

An Algorithm for Hypergraph Completion according to Hyperedge Replacement Grammars

Steffen Mazanek, Sonja Maier, Mark Minas

Universität der Bundeswehr München, Germany,
{`steffen.mazanek|sonja.maier|mark.minas`}@unibw.de

Abstract. The algorithm of Cocke, Younger, and Kasami is a dynamic programming technique well-known from string parsing. It has been adopted to hypergraphs successfully by Lautemann. Therewith, many practically relevant hypergraph languages generated by hyperedge replacement can be parsed in an acceptable time. In this paper we extend this algorithm by hypergraph completion: If necessary, appropriate fresh hyperedges are inserted in order to construct a derivation. The resulting algorithm is reasonably efficient and can be directly used, among other things, for auto-completion in the context of diagram editors.

Keywords: hypergraph completion, hyperedge replacement, parsing

1 Introduction

Hypergraphs are an extension of graphs where edges are allowed to visit an arbitrary number of nodes. A well-known way of describing hypergraph languages are hyperedge replacement grammars HRG [1]. Although restricted in power, this formalism comprises several beneficial properties: It is context-free and still quite powerful. Grammars are comprehensible, and reasonably efficient parsers can be defined for practical languages. In general, parsing is NP-complete though.

Lautemann has provided a detailed discussion of the complexity of hyperedge replacement [2]. He also has suggested a hypergraph parsing algorithm straightforwardly adopting the dynamic programming approach proposed by Cocke, Younger, and Kasami CYK [3] for string parsing. Given a string $s = a_1 \dots a_n$, the CYK algorithm computes a table where the cell in row i and column j contains derivation trees that derive the substring $a_i \dots a_{i+j-1}$. This table can be computed bottom up by joining two appropriate entries at a time – provided that the grammar is in Chomsky normalform CNF, which is no restriction.

An extended version of Lautemann’s CYK-style algorithm for hypergraphs has been proposed by the third author and incorporated in the diagram editor generator DIAGEN [4]. In analogy to the string setting, HRGs are transformed to a kind of CNF first. However, the parsing algorithm does not need to compute a table but rather n layers where n is the number of hyperedges in the hypergraph. Thereby, layer k is computed by combining two “compatible” derivation trees from layers i and j at a time where $i + j = k$.

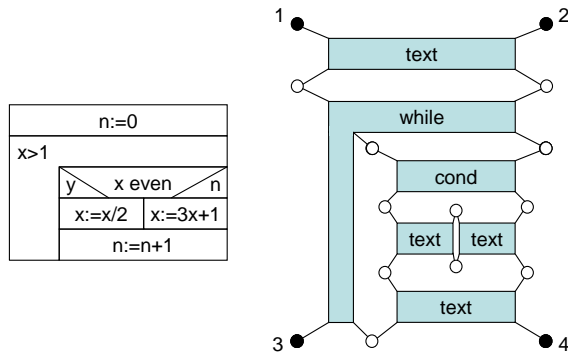


Fig. 1. Hypergraphs as a model for diagrams

Diagram editors are an important area of application for hypergraph parsing since hypergraphs have appeared to be well-suited as a model for diagrams. For instance, in *DIAGEN* the abstract syntax of a diagram language is defined using HRGs, and the parser checks which parts of a freely drawn diagram are correct. In Fig. 1 an example Nassi-Shneiderman-Diagram NSD and a corresponding hypergraph model are shown. Hyperedges are represented by boxes with the particular label inside. Nodes are represented as circles; the so-called external ones as black dots. Lines (tentacles) indicate that a hyperedge visits a node. The hypergraph language of NSDs can be defined using an HRG as we see later.

In this paper we propose an algorithm for hypergraph completion with respect to HRGs. Hypergraph completion can be used, among other things, as a powerful and flexible base for diagram completion. Indeed, *content assist* in diagram editors is just as valuable as conventional content assist as known from modern text editors and integrated development environments. The editor user normally does not only want to be notified when his diagram is incorrect, but is also interested in the particular problem and possible solutions. For instance, the insertion of a simple statement at the right place is already sufficient in order to repair the diagram shown in Fig. 2. Such suggestions are particularly important for free-hand editors like the ones generated with *DIAGEN*. By providing assistance we can effectively combine the advantages of structured and free-hand editing: The user is allowed to draw his diagram with maximal freedom, but guidance is provided if needed.

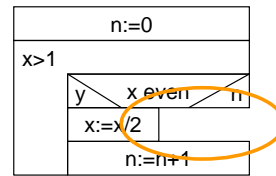


Fig. 2. Incomplete NSD

The information required for some assistance can be gathered by the parser. It is possible to infer places where new hyperedges may be added to complete a given hypergraph, although those might not be uniquely determined. We have discussed a first, logic-based approach in [5]. The proposed framework of *graph parser combinators* follows a top-down approach with backtracking, thus partial results might be computed several times.

In this paper we substantially improve efficiency by using dynamic programming techniques. We basically extend Lautemann’s CYK-style algorithm to support the computation of hypergraph completions. Our key idea is to pretend a (limited) number of hyperedges while parsing. Therefore, several fresh hyperedges are introduced initially, which visit fresh, special nodes. Later, these nodes can be glued with nodes already occurring in the input hypergraph.

We proceed as follows: First, we introduce hypergraphs and HRGs in Sect. 2 using NSDs as a running example. Thereafter, we describe our parser and how it computes so-called complement hypergraphs (Sect. 3). This is the main contribution of this paper. In Sect. 4 we discuss related work. Finally, we sketch future prospects and conclude the paper (Sect. 5).

2 Hypergraphs and Hyperedge Replacement Grammars

In this section the formal basics are introduced. Most definitions are close to [1] and [2]. We just recapitulate them to make this paper self-contained.

Let C be an arbitrary, but fixed set of *labels* and let $type: C \rightarrow \mathbb{N}$ be a *typing* function for C . Let \mathcal{V} denote a universe of nodes. A *hypergraph* H over C is a tuple $(V_H, E_H, att_H, lab_H, ext_H)$ where $V_H \subset \mathcal{V}$ is a finite set of *nodes*, E_H is a finite set of *hyperedges*, $att_H: E_H \rightarrow V_H^*$ is a mapping assigning a sequence of pairwise distinct *attachment nodes* $att_H(e)$ to each $e \in E_H$, $lab_H: E_H \rightarrow C$ is a mapping that *labels* each hyperedge such that $type(lab_H(e)) = |att_H(e)|$ (length of sequence), and $ext_H \in V_H^*$ is a sequence of pairwise distinct *external nodes* (in pictures numbers represent a node’s position in ext). We further define $type(H) := |ext_H|$, $|H| := |E_H|$, and $H_0 := (V_H, E_H, att_H, lab_H, \epsilon)$ (the underlying *type-0* hypergraph of a hypergraph H). We denote the empty hypergraph as \emptyset_G . The set of all hypergraphs over $\Sigma \subseteq C$ is denoted by \mathcal{H}_Σ . We occasionally call hypergraphs just graphs and hyperedges edges.

A hypergraph H is called *elementary* if it is induced by a single edge e , i.e., $E_H = \{e\}$, $V_H = [att_H(e)]^1$, and $ext_H = att_H(e)$. In this case, we define $lab(H) := lab_H(e)$ and $edge(H) := e$. Given a set of hypergraphs \mathcal{H} , the set of elementary hypergraphs in \mathcal{H} is denoted by $elem(\mathcal{H})$.

Given hypergraphs $H, H' \in \mathcal{H}_C$ with $E_H \cap E_{H'} = \emptyset$, $e \in E_H$ such that $V_H \cap V_{H'} = [att_H(e)]$ and $ext_{H'} = att_H(e)$, the hypergraph $H[e/H']$ resulting from the replacement of e by H' is defined as $H[e/H'] := (V_H \cup V_{H'}, (E_H \setminus \{e\}) \cup E_{H'}, att_H \cup att_{H'}, lab_H \cup lab_{H'}, ext_H)$. Let $B \subseteq E_H$ be a set of hyperedges to be replaced and let $repl: B \rightarrow \mathcal{H}_C$ be a mapping such that $H[e/repl(e)]$ is defined for all $e \in B$. We denote the replacement of all edges contained in B by $H[repl]$ (the order of their replacement does not matter [1]).

Let $H, H' \in \mathcal{H}_C$. Then H is a *sub-hypergraph* of H' , denoted $H \subseteq H'$, if $V_H \subseteq V_{H'}$, $E_H \subseteq E_{H'}$, $att_H(e) = att_{H'}(e)$, and $lab_H(e) = lab_{H'}(e)$ for all $e \in E_H$. The *ext-union* of H and H' , denoted $H \cup_{ext} H'$ where $ext \in (V_H \cup V_{H'})^*$, is defined as $(V_H \cup V_{H'}, E_H \cup E_{H'}, att_H \cup att_{H'}, lab_H \cup lab_{H'}, ext)$ provided

¹ $[a_1 \dots a_n] := \{a_1, \dots, a_n\}$

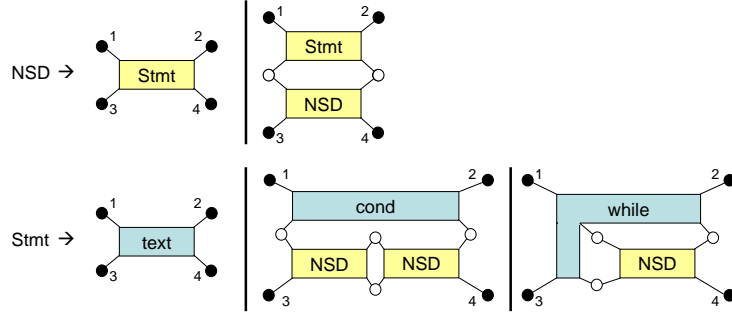


Fig. 3. Productions P_{NSD} of HRG G_{NSD}

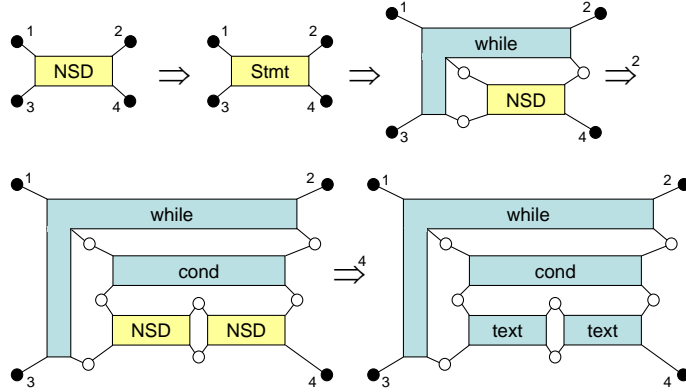


Fig. 4. A possible derivation wrt G_{NSD}

$E_H \cap E_{H'} = \emptyset$. Two hypergraphs H and H' are isomorphic, denoted $H \cong H'$, if there are bijections $\phi_V : V_H \rightarrow V_{H'}$ and $\phi_E : E_H \rightarrow E_{H'}$ such that for all $e \in E_H$ $\text{lab}_{H'}(\phi_E(e)) = \text{lab}_H(e)$, $\text{att}_{H'}(\phi_E(e)) = \phi_V^*(\text{att}_H(e))$ and $\text{ext}_{H'} = \phi_V^*(\text{ext}_H)$.²

A production $A \rightarrow R$ over $N \subseteq C$ consists of a label $A \in N$ and a hypergraph $R \in \mathcal{H}_C$ such that $\text{type}(A) = \text{type}(R)$. Let P be a set of productions, $H \in \mathcal{H}_C$, $e \in E_H$, $(\text{lab}_H(e) \rightarrow R) \in P$, $R' \in \mathcal{H}_C$ such that $R' \cong R$ and $H[e/R']$ is defined. Then H directly derives $H' = H[e/R']$, denoted $H \Rightarrow_P H'$.

A hyperedge replacement grammar is a system $G = (N, T, P, S)$ where $N \subset C$ is a set of nonterminals, $T \subset C$ with $T \cap N = \emptyset$ is a set of terminals, P is a finite set of productions over N , and $S \in N$ is the start symbol. Let $\mathcal{L}_A(G) := \{H \in \mathcal{H}_T \mid \exists L \in \text{elem}(\mathcal{H}_{\{A\}}) : L \Rightarrow_P^* H\}$. The hypergraph language $\mathcal{L}(G)$ generated by G is defined as $\mathcal{L}(G) := \mathcal{L}_S(G)$.

As an example consider the grammar of NSD graphs $G_{\text{NSD}} = (\{\text{NSD}, \text{Stmt}\}, \{\text{text}, \text{cond}, \text{while}\}, P_{\text{NSD}}, \text{NSD})$ where the set of productions P_{NSD} is shown in

² $f^*(a_1 \dots a_n) := f(a_1) \dots f(a_n)$

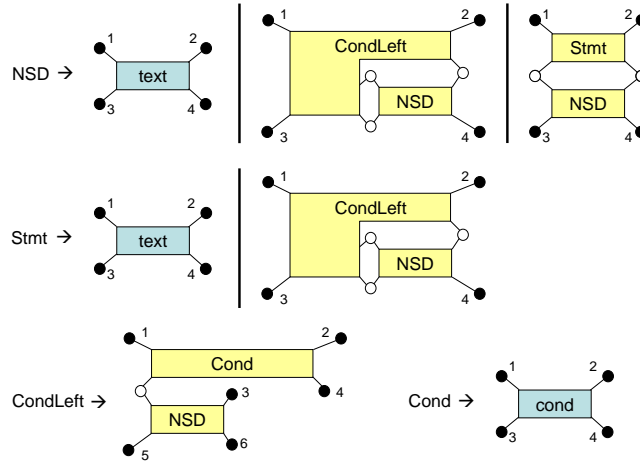


Fig. 5. Chomsky normalform of Fig. 3 (without “while”)

Fig. 3. A possible derivation is given in Fig. 4. Note, that in the following we omit the “while”-production for the sake of brevity.

In the string setting the CYK-algorithm requires grammars to be in so-called Chomsky normalform CNF. This is no restriction, since every context-free grammar (that does not generate the empty string) can be transformed to a grammar that defines the very same language and is in CNF. A similar notion can be defined for HRGs:

Definition 1 (Chomsky normalform). *An HRG is in Chomsky normalform CNF, iff for every production $A \rightarrow R$ holds: R does not contain isolated nodes, and either $R \in \mathcal{H}_T \wedge |R| = 1$ (terminal production) or $R \in \mathcal{H}_N \wedge |R| = 2$ (expansion production).*

Every HRG whose language does not contain hypergraphs with isolated nodes or the empty graph \emptyset_G can be transformed to an equivalent grammar in CNF. A constructive proof of this proposition is given in [6]. The productions of an HRG in CNF equivalent to G_{NSD} (without “while”) are shown in Fig. 5.³ Note, that isolated nodes do not occur in the context of visual language specifications. However, if they cannot be avoided in a particular situation, it might be sufficient to add unary dummy edges.

³ Basically, productions with more than two hyperedges at their right-hand side are split successively (therefore, in our example the new nonterminal “CondLeft” and the corresponding production are added). Where necessary, special nonterminal labels \hat{l} are introduced that only derive an elementary graph labeled l , see, e.g., “Cond”. And finally, so-called chain productions like the derivation of a single “Stmt” nonterminal from “NSD” are eliminated by adding the productions over “Stmt” also to “NSD”. This transformation is very similar to the corresponding transformation in the string setting and can be performed automatically (as realized in DIAGEN).

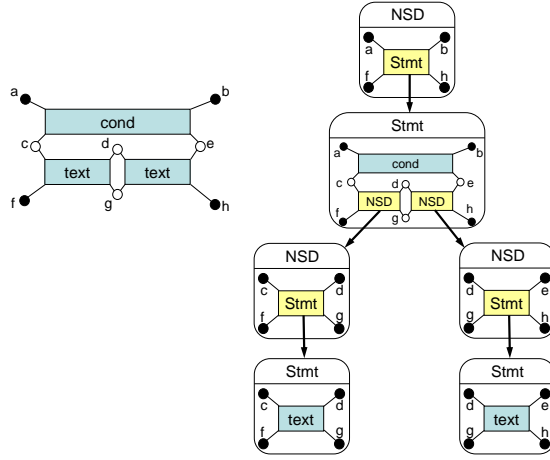


Fig. 6. A hypergraph and its derivation tree wrt G_{NSD}

Since HRGs are context-free (cf. *context-freeness lemma* in [1]) the notion of a *derivation tree* is meaningful. Let $G = (N, T, P, S)$ be an HRG. The set of derivation trees $\text{TREE}(G)$ is recursively defined: Consider a triple $t = (L, R, \text{branch})$ where $L \in \text{elem}(\mathcal{H}_N)$, $R \in \mathcal{H}_C$, $L \implies_P R$, $\text{branch} : E_R^N \rightarrow \text{TREE}(G)$ ⁴. Let $\text{root}(t) := L$. The triple t is in $\text{TREE}(G)$ iff $\text{lab}_R(e) = \text{lab}(\text{root}(\text{branch}(e)))$ for all $e \in E_R^N$ and $\text{result}(t) := R[\{e \mapsto \text{result}(\text{branch}(e)) \mid e \in E_R^N\}]$ (the graph spanned by this tree) is defined. Note, that according to this definition the leaves of a derivation tree are triples where $R \in \mathcal{H}_T$.

An example graph and its derivation tree are shown in Fig. 6. The arrows represent the mapping determined by *branch*. The elementary *root* graphs L can be represented by their labels $\text{lab}(L)$, since the nodes visited by $\text{edge}(L)$ are just the external nodes of R . Numbers of external nodes are omitted. All nodes are marked with letters to make them distinguishable. The notions derivation and derivation tree are indeed equivalent as, e.g., proven in [1].

Next, we introduce the concept of a *complement hypergraph*. Informally, this is a graph such that its union with the given input graph can be derived from a particular start graph.

Definition 2 (complement hypergraph). *Given an HRG $G = (N, T, P, S)$, type-0 hypergraphs $H, H_c \in \mathcal{H}_T$, and $L \in \text{elem}(\mathcal{H}_N)$. H_c is a complement hypergraph of H with respect to G and L iff $E_{H_c} \cap E_H = \emptyset$ and $L \implies_P^* H \cup_{\text{ext}_L} H_c$.*

Note, that we do not assume anything about the correctness of the given graph H wrt G . If H is incorrect, H_c is a completion. Otherwise, it is a correctness-preserving extension of H , just as needed, among other things, for the realization of situation-dependent structured editing operations in diagram editors.

⁴ $E_R^N := \{e \in E_R \mid \text{lab}_R(e) \in N\}$

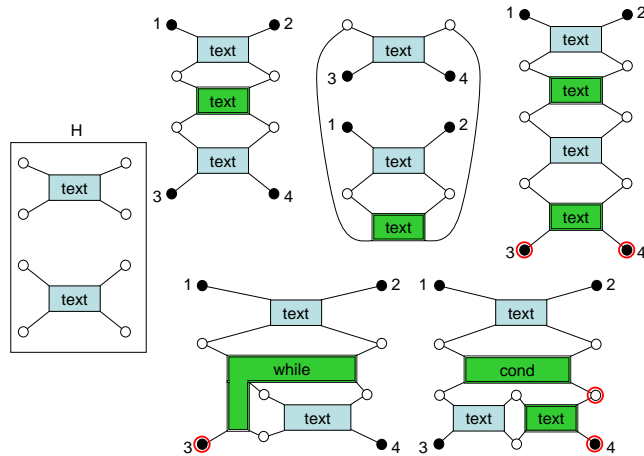


Fig. 7. Some complement graphs of the given graph H

This notion is illustrated by example in Fig. 7. Given the *type-0* hypergraph H surrounded by the box. In the figure several complement hypergraphs of H are shown (in union with H) – each with respect to the elementary graph labeled NSD whose external nodes are just the ones marked in the particular image. Note, that our definition does not relate V_H and V_{H_c} . Indeed those sets may either overlap or be disjoint. Nodes of the complement graphs that do not already belong to the original hypergraph H are surrounded by an extra circle in the figure. In general, the number and size of complement graphs is not restricted, although a practical implementation surely has to impose meaningful bounds.

3 A CYK-style Complementing Parser

Given a *type-0* hypergraph H , a conventional parser analyzes H with respect to an HRG G . If possible, a sequence of external nodes is established such that the resulting graph is in $\mathcal{L}(G)$. The corresponding derivation tree can be constructed, e.g., by using dynamic programming similar to the CYK algorithm. So the hypergraph parser of the DIAGEN system computes $n = |H|$ layers. In layer k , $1 \leq k \leq n$, all derivation trees are contained whose *result* is a graph $H' \subseteq H$ such that $|H'| = k$. Since HRGs are transformed to CNF in advance, for $k > 1$ layer k can be computed by combining two derivation trees from layers i and j at a time where $i + j = k$. Thereby, expansion productions are applied reversely.

The complementing parser proposed in this section does not only check if H is a member of the language. It also computes complement hypergraphs up to a particular size *max*. If the parameter *max* is set to zero it simply is a conventional parser. Our key idea is to introduce fresh edges that can be embedded into the input graph in a flexible manner. If these edges are actually used in a particular derivation tree they constitute the complement graph. For every terminal symbol

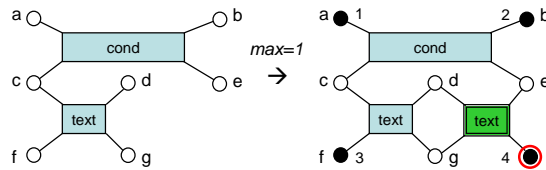


Fig. 8. Example input/output of the algorithm

t of the language, max fresh edges with label t are introduced. Each of these edges visits $type(t)$ nodes, which are also fresh and special in the sense, that – in the process of parsing – they might be identified with nodes from H or even with other fresh nodes (similar to logic variables).

Before we define the algorithm more formally we provide an example run with $max = 1$ to clarify its basic principle. Fig. 8 shows a possible input hypergraph and, resulting from the algorithm, its union with the only complement graph wrt NSD of size up to 1. The corresponding external nodes are also marked.

Fig. 9 illustrates how the layers are filled according to this example. The number enclosed by a circle in the upper right of a derivation tree indicates, how many of the fresh edges have already been used in this derivation. The capital letters A, B, C and D are just shorthands for the particular trees to avoid cluttering the figure by too many arrows. We further simplify the figure by joining derivation trees that only differ in the labels of their *root* graphs. Those mainly appear due to the elimination of chain productions. In this case, trees are marked with all possible *root* labels, here NSD and Stmt.

Furthermore, all “imaginable” derivation trees in layer 2 are shown, although most of them are invalid for some reason and, thus, disregarded by the parser. Such invalid trees are shaded in the figure and the particular problem is marked with a lightning symbol. When constructing derivation trees, we have to ensure that at most max of the fresh edges are used at a time. The derivation tree in the lower right of layer two, for instance, consumes two fresh edges, which is prevented by the restriction $max=1$. The others violate the so-called gluing condition. Here, a node is reduced, i.e., it is non-external within the right-hand side of an “instantiated” production, although there still is an edge in the remaining graph visiting this node. It is important to filter invalid derivation trees as soon as possible to reduce the number of combinations in the layers above.

We compute an equivalence relation \sim between nodes to realize the gluing of fresh nodes with nodes actually occurring in the input graph. The “significant” subset of this relation is shown in the figure. A simple mapping indeed is not sufficient, since an arbitrary number of fresh nodes may coincide.

Complete derivation trees are those consuming the whole input graph. They are surrounded by thicker lines in the figure and can, of course, only occur in the layers between $|H|$ and $|H| + max$. In the example, there is only one tree with complete coverage of the input graph whose *root* is labeled with the start symbol NSD at the same time: the one whose *result* is shown in Fig. 8.

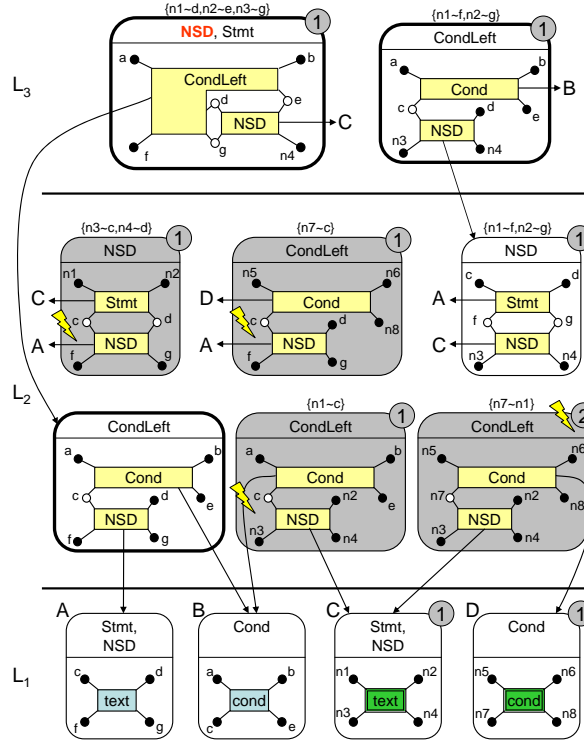


Fig. 9. Illustration of the CYK-style graph parser with completion

Next, we define the parsing algorithm more formally. Let $G = (N, T, P, S)$ be a HRG in CNF, $H \in \mathcal{H}_T$ a *type-0* hypergraph, and $max \in \mathbb{N}$. The parser successively constructs layers $L_i \subseteq \mathcal{H}_T \times \mathcal{H}_T \times 2^{\mathcal{V} \times \mathcal{V}} \times \text{TREE}(G)$ for $1 \leq i \leq |H| + max$. We first provide a lemma which states the properties holding for the elements of the computed layers. This simplifies the understanding of the algorithm defined afterwards.

Lemma 1. *The following properties hold for the elements $(H', H_c, \sim, t) \in L_i$ if we identify nodes equivalent wrt $\text{equi}(\sim)$ ⁵:*

1. $|H'| + |H_c| = i$,
2. $H' \subseteq H$,
3. $\text{result}(t) = H' \cup_{\text{ext, result}(t)} H_c$,
4. $|H_c| \leq max$,
5. H_c is a complement hypergraph of H' with respect to G and $\text{root}(t)$.

⁵ $\text{equi}(\sim) := (\sim \cup \sim^T)^*$ denotes the smallest equivalence relation containing \sim . Formally, the identification of equivalent nodes means to deal with the corresponding quotient graph (whose nodes actually are equivalence classes of nodes), but to avoid cluttering we just assume the identification of equivalent nodes implicitly.

A proof sketch of this lemma is given in Appendix A. After processing the layers we are mainly interested in entries (H', H_c, \sim, t) where $lab(root(t)) = S$ and $|H'| = |H|$, i.e., the whole input graph is covered. To simplify the definitions of the layers let us define an auxiliary predicate for ensuring the gluing condition:

$$\text{For a graph } R, \quad gc_H(R) :\Leftrightarrow \forall e \in E_H \setminus E_R : [att_H(e)] \cap V_R \subseteq [ext_R]$$

Thus, $gc_H(R)$ holds if and only if no edge of H that is not already in R visits a non-external node of R . Next, we define the layers recursively. Layer 1 only contains derivation trees resulting from the reverse-application of terminal productions. Thereby, either a single, terminal edge of H is derived, or one of $max \cdot |T|$ fresh edges. This possibly large number of fresh edges is necessary, since edges should keep their label and at this early stage we cannot know how many edges with a particular label we will eventually need:

$$\begin{aligned} L_1 := & \{ (R_0, \emptyset_G, \emptyset, (L, R, \emptyset)) \mid \\ & R \subseteq H, |R| = 1, L \in elem(\mathcal{H}_N), L \Longrightarrow_P R, gc_H(R) \} \\ \cup & \{ (\emptyset_G, R_0, \emptyset, (L, R, \emptyset)) \mid t \in T, k \in \{1, \dots, max\}, \\ & e_k \text{ fresh edge, } ns_k \text{ sequence of } type(t) \text{ fresh nodes,} \\ & L \in elem(\mathcal{H}_N), R = ([ns_k], \{e_k\}, \{e_k \mapsto ns_k\}, \{e_k \mapsto t\}, ext_L), L \Longrightarrow_P R \} \end{aligned}$$

As illustrated in Fig. 9, from layer 2 on the derivation trees are composed by combining two compatible derivation trees t_1 and t_2 of lower layers at a time by reverse-applying expansion productions. Thus, layer $L_i, i > 1$ is constructed from already computed layers L_j and L_{i-j} where $j < i$. For $1 < i \leq |H| + max$ we define:

$$L_i := \bigcup_{j=1}^{\lfloor i/2 \rfloor} (L_j \oplus L_{i-j})$$

Thereby, the combination of two sets $M, N \subseteq \mathcal{H}_T \times \mathcal{H}_T \times 2^{\mathcal{V} \times \mathcal{V}} \times \text{TREE}(G)$ is defined as follows:

$$\begin{aligned} M \oplus N := & \{ (H'_1 \cup_\epsilon H'_2, H_{1_c} \cup_\epsilon H_{2_c}, \sim_n, (L, R, \{e_1 \mapsto t_1, e_2 \mapsto t_2\})) \mid \\ & (H'_1, H_{1_c}, \sim_1, t_1) \in M, (H'_2, H_{2_c}, \sim_2, t_2) \in N, \\ & E_{H'_1} \cap E_{H'_2} = \emptyset, E_{H_{1_c}} \cap E_{H_{2_c}} = \emptyset, |H_{1_c}| + |H_{2_c}| \leq max, \\ & \text{let } L_k := root(t_k), e_k := edge(L_k) \text{ for } k \in \{1, 2\}, \\ & \exists \sim_n \text{ minimal relation in } 2^{\mathcal{V} \times \mathcal{V}} \text{ such that} \\ & \sim_1 \subseteq \sim_n, \sim_2 \subseteq \sim_n, \sim := equi(\sim_n), preserves_{V_H}(\sim), \\ & \text{when identifying nodes equivalent wrt to } \sim \\ & \exists L \in elem(\mathcal{H}_N), R := L_1 \cup_{ext_L} L_2, L \Longrightarrow_P R, gc_H(R) \} \end{aligned}$$

When combining derivation trees the fresh nodes can be glued to other nodes. For this purpose an equivalence relation between nodes is established such that

the union of the roots of t_1 and t_2 is isomorphic to the right-hand side of the particular production. It must not happen, however, that nodes of the input graph V_H are identified among each other. Rather their identities have to be preserved. This restriction is ensured by using the predicate $preserves_V(\sim) :\Leftrightarrow \forall n_1, n_2 \in V : n_1 \sim n_2 \Rightarrow n_1 = n_2$. Thus, the relation $\{(n_1, n_2) \in \sim \mid n_1, n_2 \in V_H\}$ has to be the identity. Since the layers are recursively defined Lemma 1 now can be proven by induction as sketched in Appendix A.

Discussion

This algorithm also ensures that indeed all *structurally different* complement graphs up to size max are computed, because all possible embeddings of up to max arbitrarily labeled, fresh edges into H are constructed. At the end, those equivalence classes of nodes not containing a node of the input graph can be considered as new nodes contributed by a complement graph.

Performance Although the algorithm is correct and complete, it suffers from an inherent problem. If the bound max is increased, we get a lot of redundant derivation trees, since new derivation trees in layer 1 (where fresh edges have the same label) can be embedded at different places *interchangeably*. Unsurprisingly, this effect has a negative impact on the performance of the algorithm.

It is possible to avoid this problem though. In our current implementation, layer 1 only contains derivation trees for edges originally occurring in the input graph. In addition, for each terminal production a special leaf derivation tree is constructed that can be *cloned* if required. This approach is more syntax-driven and, thus, yields only structurally different solutions. Therefore, it can even be used as a reasonably efficient language generator. However, the formulation of this paper is less technical. It has been preferred for the sake of presentation.

Indeed our current implementation has turned out to be sufficiently efficient even for interactive applications. In Fig. 10 we provide performance data for

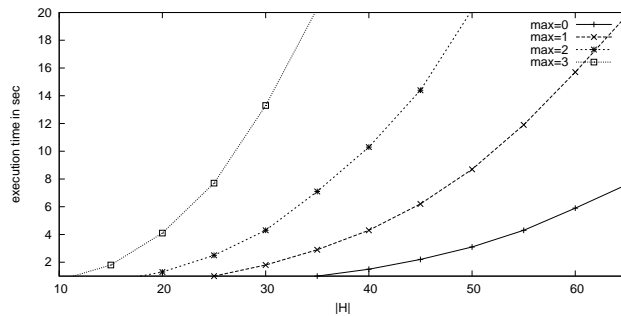


Fig. 10. Execution time of our implementation applied to NSD graphs with different values of max on standard hardware

different values of max . The input graphs have been simple sequences of statements, i.e., $|H|$ is the length of the sequence. Thus, the corresponding completions are extensions of correct graphs (incorrect graphs show a similar behavior, but are more difficult to construct in a homogeneous way). The algorithm shows a polynomial runtime behavior. Note, that our implementation has not been optimized with respect to performance yet. Further enhancements can be achieved by a prior analysis of the grammar such that compatible derivation trees can be found more efficiently. We also work on an incremental version of the algorithm where additional fresh edges can be efficiently incorporated on demand.

There is one factor particularly known for its strong impact on the performance: the degree of connectedness. We already know that parsing performance suffers if graphs of a language are highly disconnected, cf. [1, 2]. This effect unfortunately becomes worse with our approach, since the gluing condition cannot be used as effectively anymore to exclude derivation trees at an early stage.

Gluing nodes Per definition the algorithm presented in this section preserves the nodes of the input graph. However, we have noted that it is sometimes convenient to relax this condition. Incorrect input graphs often can already be corrected by gluing some nodes appropriately.

For instance, reconsider the example graph given in Fig. 7. It can be corrected by inserting an artificial extra edge. However, there also is a more lightweight way: The two isolated edges can be joined together by gluing some nodes. There are two ways to do this both shown in Fig. 11. Support for this kind of repair action can be achieved

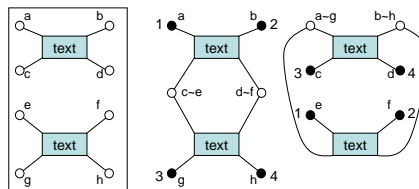


Fig. 11. Gluing original nodes

with a very little adaptation of the given algorithm. However, formally this approach would have to rely on a broader notion of completion.

4 Related Work

To the best of our knowledge graph completion wrt hyperedge replacement grammars has not been considered yet. Due to their logic-based approach *graph parser combinators* [5] provide some support for completion as a nice side effect. However, from a performance point of view this framework cannot be used for practical, interactive applications. In contrast, the algorithm presented in this paper does not suffer from this problem.

In the more general context of graphs, several approaches exist to error correction and inexact matching, respectively. Most practical approaches propose either particular restrictions on the graph grammar formalism, incorporate application-specific knowledge or even make use of heuristics [7] in order to solve the particular problem with an acceptable performance.

For instance, Kaul proposed a fast parser for the computation of a minimum error distance [8]. It depends on so-called *precedence graph grammars* and runs in $\mathcal{O}(n^3)$. Sánchez et al. add special error correcting rules to the graph grammar [9], which they define in terms of *region adjacency graphs*. Their approach appears to be beneficial in the domain of texture symbol recognition.

In the context of diagram editors, the grammar-based system VLDESK [10] provides support for so-called *symbol prompting*. Here, the parsing table of a YACC-based parser is exploited to extract information about possible contexts of a particular symbol. Following this approach, local suggestions can be made that may, however, be misleading from an overall perspective (although local context in general can provide more suggestions and can be computed more efficiently). In the tool AToM³ [11] model completion can be realized by solving a *constraint-logic program* that can be generated from the metamodel of the particular language [12].

5 Conclusion and Further Work

We have presented an approach for hypergraph completion with respect to hyperedge replacement grammars. The practicability of the proposed algorithm has been validated by incorporating it in the diagram editor generator DIAGEN. The algorithm appears to be widely applicable and sufficiently efficient even for interactive applications. It can be directly used for the realization of content assist in the domain of diagram editors.

Our algorithm is quite beneficial to correct errors in a graph. However, we do not require the given graph to be incomplete. Whereas incomplete graphs can be completed, we can further compute powerful structured editing operations from the complement graphs of already complete graphs. In both DIAGEN and also TIGER [13] complex editing commands can be specified by means of graph transformation rules. While this is a powerful way to specify editing operations, it is also quite tedious and error-prone. With our approach a set of applicable commands, which automatically preserve or even establish the syntactical correctness of the resulting graph/diagram, can be computed on the fly.

We already have specified several, prototypical diagram editors with completion support. Nevertheless, in the future we have to study in depth how hypergraph completions can be translated back to diagram completions in a systematic way. We further have to realize more sophisticated user interaction mechanisms which provide maximal benefit of the different completions.

Our algorithm can be extended in a variety of ways. A quite severe restriction is that it can only be applied to context-free languages. Unfortunately, many graph/diagram languages are not context-free. A restrictive embedding mechanism all practical diagram languages can be defined with has been proposed by the third author to address this issue. The adopted parser (as incorporated in DIAGEN) is still efficient. In the future we want to investigate how context-sensitive embedding rules can be supported by our parser to further widen its range of applicability.

Acknowledgment

We greatly appreciate the discussions with Birgit Elbl and Wolfram Kahl. Their feedback has been very helpful. Furthermore, the first author is very grateful to Lothar Schmitz for polishing the wording.

References

1. Drewes, F., Habel, A., Kreowski, H.J.: Hyperedge replacement graph grammars. In Rozenberg, G., ed.: *Handbook of Graph Grammars and Computing by Graph Transformation*. Vol. I: Foundations. World Scientific (1997) 95–162
2. Lautemann, C.: The complexity of graph languages generated by hyperedge replacement. *Acta Inf.* **27**(5) (1989) 399–421
3. Kasami, T.: An efficient recognition and syntax analysis algorithm for context free languages. Scientific Report AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachusetts (1965)
4. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming* **44**(2) (2002) 157–180
5. Mazanek, S., Minas, M.: Functional-logic graph parser combinators. In: *Proc. of the 19th Intl. Conference on Rewriting Techniques and Applications*. LNCS, Springer (2008)
6. Minas, M.: Spezifikation und Generierung graphischer Diagrammeditoren. Shaker-Verlag, Aachen (2001) zugl. Habilitationsschrift Universität Erlangen-Nürnberg, 2000.
7. Bengoetxea, E., Pedro Larrañaga, I., Bloch, I., Perchant, A.: Estimation of distribution algorithms: A new evolutionary computation approach for graph matching problems. In: *EMMCVPR '01: Proc. of the Third Intl. Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition*, London, UK, Springer (2001) 454–468
8. Kaul, M.: Specification of error distances for graphs by precedence graph grammars and fast recognition of similarity. In: *WG '86: Proc. of the Intl. Workshop on Graphtheoretic Concepts in Computer Science*, London, UK, Springer (1987) 29–40
9. Sánchez, G., Lladós, J., Tombre, K.: An error-correction graph grammar to recognize texture symbols. In: *GREC '01: Selected Papers from the Fourth Intl. Workshop on Graphics Recognition Algorithms and Applications*, London, UK, Springer (2002) 128–138
10. Costagliola, G., Deufemia, V., Polese, G., Risi, M.: Building syntax-aware editors for visual languages. *Journal of Visual Languages and Computing* **16**(6) (2005) 508–540
11. de Lara, J., Vangheluwe, H.: Atom3: A tool for multi-formalism and meta-modelling. In: *FASE '02: Proc. of the 5th Intl. Conference on Fundamental Approaches to Software Engineering*, London, UK, Springer (2002) 174–188
12. Sen, S., Baudry, B., Vangheluwe, H.: Domain-specific model editors with model completion. In: *Multi-paradigm Modelling Workshop at MoDELS 2007*. (2007)
13. Taentzer, G., Crema, A., Schmutzler, R., Ermel, C.: Generating domain-specific model editors with complex editing commands. In: *Proc. Third Intl. Workshop and Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007)*. (2007)

A Proof sketch of Lemma 1

Proof. By induction on i . For $i = 1$ layer i is defined as the union of two sets. It can be easily checked that both satisfy the given properties.

Induction step: Let $r = (H', H_c, \sim, t) \in L_i$ where $t = (L, R, \text{branch}) \in \text{TREE}(G)$. Then there has to be a j , $1 \leq j \leq \lfloor i/2 \rfloor$, such that $r \in L_j \oplus L_{i-j}$. This means, that there are two triples $(H'_1, H_{1_c}, \sim_1, t_1) \in L_j$, $(H'_2, H_{2_c}, \sim_2, t_2) \in L_{i-j}$ with $E_{H'_1} \cap E_{H'_2} = \emptyset$, $E_{H_{1_c}} \cap E_{H_{2_c}} = \emptyset$, $|H_{1_c}| + |H_{2_c}| \leq \text{max}$, a minimal relation $\sim_n \in 2^{\mathcal{V} \times \mathcal{V}}$, $L \in \text{elem}(\mathcal{H}_N)$ such that $\sim_1 \subseteq \sim_n$, $\sim_2 \subseteq \sim_n$, $\text{preserves}_{V_H}(\sim)$ and, when identifying nodes equivalent wrt \sim , $L \Longrightarrow_P R$ and $\text{gc}_H(R)$. Thereby, $L_k := \text{root}(t_k)$, $e_k := \text{edge}(L_k)$, $k \in \{1, 2\}$, $\sim := \text{equi}(\sim_n)$ and $R := L_1 \cup_{\text{ext}_L} L_2$.

1. $|H'| + |H_c| = i$: follows by induction hypothesis and the fact that only graphs with disjoint edge sets are combined.
2. $H' \subseteq H$: this statement holds, since, by induction hypothesis, both H'_1 and H'_2 are subgraphs of H .
- 3.

$$\begin{aligned}
\text{result}(t) &= R[\{e_k \mapsto \text{result}(\text{branch}(e_k)) \mid k \in \{1, 2\}\}] && \text{(def. result)} \\
&= R[\{e_k \mapsto \text{result}(t_k) \mid k \in \{1, 2\}\}] && \text{(constr. branch)} \\
&= R[\{e_k \mapsto H'_k \cup_{\text{ext}_{\text{result}(t_k)}} H_{k_c} \mid k \in \{1, 2\}\}] && \text{(ind. hypothesis)} \\
&= L_1[\{e_1 \mapsto H'_1 \cup_{\text{ext}_{\text{result}(t_1)}} H_{1_c}\}] \cup_{\text{ext}_L} \\
&\quad L_2[\{e_2 \mapsto H'_2 \cup_{\text{ext}_{\text{result}(t_2)}} H_{2_c}\}] && (e_k \in E_{L_k}, \text{def. } R) \\
&= (H'_1 \cup_{\text{ext}_{\text{result}(t_1)}} H_{1_c}) \cup_{\text{ext}_L} (H'_2 \cup_{\text{ext}_{\text{result}(t_2)}} H_{2_c}) && (L_k \text{ elementary}) \\
&= (H'_1 \cup_{\epsilon} H'_2) \cup_{\text{ext}_L} (H_{1_c} \cup_{\epsilon} H_{2_c}) && \text{(outermost ext)} \\
&= H' \cup_{\text{ext}_L} H_c && \text{(constr. } H', H_c) \\
&= H' \cup_{\text{ext}_{\text{result}(t)}} H_c && (L \Longrightarrow_P R)
\end{aligned}$$

4. $|H_c| \leq \text{max}$: holds by definition of the layer ($|H_c| = |H_{1_c}| + |H_{2_c}| \leq \text{max}$).
5. H_c is complement hypergraph of H' with respect to G and $\text{root}(t)$: We just argue here, since otherwise we would need to formally introduce quotient graphs. The prerequisites for the definition of complement graphs are satisfied. Disjointness of edge sets is maintained while constructing the layers. So the only statement to prove is $\text{root}(t) = L \Longrightarrow_P^* H' \cup_{\text{ext}_L} H_c = \text{result}(t)$. Since by construction t is a proper derivation tree this statement normally holds. However, we have implicitly dealt with equivalence classes of nodes, so that two issues have to be clarified. Firstly, with *preserve* we have prevented nodes of the input graph H to coincide via \sim . Thus, H is isomorphic with its quotient graph. Second, it must not happen that by joining nodes the gluing condition ensured at a particular layer can be hurt afterwards. This cannot happen though, since the gluing condition only prevents non-external nodes of the right-hand side of an instantiated production to occur in the remaining graph. In the following, however, only external nodes are joined due to the minimality of \sim_n .