal arcs as well as on the width of the arcs connecting these trees to the remaining nodes of T_M. Finally, we observe that the arc connecting nodes n_L and n_R also routes $4k$ edges of G_i, namely the edges from the two left quadrants to the two right quadrants. ∎

In the case of $d_i = k$, the grid component X_i is the same as the top left quadrant of G_i. Therefore, the analysis provided in the proof of Lemma 2 can also be used to prove the next lemma.

Lemma 3
Within a tree layout T_M of M, we can organize all vertices of X_i, for any $i \in [n]$, into a subtree T_{X_i} of T_M in such a way that there is an upper bound of $4k$ for the width of any arc internal to T_{X_i} and for the width of any arc connecting T_{X_i} to the remaining nodes of T_M.

The tree layout for X_i is a long chain, isomorphic to the tree layout of the upper left quadrant of G_i in Figure 12. In what follows, we refer to the node of the layout immediately above $x_{1,1}^{(i)}$ as top-most, and the node immediately above $x_{2d_i,2d_i}^{(i)}$ as bottom-most. We can now prove the correctness of our reduction for one direction.

Lemma 4
If G has a tree layout T of carving width k, then M has a tree layout T_M of carving width $4k$.

Proof
Each leaf node of T is associated with a vertex i of G. The main idea of this proof is to construct a tree layout T_M by using T as the top level of T_M, and by attaching a

tree layout for each grid X_i and G_i under the node of T associated with vertex i. As an example, for the cyclic permutation multigraph M of Figure 11, this will provide the tree layout schematically depicted in Figure 7.

For each $i \in [n]$, we use the tree layout $T(G_i)$ from Lemma 2 and connect it to the tree layout $T(X_i)$ from Lemma 3. The connection arc is created from the bottom-most internal node of subtree T_{UL} of $T(G_i)$, to the top-most node of $T(X_i)$. The tree layout $T(X_i)$ is in turn connected to the tree layout T for G. The connection is established by merging the bottom-most internal node of $T(X_i)$ and the leaf node of T associated with vertex i of G.

From Lemma 2 and Lemma 3, we have that any of the trees $T(G_i)$ and $T(X_i)$ have width at most $4k$. The edges of M that are routed through the arc connecting $T(G_i)$ to $T(X_i)$ are exactly the $4d_i \leq 4k$ interconnection edges between G_i and X_i. The edges of M routed through the connection between $T(X_i)$ and T are the $4d_i \leq 4k$ interconnection edges between X_i and all the X_j for j a neighbor of vertex i in G. Finally, the top-level subtree T within T_M also has carving width at most $4k$. To see this, observe that for each edge (i, j) routed by an arc a of the tree layout T of G, the same arc a in (the copy of) T used as a subtree of the tree layout T_M routes two red and two green edges connecting components X_i and X_j of M. Because the carving width of the tree layout T of G is k, we conclude that the carving width of the tree layout T within T_M is $4k$. ∎

We now deal with the other direction in the correctness of our reduction. We must show that if M has a tree layout of carving width $4k$, then G has a tree layout of carving width k. This will be Lemma 8, for which we will need to develop a few intermediate lemmas. We introduce our first intermediate lemma by means of a simple example.

Example 7
Consider the tree layout of grid G_i, $i \in [n]$, that has been depicted in Figure 12 as a subtree of a tree layout T_M of M. Let a be the arc that connects the bottom right quadrant of the grid to the remainder of the layout. Using the notation in the proof of Lemma 2, arc a can be written as (n_R, n_{BR}), with n_{BR} the top-most node of linear tree T_{BR}. Observe that there are $4k$ edges of M routed through arc a (in Figure 12 we have $k = 3$). These are the $2k$ red edges that connect the bottom right quadrant with the bottom left quadrant of G_i, and the $2k$ green edges that connect the bottom right quadrant with the top right quadrant. These $4k$ edges of M are all internal to grid G_i, meaning that they connect vertices within G_i. Finally, observe the subtree rooted at node n_{BR} contains $4k^2$ vertices from G_i.

The special properties of arc a that we have mentioned here are not dependent on the choice of the tree layout T_M. In fact the next lemma shows that, for every choice of a tree layout of M having carving width $4k$, and for every grid G_i in M, $i \in [n]$, there always exists an arc that satisfies the properties of Example 7. Intuitively, this happens because the vertices in a grid have a relatively high degree of interconnections, and when we pick up a sufficiently large set of vertices of a grid G_i, we have a large number of edges connecting the vertices in our set to the remaining vertices of G_i. This in turn means that, in any tree layout T_M, any subtree T_i containing a sufficiently large set of vertices of G_i must route a large number of edges from G_i through the arc a that connects T_i to the rest of T_M. If T_M has bounded carving width, the edges routed through arc a saturate the width of this arc, making it impossible for other vertices not from G_i to be placed within T_i. In other words, there is always some core set of vertices from G_i that is placed in some subtree of a tree layout of M, and this subtree must exclude vertices from other grids.

Lemma 5
Let T_M be a tree layout of M having carving width $4k$. For each vertex i of G, there exists an arc a_i of T_M satisfying all of the following properties:

   (i)   The width of a_i is $4k$.

   (ii)   The edges of M routed through a_i are all internal to grid G_i.

   (iii)   One of the two subtrees obtained by removing a_i from T_M contains only vertices internal to G_i; we call this the subtree below a_i.

   (iv)   The subtree below a_i contains at least $4k^2$ vertices from G_i.

Proof
For a subtree T of T_M, let $L_i(T)$ be the number of vertices of G_i that are at the leaves of T. Consider an arc a of T_M, and let T_1 and T_2 be the two subtrees of T_M obtained by removing a from T_M; this is exemplified in Figure 13. We say that a is **balanced** for G_i if the choice of a minimizes $|L_i(T_1) - L_i(T_2)|$.

The following argument is a standard one, see for instance Kozawa, Otachi, and Yamazaki (2010, Lemma 3.1). Assume that a is a balanced arc in T_M. Let $|G_i| = (4k)^2$ be the number of vertices of G_i. Without loss of generality, we assume that $L_i(T_1) \geq L_i(T_2)$. Because $|G_i| = L_i(T_1) + L_i(T_2)$, this implies $L_i(T_2) \leq \frac{1}{2}|G_i|$. Because $|G_i| \geq 16$, we must have $L_i(T_1) > 1$, otherwise a would not be balanced. Then the root of T_1 must have two children, rooting subtrees T_1' and T_1'' of T_1; see again Figure 13. We must have $L_i(T_1') \leq L_i(T_2)$ and $L_i(T_1'') \leq L_i(T_2)$, otherwise a would not be balanced. This in turn implies $L_i(T_2) \geq \frac{1}{3}|G_i|$. To summarize these inequalities, we have

$$\frac{1}{3}|G_i| \leq L_i(T_2) \leq \frac{1}{2}|G_i| \tag{6}$$

For each vertex i of G, we now choose a_i in the statement of the lemma to be a balanced arc for G_i. In the following discussion we focus on an arbitrary choice of $i \in [n]$, and again we let T_1 and T_2 be the two subtrees of T_M obtained by removing a_i from T_M, with $L_i(T_1) \geq L_i(T_2)$, so that we can use Equation (6).
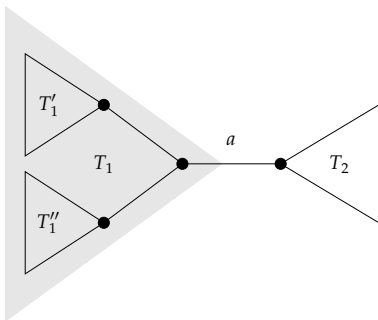

Figure 13
Definition of balanced arc a of T_M for some grid G_i. Trees T_1 and T_2 are the two subtrees of T_M obtained by removing a from T_M.

Let S be a grid of size $4k \times 4k$, that is, a square grid with $|G_i|$ nodes. It is known that, for any choice of s nodes from S with $\frac{1}{3}|G_i| \leq s \leq \frac{1}{2}|G_i|$, there are at least $4k$ edges of S connecting the chosen nodes to nodes in the complement set. For this claim, see for instance Ahlswede and Bezrukov (1995) or Rolim, Sýkora, and Vrt'o (1995). This claim must also be true for the edges internal to G_i, since G_i embeds the grid S. From Equation (6) we then have that at least $4k$ edges internal to G_i are routed through a_i. Furthermore, since T_M has carving width $4k$, the width of a_i cannot exceed $4k$. We then conclude that conditions (i) and (ii) are both satisfied.

Let $\overline{N}_i$ be the set of vertices of M that are not vertices of G_i. It is easy to see from the specification in Section 4.2 that the sub-multigraph of M induced by $\overline{N}_i$ is connected. (This is the very reason why we introduced the components X_i in our construction of M.) This implies that for any pair of vertices in $\overline{N}_i$, there exists a path in M connecting the two vertices that is entirely composed of edges that are not internal to G_i. If both T_1 and T_2 have some vertex from $\overline{N}_i$, there would be an edge not internal to G_i routed through a_i, in contrast to condition (ii). We therefore conclude that one among T_1 and T_2 contains only vertices internal to G_i, satisfying condition (iii). This subtree is called the subtree below a_i.

From Equation (6) and by our choice of T_1, we have $\frac{1}{3}|G_i| \leq L_i(T_2) \leq L_i(T_1)$. Furthermore, $4k^2 < \frac{1}{3}|G_i|$, and hence both T_1 and T_2 have at least $4k^2$ vertices from G_i. Regardless of which of T_1 or T_2 is below a_i, we conclude that condition (iv) is satisfied. ∎

Consider a square grid Q of size $4k \times 4k$, and let d be some fixed integer with $d \in [k]$. Assume that $2d$ vertices in the left-most column of Q and $2d$ vertices in the top-most row of Q have been selected as **connector vertices**. Observe that the number of connector vertices in Q is either $4d - 1$ or $4d$, depending on whether the vertex at the top-left corner of Q is selected twice or not. Let S be any subset of the vertices of Q. Recall from Section 3 that the **edge boundary** of S in Q, written $\partial_Q(S)$, is the set of edges of Q that connect vertices in S and vertices in $\overline{S}$, the complement set of S.

Lemma 6
Let Q be a square grid of size $4k \times 4k$ and let $d \in [k]$. Assume $2d$ connector vertices in the left-most column and $2d$ connector vertices in the top-most row of Q. Let also S be a set of vertices of Q such that $|S| \geq 4k^2$ and S does not contain any connector vertex. Then $|\partial_Q(S)| \geq 4d$.

Proof
We distinguish three cases in the following, depending on whether the set $\overline{S}$ contains any entire column and/or any entire row of Q.

Case 1: $\overline{S}$ contains at least one entire column of Q and at least one entire row of Q. (This part of the proof is adapted from Kozawa, Otachi, and Yamazaki [2010, Propositions 4.4, 4.5, and 4.7].) Let r be the number of rows of Q that have at least one vertex in S. Within each such row there must be a vertex not in S, since at least one entire column of Q is not in S. This means that at least one edge of this row is in the set $\partial_Q(S)$, for a total of r edges in $\partial_Q(S)$. A similar argument applies to the number of columns c of Q that have at least one vertex in S.

Because rows and columns have disjoint sets of edges, we have $|\partial_Q(S)| \geq r + c$. It is well known that the arithmetic mean $(r + c)/2$ is always larger than or equal to the geometric mean $\sqrt{rc}$. Because $rc \geq |S|$, we can write $|\partial_Q(S)| \geq 2\sqrt{rc} \geq 2\sqrt{|S|} \geq 2\sqrt{4k^2} = 4k \geq 4d$.

Case 2: $\overline{S}$ contains at least one entire column of Q but no entire row, or else $\overline{S}$ contains at least one entire row of Q but no entire column. In the first case we have $r = 4k$ and, as explained in Case 1, $4k \geq 4d$ edges in the set $\partial_Q(S)$. Symmetrically, in the second case we have $c = 4k$ and thus $4k \geq 4d$ edges in $\partial_Q(S)$.

Case 3: $\overline{S}$ does not contain any entire column of Q, nor any entire row of Q. In this case each row of Q has at least one vertex in S. Because the connector vertices are all contained in the set $\overline{S}$, each of the $2d$ rows of Q that have a connector vertex contributes at least one edge to the set $\partial_Q(S)$. A symmetrical argument applies to the $2d$ columns of Q that have a connector vertex. Again, because the rows and the columns of Q have disjoint edges, we conclude that $|\partial_Q(S)| \geq 4d$. ∎

Let T_M be any tree layout of M with carving width $4k$. In Lemma 5 we associated each vertex i of G with an arc a_i of T_M having some specific properties. Similarly, our next lemma associates each edge (i, j) of G with a set of four paths in T_M. These paths are routed through a_i and through a_j, and do not share any of their edges. Using this property, we will later be able (in Lemma 8) to derive the carving width of G from the carving width of M. We use a simple example here to illustrate the special paths we are looking for.

Example 8

Consider once more the grid G_i and its subtree layout depicted in Figure 12, where we have $d_i = k = 3$. We report G_i in Figures 14 and 15, where for convenience we have ignored the associated tree layout. As in Example (7), consider the arc in the linear layout that connects the bottom right quadrant of G_i to the remainder of the layout. Here we denote this arc as a_i. There are $d_i = 3$ neighbors of i in G, and we need to associate four paths with each neighbor, for a total of 12 paths routed through a_i. In
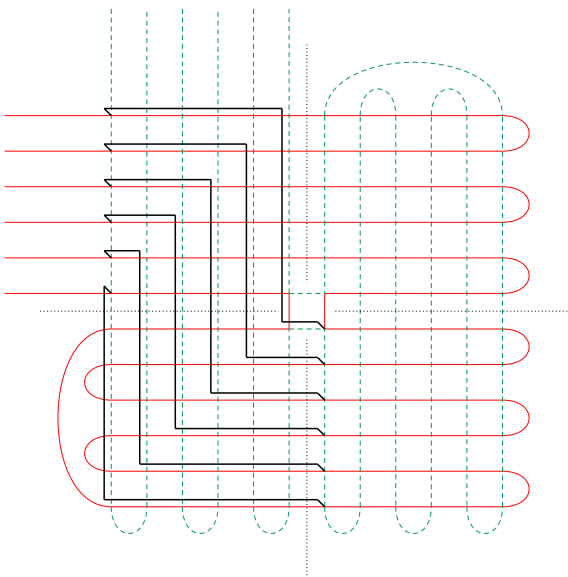

Figure 14
Paths starting with the red edges routed through arc a_i of T_M. These paths and the paths in Figure 15 do not share any of their edges.
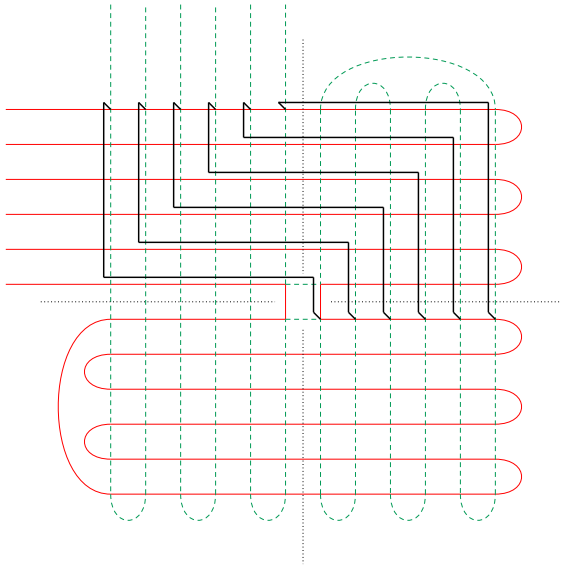
Figure 15
Paths starting with the green edges routed through arc a_i of T_M. These paths and the paths in
Figure 14 do not share any of their edges.

what follows we focus our attention on the first segment of each of these 12 paths,
more precisely, the segment of these paths that starts at an edge of M routed through a_i
and ends at some edge of M that leaves G_i to reach grid X_i.

In Figure 14 we outline the first segment of six paths of M routed through arc a_i,
starting with the red edges that connect the bottom right quadrant with the bottom left
quadrant of G_i. These paths reach the six vertices in the left-most column of G_i that are
connected with the grid X_i and will eventually reach the grids X_j and G_j, for each vertex
j that is a neighbor of i in G. Similarly, in Figure 15 we outline six more paths of M routed
through arc a_i, starting with the green edges that connect the bottom right quadrant with
the top right quadrant of G_i. These paths reach the six vertices in the top-most row of G_i
that will eventually reach the grids X_j and G_j for each neighbor j of i.

Finally, observe that these 12 paths routed through a_i share some of their vertices in
the top left quadrant of G_i but do not share any of their edges. To see this, consider the
diagonal of the top left quadrant from the bottom left corner to the upper right corner.
Notice then that the paths in Figure 14 use the vertical edges of G_i below the diagonal,
and the paths in Figure 15 use the horizontal edges below the diagonal. Similarly,
the paths in Figure 14 use the horizontal edges above the diagonal, and the paths in
Figure 15 use the vertical edges above the diagonal.

The next lemma generalizes the previous example, showing that for any tree layout
T_M we can associate each edge (i, j) of G with a set of four paths in M that are edge
disjoint. We need to introduce some auxiliary notation. Let T_M and a_i, $i \in [n]$, be defined
as in Lemma 5. Consider an arc (i, j) in the source graph G. A path in M is called an
(i, j)-**path** for T_M if it starts at a vertex inside the subtree below a_i (see Lemma 5(iii) for
the definition of this subtree) and it ends at a vertex inside the subtree below a_j. We also
say that two paths in M are **edge disjoint** if they do not share any edge (they might
however share some vertex).

Lemma 7

Let T_M be a tree layout of M having carving width $4k$. There exists a set $\mathcal{P}$ of paths in M satisfying both of the following properties:

(i)   For every edge (i, j) in the source graph G, $\mathcal{P}$ contains four (i, j)-paths for T_M.

(ii)   Every two paths in $\mathcal{P}$ are edge-disjoint.

Proof

We call **connector edges** the edges of M that are not internal to any component G_i, $i \in [n]$. We call **connector vertices** those vertices of G_i, $i \in [n]$, that have an impinging connector edge. In this way, each G_i has $4d_i - 1$ connector vertices. These are placed in the left-most column of G_i and in the top-most row of G_i. We claim that, for each $i \in [n]$, the connector vertices of G_i must all be outside of the subtree of T_M below a_i. To see this, let us assume that some connector vertex of G_i is found inside the tree below a_i. Because connector vertices are connected to vertices outside of G_i, there would be a vertex not internal to G_i inside the tree below a_i, against Lemma 5(iii), or else there would be a connector edge routed through a_i, against Lemma 5(ii). We therefore conclude that the connector vertices of G_i must all be outside of the tree below a_i.

The idea underlying the proof is to define each (i, j)-path in $\mathcal{P}$ as the concatenation of three sub-paths, called **segments**, specified as follows.

- The first segment starts with an edge routed through arc a_i. By definition, one of the two end vertices of such an edge is inside the subtree below a_i, and this is also the starting vertex of the segment. Furthermore, the segment only uses edges that are internal to G_i, and ends with a connector vertex of G_i.

- The second segment only uses connector edges, and ends with a connector vertex of G_j.

- The third segment only uses edges internal to G_j, and ends with an edge routed through arc a_j. The end vertex of such an edge that is inside the subtree below a_j is the end vertex of the segment.

From now on, we focus on a specific vertex i of G and prove the existence of $4d_i$ edge disjoint (i, j)-paths for vertices j that are neighbors of i in G. We do this by separately specifying the three segments of the paths.

First segment. From Lemma 5(ii), there are $4k$ edges internal to G_i that are routed through arc a_i. We know that $k \geq d_i$, since the carving width of a graph is always at least its degree. Thus we start our segments with any choice of $4d_i$ disjoint edges routed through arc a_i. We now show that these edges can be extended to $4d_i$ segments that reach the $4d_i - 1$ connector vertices of G_i, without sharing any of their edges. We do this by reducing our problem to a network flow problem.

If we assume that edges of G_i have flow capacity of one, each segment corresponds to a flow through G_i having capacity one. The existence of $4d_i$ edge disjoint segments is then equivalent to the existence of a flow of capacity $4d_i$ from the subtree below a_i to the $4d_i - 1$ connector vertices of G_i. Let $V(G_i)$ be the set of vertices of G_i. The max-flow min-cut theorem states that this flow can be realized if and only if any cut of G_i with the

source vertices on one side and the target vertices on the other side has capacity not less than $4d_i$.

In our specific case, consider the square grid Q underlying component G_i. Let $S \subseteq V(G_i)$ be a vertex set including all the vertices in the subtree below a_i and excluding all the $4d_i - 1$ connector vertices of G_i. If we can show the relation

$$|\partial_Q(S)| \geq 4d_i \qquad (7)$$

for any choice of S as before, then we have proved the existence of the desired $4d_i$ edge disjoint segments.

In order to prove Equation (7), we observe that the set of vertices in the subtree below a_i has size greater than or equal to $4k^2$, by Lemma 5(iv). Furthermore, observe that there are $2d_i$ connector vertices of G_i in the left-most column of Q as well as $2d_i$ connector vertices in the top-most row. Under these conditions, we can apply Lemma 6 to the grid Q, with $d = d_i \leq k$, and derive Equation (7). We then conclude that there exist $4d_i$ edge disjoint first segments starting at vertices in the subtree below a_i, as desired.

Second segment. As already observed in Section 4.2.2, for every connector vertex in the left-most column of G_i there is a red segment to some connector vertex in the left-most column of G_j, where j is some neighbor of i in G. Symmetrically, for every connector vertex in the top-most row of G_i there is a green segment to some connector vertex in the top-most row of some G_j. This provides a total of $4d_i$ segments. These segments are all edge disjoint, by construction of the components X_h, $h \in [n]$, and by construction of the interconnection edges.

Third segment. We observe that the third segment of an (i, j)-path is the reversal of the first segment of a (j, i)-path. Therefore we can use our argument for the first segments to show the existence of $4d_i$ edge disjoint third segments ending at a vertex in the subtree below a_j.

To summarize, for each $i \in [n]$ we have shown the existence of $4d_i$ edge disjoint (i, j)-paths for vertices j that are neighbors of i in G, to be added to set $\mathcal{P}$. To complete the proof, we observe that two paths in $\mathcal{P}$ having disjoint start vertices must have edge disjoint first segments, since these segments are defined for edges that are internal to different components G_i. A similar argument applies to the third segments of paths having disjoint end vertices. ∎

Lemma 8
If M has a tree layout T_M of width at most $4k$, then G has a tree layout T of width at most k.

Proof
For each $i \in [n]$, let arc a_i and the subtree below a_i be defined as in Lemma 5. We denote by $r(a_i)$ the root of the subtree below a_i. The tree layout T of G is constructed from T_M as follows.

- For each $i \in [n]$, we prune from T_M the subtree below a_i, that is, we remove from T_M all the arcs of this subtree and all of its nodes but $r(a_i)$.

- For every $i, j \in [n]$ with $i \neq j$, consider the unique path in T_M that joins nodes $r(a_i)$ and $r(a_j)$. We remove from T_M all nodes that are not in any such path, along with the arcs impinging on the removed nodes.

- For any of the remaining nodes of T_M with degree 2, we remove the node and "merge" the impinging arcs into a new arc.

Observe that T has exactly n leaves, represented by the nodes $r(a_i)$, $i \in [n]$. We then place each vertex i of G at node $r(a_i)$. We show now that this tree layout of G has width no larger than k.

Let a be an arbitrary arc of T. Let a_M be the arc of T_M equal to a, if a_M has been preserved in the construction of T; otherwise, let a_M be an arbitrary arc of T_M that has been merged into a (in the third bullet of the construction). Assume that w_a is the width of a. According to the definition of width of an arc, there are exactly w_a edges (i, j) in G, such that the paths in T joining nodes $r(a_i)$ and $r(a_j)$ share arc a. We denote by $\mathcal{S}_a$ this set of edges.

Lemma 7 states that for each edge $(i, j) \in \mathcal{S}_a$, there are 4 (i, j)-paths through M, for a total of $4w_a$ paths through M that are pairwise edge-disjoint. From the construction of T, we know that arc a_M must be in the path within T_M joining nodes $r(a_i)$ and $r(a_j)$. If we remove a_M from T_M, nodes $r(a_i)$ and $r(a_j)$ are no longer connected, since paths are unique in a tree. This implies that, in the tree layout T_M, our $4w_a$ paths through M associated with the edges in $\mathcal{S}_a$ must all be routed through arc a_M. Because these paths are pairwise edge-disjoint, we conclude that there must be $4w_a$ distinct edges of M routed through a_M. Because in our tree layout the width of a_M is at most $4k$, we can write $4w_a \leq 4k$ and hence $w_a \leq k$. Since arc a was chosen arbitrarily, this concludes our proof. ∎

We can now provide the main result of this section.

Theorem 1
Let M be a cyclic permutation multigraph and let $k \geq 1$ be some integer. The problem of deciding whether M has carving width less than or equal to k is NP-complete.

Proof
For the hardness part, we reduce from the problem of deciding whether a (standard) graph G has carving width less than or equal to k, which is an NP-complete problem, as already mentioned. We construct the cyclic permutation multigraph M from G as in Section 4.2. It is not difficult to see that the construction can be carried out in time $\mathcal{O}(nk^2 + e)$, where n and e are the number of vertices and edges, respectively, in G. Because we can assume $k \leq e$, we have that the reduction takes polynomial time in the size of the input. By combining Lemmas 4 and 8, we have that G has carving width k if and only if M has carving width $4k$. This completes our reduction.

For the completeness part, given a cyclic permutation multigraph M, we can guess a tree layout T, and accept if the width of T is less than or equal to k. All of this computation can be carried out in linear time. ∎

4.4 NP-Completeness of Fan-out

We can now present the main result on the optimization of the fan-out of parsing strategies for an SCFG rule.

Theorem 2

Let s be a synchronous rule and let $k \geq 1$ be some integer. The problem of deciding whether there exists a binary parsing strategy for s with fan-out less than or equal to k is NP-complete.

Proof

This directly follows from Lemma 1 and from Theorem 1. ■

5. Time Complexity

In this section, we address the problem of finding the parsing strategy with the optimal time complexity. The time complexity of one combination step in a parsing strategy is determined by the total number of variables involved in the step, which is the total number of distinct endpoints of the spans being combined. Time complexity can differ from space complexity due to the fact that the time complexity takes into account which endpoints are shared in the two groups of spans being combined.

Time complexity of general parsing problems can be analyzed using the graph-theoretic concept of treewidth, which we now proceed to define precisely. A **tree decomposition** of a graph $G = (V, E)$ is a type of tree having a subset of G's vertices at each node. We define the nodes of this tree T to be the set I, and its edges to be the set F. The subset of V associated with node i of T is denoted by X_i. A tree decomposition is therefore defined as a pair $(\{X_i \mid i \in I\}, T = (I, F))$, where each $X_i, i \in I$, is a subset of V, and tree T has 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 $X_i \bigcap X_k \subseteq X_j$.
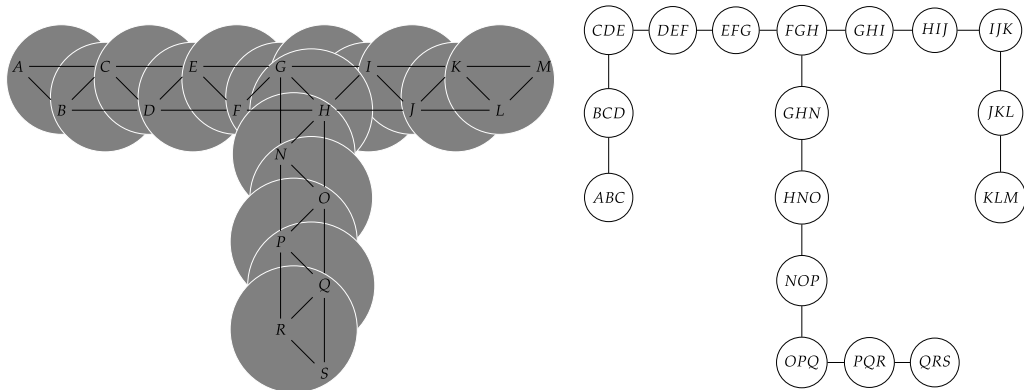

Figure 16
A tree decomposition of a graph is a set of overlapping clusters of the graph's vertices, arranged in a tree. This example has treewidth 2.

The **treewidth** of a tree decomposition $(\{X_i\}, T)$ is $\max_i |X_i| - 1$. The treewidth of a graph is the minimum treewidth 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 treewidth as being optimal.

In general, more densely interconnected graphs have higher treewidth. Any tree has treewidth 1, a graph consisting of one large cycle has treewidth 2, and a fully connected graph of n vertices has treewidth $n - 1$. Low treewidth indicates some treelike structure in the graph, as shown by the example with treewidth 2 in Figure 16. As an example of the running intersection property, note that the vertex N appears in three adjacent nodes of the tree decomposition. Finding the treewidth of a graph is an NP-complete problem (Arnborg, Corneil, and Proskurowski 1987).

We use a representation known as a **dependency graph** to analyze the time complexity of parsing for a given SCFG rule. Dependency graphs (known under a large number of different names) are used to represent the interaction between variables in constraint satisfaction problems, and have one vertex for each variable, and an edge between vertices representing any two variables that appear together in a constraint. In parsing problems, the variables consist of indices into the string being parsed, and the constraints require that a nonterminal on the right-hand side of a rule has previously been identified. Thus, the graph has a vertex for each endpoint involved in the rule, and a clique for each nonterminal (including the left-hand side nonterminal) connecting all its endpoints.

Example 9
The dependency graph for a CFG rule with n nonterminals, including the left-hand side nonterminal, is a cycle of n vertices. For instance, the CFG rule $S \to A\,B\,C\,D$ is shown in the left part of Figure 17. The five vertices represent the possible endpoints that need to be considered when parsing any instantiation of the rule, and each arc is a nonterminal appearing in the rule, including the left-hand side nonterminal. In this way, each nonterminal has two endpoints, and adjacent nonterminals share one endpoint. In particular, the left-hand side nonterminal shares one endpoint with the left-most
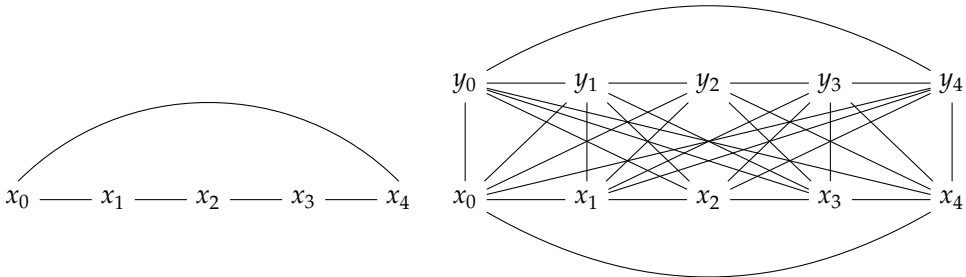

Figure 17
Dependency graph for the CFG rule $S \to A\,B\,C\,D$ (left) and the SCFG rule
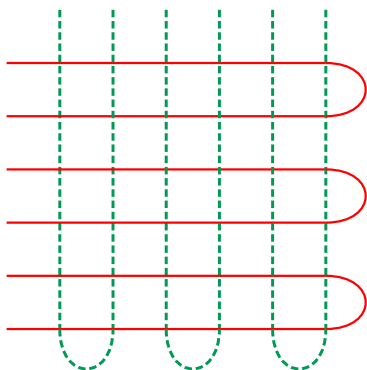$[S \to A^{[1]} B^{[2]} C^{[3]} D^{[4]},\ S \to B^{[2]} D^{[4]} A^{[1]} C^{[3]}]$ (right).

Figure 18
Gadget grid G_i in the construction of cyclic permutation multigraph M, assuming $d_i = 3$.

nonterminal in the right-hand side and one endpoint with the right-most nonterminal in the right-hand side.

In the case of an SCFG rule, each nonterminal has four endpoints, two on the English side and two on the Chinese side. If the rule has n linked nonterminals, including the left-hand side linked nonterminal, the dependency graph consists of $2n$ vertices and n cliques of size four. For instance, the SCFG rule $[S \rightarrow A^{\text{\textcircled{1}}} B^{\text{\textcircled{2}}} C^{\text{\textcircled{3}}} D^{\text{\textcircled{4}}},\ S \rightarrow B^{\text{\textcircled{2}}} D^{\text{\textcircled{4}}} A^{\text{\textcircled{1}}} C^{\text{\textcircled{3}}}]$ is shown in the right part of Figure 17.

A tree decomposition of the dependency graph corresponds directly to a parsing strategy, and the treewidth of the graph plus one is the exponent in the time complexity of the optimal parsing strategy (Gildea 2011). Each cluster of vertices in the tree decomposition corresponds to a combination step in a parsing strategy. The running intersection property of a tree decomposition ensures that each endpoint in the parsing rule has a consistent value at each step. Treewidth depends on the number of vertices in the largest cluster of the tree decomposition, which in turn determines the largest number of endpoints involved in any combination step of the parsing strategy.

Our hardness result in this section is a reduction from treewidth of general graphs. Given an input graph G to the treewidth problem, we construct an SCFG rule using techniques similar to those in the previous section. We then show that finding the parsing strategy with optimal time complexity for this rule would imply an approximation bound on the treewidth of the original graph G.

The precise construction proceeds as follows. Given an instance of the treewidth problem consisting of a graph G and integer k, we construct a cyclic permutation multigraph M as in the previous section, with the exception that the grid G_i will contain only $2d_i$ rows and $2d_i$ columns. This grid, shown in Figure 18, is a simplified version of the grid used in Figure 10. Furthermore, there are no gadgets X_i in M, but rather, for each edge (i, j) in G, there are a total of four edges in M connecting G_i directly to G_j. We refer to these edges as **connector edges**, and we refer to edges internal to some G_i as **internal edges**.

We construct a dependency graph D by replacing each edge (i, j) in M with a vertex $v(i, j)$ in D, and adding edges $(v(i, j), v(i, k))$ to D connecting all vertices in D that derive from adjacent edges in M, as shown in Figure 19. As is required for the dependency graph representation, this gives us a graph with one vertex corresponding to each boundary between adjacent nonterminals in the SCFG rule and a clique connecting the
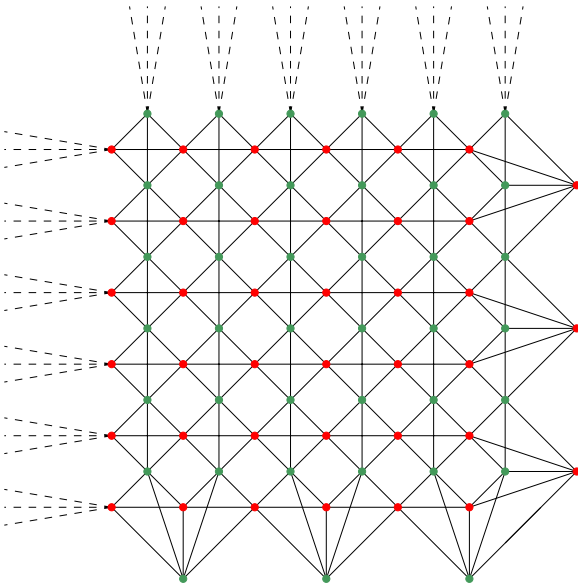
Figure 19
Grid D_i of dependency graph D. Connector vertices are in the left-most column and top-most row, and have edges, shown as dashed lines, internal to other grids D_j.

vertices for the four boundary points of each nonterminal (two boundaries in English and two in Chinese).

We refer to vertices in D derived from connector edges in M as **connector vertices**, and we refer to vertices in D derived from internal edges in M as **internal vertices**. We refer to the subgraph of D derived from vertex i of G as D_i. To be precise, D_i consists of connector vertices and internal vertices derived from edges of M incident on vertices of G_i, and the edges of D connecting these vertices. We refer to the vertices of D_i as red or green, depending on whether they are derived from red or green edges in M.

We now show that any dependency graph D constructed from a cyclic permutation multigraph M is the dependency graph for some SCFG rule r. Interpret an arbitrary vertex in M as the left-hand side nonterminal. Each green edge represents a boundary in English shared between two nonterminals, including the boundary shared between the first right-hand side nonterminal and the left-hand side nonterminal, and the boundary shared between the last right-hand side nonterminal and the left-hand side nonterminal. Similarly, each red edge represents the boundary between two nonterminals in Chinese. We then replace each edge in M with a vertex in D, and replace each vertex in M with a clique in D. D is the dependency graph for an SCFG rule, because it has a vertex for each boundary, and a clique that connects the boundaries of each nonterminal (including the left-hand side nonterminal).

With this construction in place, we can prove the main result of this section.

Theorem 3
A polynomial time algorithm for finding the parsing strategy of an SCFG rule having the lowest time complexity would imply a polynomial time constant-factor approximation algorithm for the treewidth of graphs of fixed degree.

Proof
Let T be a tree decomposition of G having treewidth $k-1$, and let $\Delta(G)$ be the maximum degree of a vertex of G. Let M and D be the cyclic permutation multigraph and the dependency graph, respectively, constructed from G as previously specified. We can construct a tree decomposition T_D of D having treewidth $4k\Delta(G)-1$ by replacing each occurrence of vertex i in T with the $4d_i$ connector vertices in D that are derived from the $4d_i$ connector edges in M that are adjacent to G_i. From one of the nodes of T_D containing the connector vertices of i, we attach a subtree containing the remaining vertices of D_i. This subtree has treewidth $4d_i-1$, and can be constructed in a linear chain adding vertices from D_i one at a time. That is, we process the nodes in D_i in column-major order, at each step constructing a new node in the tree decomposition of D by adding one vertex of D_i, either red or green, and removing the corresponding vertex from the previous red or green column respectively. In sum, if $k-1 = OPT = \mathbf{tw}(G)$, then

$$\mathbf{tw}(D) < 4\Delta(G)(OPT+1)$$

Suppose that we have an algorithm for finding the treewidth of dependency graphs derived from SCFG rules. Given a tree decomposition T_D of D of treewidth $k'-1$, a valid tree decomposition T_G of G having treewidth at most $2k'-1$ can be constructed by replacing each connector vertex in T_D with the corresponding two vertices i and j of G, and replacing occurrences of internal vertices of D with the corresponding vertex i of G. This is a valid tree decomposition of G, because of the following.

- For any edge (i, j) in G, there is a node in T_D containing the corresponding connector vertex in D, and hence a node in T_G containing both i and j.

- Any two vertices u and v in D that are associated with i in G (either as connector vertices or as internal vertices) are connected by a path in D containing only vertices associated with i. Hence, the path between any two occurrences of i in T_G contains nodes that also contain i, guaranteeing the running intersection property of T_G.

Thus, by constructing D from G, finding an optimal tree decomposition of D, and then translating the result into a tree decomposition of G, we have a tree decomposition of G having width SOL where $SOL \leq 2\mathbf{tw}(D)$. Combining the results, we have

$$SOL < 8\Delta(G)(OPT+1)$$

Therefore, an algorithm for minimizing the treewidth of dependency graphs of SCFG rules would imply the existence of a constant-factor approximation algorithm for the treewidth of graphs of fixed degree. ∎
The existence of a polynomial time constant-factor approximation algorithm for the treewidth of graphs of fixed degree is a longstanding open problem. Therefore Theorem 3 suggests that designing a polynomial time algorithm for finding the parsing strategy of an SCFG rule having optimal time complexity is a difficult task.

In the other direction, a proof of hardness for finding the optimal time complexity for SCFG parsing would also require progress on a longstanding open problem, as stated by the following theorem.

Theorem 4
If it is NP-hard to find the parsing strategy with optimal time complexity for an SCFG rule, then it is also NP-hard to find the treewidth of graphs of degree 6.

Proof
Dependency graphs constructed from an SCFG rule have degree at most 6. This is because each vertex in the graph is a member of two cliques corresponding to the two nonterminals meeting at a given boundary point in the rule. Each clique has three other vertices, for a total degree of 6. Any NP-hardness result for dependency graph of SCFG rules would imply NP-hardness for general graphs of degree 6. ∎

6. Conclusion

In the context of machine translation, the problem of synchronous parsing with an SCFG corresponds to the problem of analyzing parallel text into grammar derivations, often as part of learning a translation model with Expectation Maximization or related algorithms. The synchronous parsing problem also applies to decoding with an SCFG and an integrated n-gram language model.

Parsing with an SCFG requires time (and space) polynomials in the sentence length, but with the degree of the polynomial depending on the specific grammar. A loose time upper bound obtained using dynamic programming techniques is $\mathcal{O}(n^{2r+2})$, where n is the sentence length and r is the maximum rule length, that is, the maximum number of linked nonterminals in the right-hand side of a rule. Gildea and Štefankovič (2007) show a lower bound of $\Omega(n^{cr})$ for some constant c, meaning that the exponent grows linearly with the rule length. In this article, we show that even finding this exponent is itself computationally difficult. For space complexity, finding the best parsing strategy is NP-hard. For time complexity, finding the best parsing strategy would require progress on approximation algorithms for the treewidth of general graphs.

To our knowledge, the use of the notion of carving width of a graph in connection with the space complexity of dynamic programming parsing is novel to this article. In previous work, Crescenzi et al. (2015) used the notion of graph cutwidth to investigate space complexity for SCFGs restricted to *linear* parsing strategies, that is, strategies that add only one nonterminal at a time. However, graph cutwidth does not extend to the general binary parsing strategies that we consider in this article.

The connection of space complexity to carving width means that we can use existing linear-time algorithms for computing the tree layout of graphs of bounded carving width (Thilikos, Serna, and Bodlaender 2000), and approximation algorithms for carving width such as the $\mathcal{O}(\log n)$-factor approximation algorithm of Khuller, Raghavachari, and Young (1994). The connection of time complexity to treewidth means that we can apply algorithms for treewidth of general graphs that tend to find optimal results quickly in practice, such as the branch-and-bound algorithm of Gogate and Dechter (2004). We can also apply linear time algorithms for graphs of bounded

treewidth (Bodlaender 1996), and approximation algorithms such as the $\sqrt{\log(k)}$-factor approximation algorithm of Feige, Hajiaghayi, and Lee (2005).

Because SCFG is an instance of the general class of linear context-free rewriting systems (LCFRSs) (Vijay-Shankar, Weir, and Joshi 1987), our hardness results also apply to the latter class. Therefore our results generalize those of Crescenzi et al. (2011), who show NP-hardness for optimizing space and time complexity for LCFRSs, again with the restriction for linear parsing strategies.

Viewing SCFG as an instance of LCFRS also allows connecting the parsing optimization problems investigated in this article to the rule factorization problem for SCFGs, investigated by Huang et al. (2009). Rule factorization is the problem of replacing an individual rule with a large size with several rules of smaller size, in a way that the language generated by the overall grammar is preserved. (The algorithm for casting a CFG in Chomsky normal form is an example of rule factorization.) As already mentioned in the Introduction, it is known that SCFGs do not admit any canonical binary form (Aho and Ullman 1972). This means that, when we attempt to factorize a SCFG rule into "smaller pieces," some of the resulting pieces might no longer be SCFG rules, because they span more than two substrings; see again Figure 3. However, when viewing a SCFG as a LCFRS, factorization is always possible, and the parsing strategy trees defined in Section 2.3 directly provide a binary factorization of a SCFG into a LCFRS. Under this view, the problems investigated in this article are related to the problem of detecting rule factorizations for certain LCFRS, where we require that the factorized LCFRS are space or time optimal when used in standard algorithms for parsing based on dynamic programming. Rule factorization for general LCFRS has been investigated by Gómez-Rodríguez et al. (2009) and Gildea (2010).

As a final remark, if a strategy of space or time $\mathcal{O}(n^k)$ exists for a fixed k, it can be found in space or time $\mathcal{O}(r^k)$, respectively, by parsing the SCFG rule's right-hand itself, represented as a string of nonterminals, with a grammar representing all possible parsing strategies. Such a grammar can be constructed by instantiating all possible LCFRS rules of space or time complexity $\mathcal{O}(n^k)$. If a parse of the SCFG rule's right-hand side is found, the resulting parse tree can be used as a parsing strategy for the original rule.

References

Ahlswede, R. and S. L. Bezrukov. 1995. Edge isoperimetric theorems for integer point arrays. *Applied Mathematics Letters*, 8(2):75–80.

Aho, Alfred V. and Jeffery D. Ullman. 1969. Syntax directed translations and the pushdown assembler. *Journal of Computer and System Sciences*, 3:37–56.

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

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.

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

Chiang, David. 2007. Hierarchical phrase-based translation. *Computational Linguistics*, 33(2):201–228.

Crescenzi, Pierluigi, Daniel Gildea, Andrea Marino, Gianluca Rossi, and Giorgio Satta. 2011. Optimal head-driven parsing complexity for linear context-free rewriting systems. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL-11)*, pages 450–459. Portland, OR.

Crescenzi, Pierluigi, Daniel Gildea, Andrea Marino, Gianluca Rossi, and Giorgio Satta.

2015. Synchronous context-free grammars
and optimal linear parsing strategies.
Journal of Computer and System Sciences,
81(7):1333–1356.

Feige, Uriel, MohammadTaghi Hajiaghayi,
and James R. Lee. 2005. Improved
approximation algorithms for
minimum-weight vertex separators. In
*STOC '05: Proceedings of the Thirty-Seventh
Annual ACM Symposium on Theory of
Computing*, pages 563–572. Baltimore,
MD.

Gildea, Daniel. 2010. Optimal parsing
strategies for linear context-free rewriting
systems. In *Proceedings of the 2010 Meeting
of the North American Chapter of the
Association for Computational Linguistics
(NAACL-10)*, pages 769–776. Los Angeles,
CA.

Gildea, Daniel. 2011. Grammar factorization
by tree decomposition. *Computational
Linguistics*, 37(1):231–248.

Gildea, Daniel and Daniel Štefankovič. 2007.
Worst-case synchronous grammar rules. In
*Proceedings of the 2007 Meeting of the North
American Chapter of the Association for
Computational Linguistics (NAACL-07)*,
pages 147–154. Rochester, NY.

Gogate, Vibhav and Rina Dechter. 2004. A
complete anytime algorithm for treewidth.
In *Proceedings of the 20th Conference on
Uncertainty in Artificial Intelligence (UAI)*,
pages 201–208.

Gómez-Rodríguez, Carlos, Marco
Kuhlmann, Giorgio Satta, and David Weir.
2009. Optimal reduction of rule length in
linear context-free rewriting systems. In
*Proceedings of the 2009 Meeting of the
North American Chapter of the Association for
Computational Linguistics (NAACL-09)*,
pages 539–547. Boulder, CO.

Huang, Liang, Hao Zhang, Daniel Gildea,
and Kevin Knight. 2009. Binarization of
synchronous context-free grammars.
Computational Linguistics, 35(4):559–595.

Kay, M. 1980. Algorithm schemata
and data structures in syntactic
processing. Technical report CSL-80,
Xerox Palo Alto Research Center,
Palo Alto, CA.

Khuller, Samir, Balaji Raghavachari, and
Neal Young. 1994. Designing
multi-commodity flow trees. *Information
Processing Letters*, 50:49–55.

Kozawa, Kyohei, Yota Otachi, and Koichi
Yamazaki. 2010. The carving-width of
generalized hypercubes. *Discrete
Mathematics*, 310(21):2867–2876.

Lewis, II, P. M. and R. E. Stearns. 1968.
Syntax-directed transduction. *Journal of the
Association for Computing Machinery*,
15(3):465–488.

Rolim, José, Ondrej Sýkora, and Imrich Vrt'o.
1995. Optimal cutwidths and bisection
widths of 2- and 3-dimensional meshes. In
Manfred Nagl, editor, *Graph-Theoretic
Concepts in Computer Science*, volume 1017
of *Lecture Notes in Computer Science*.
Springer, Berlin Heidelberg,
pages 252–264.

Seki, H., T. Matsumura, M. Fujii, and
T. Kasami. 1991. On multiple context-free
grammars. *Theoretical Computer Science*,
88:191–229.

Seymour, P. D. and R. Thomas. 1994. Call
routing and the ratcatcher. *Combinatorica*,
14(2):217–241.

Thilikos, Dimitrios M., Maria J. Serna, and
Hans L. Bodlaender. 2000. Constructive
linear time algorithms for small cutwidth
and carving-width. In *Algorithms and
Computation (ISAAC 2000)*, Taipei.
pages 192–203.

Vijay-Shankar, K., D. L. Weir, and A. K. Joshi.
1987. Characterizing structural
descriptions produced by various
grammatical formalisms. In *Proceedings of
the 25th Annual Conference of the Association
for Computational Linguistics (ACL-87)*,
pages 104–111, Stanford, CA.